К. Ю. Богачёв

ОСНОВЫ параллельного программирования

ИЗДАТЕЛЬСТВО БИНОМ

К. Ю. Богачёв

ОСНОВЫ
параллельного программирования

3-е издание (электронное)

Москва
БИНОМ. Лаборатория знаний
2015

УДК 004.65
ББК 32.073
Б73

Богачёв К. Ю.

Б73 Основы параллельного программирования [Электронный ресурс] : учебное пособие / К. Ю. Богачёв. — 3-е изд. (эл.). — Электрон. текстовые дан. (1 файл pdf : 345 с.). — М. : БИНОМ. Лаборатория знаний, 2015. — (Математика). — Систем. требования: Adobe Reader XI ; экран 10".

ISBN 978-5-9963-2995-3

Данная книга представляет собой введение в методы программирования для параллельных ЭВМ.

Основной ее целью является научить читателя самостоятельно разрабатывать максимально эффективные программы для таких компьютеров.

Вопросы распараллеливания конкретных алгоритмов рассмотрены на многочисленных примерах программ на языке С. В основу книги положен курс лекций для студентов механико-математического факультета МГУ им. М. В. Ломоносова.

Для студентов, аспирантов, научных работников, программистов и всех, кто хочет научиться разрабатывать программы для параллельных ЭВМ.

УДК 004.65
ББК 32.073

Деривативное электронное издание на основе печатного аналога: Основы параллельного программирования : учебное пособие / К. Ю. Богачёв. — 2-е изд. — М. : БИНОМ. Лаборатория знаний, 2013. — 342 с. : ил. — (Математика). — ISBN 978-5-9963-1616-8.

ISBN 978-5-9963-2995-3

Оглавление

Предисловие

Параллельные ЭВМ, возникшие преимущественно для высокопроизводительных научных вычислений, получают все более широкое распространение: их можно встретить в небольших научных подразделениях и офисах. Этому способствуют как непрерывное падение цен на них, так и постоянное усложнение решаемых задач. Можно без преувеличения сказать, что параллельные компьютеры сейчас используются или планируются к использованию всеми, кто работает на передовых рубежах науки и техники:

- научными работниками, применяющими ЭВМ для решения **реальных** задач физики, химии, биологии, медицины и других наук, поскольку упрощенные модельные задачи уже рассчитаны на «обычных» ЭВМ, а переход к реальным задачам сопряжен с качественным ростом объема вычислений;
- программистами, разрабатывающими системы управления базами данных (СУБД), разнообразные Internet-серверы запросов (WWW, FTP, DNS и др.) и совместного использования данных (NFS, SMB и др.), автоматизированные системы управления производством (АСУП) и технологическими процессами (АСУТП), поскольку требуется обеспечить обслуживание максимального количества запросов в единицу времени, а сложность самого запроса постоянно растет.

В настоящий момент практически все крупные разрабатываемые программные проекты как научной, так и коммерческой

направленности либо уже содержат поддержку параллельной работы на компьютерах разных типов, либо эта поддержка запланирована на ближайшее время (причем промедление здесь часто вызывает поражение проекта в конкурентной борьбе).

Настоящая книга представляет собой введение в методы программирования параллельных ЭВМ. Основной ее целью является научить читателя самостоятельно разрабатывать максимально эффективные программы для таких компьютеров. Вопросы распараллеливания конкретных алгоритмов рассматриваются на многочисленных примерах. В качестве языка программирования использован язык C, как наиболее распространенный (и, заметим, единственный (не считая своего расширения C++), на котором можно реализовать все приведенные примеры). Программы, посвященные использованию параллелизма процесса и MPI, могут быть легко переписаны на языке FORTRAN-77. Для иллюстрации подпрограмма умножения двух матриц, дающая почти 14-кратное ускорение на одном процессоре, приведена на двух языках: C и FORTRAN-77.

Изложение начинается с изучения параллелизма в работе процессора, оперативной памяти и методов его использования. Затем приводится описание архитектур параллельных ЭВМ и базовых понятий межпроцессного взаимодействия. Для систем с общей памятью подробно рассматриваются два метода программирования: с использованием процессов и использованием задач (threads). Для систем с распределенной памятью рассматривается ставший фактическим стандартом интерфейс MPI. Для указанных систем приведены описания основных функций и примеры их применения. В описаниях намеренно выброшены редко используемые детали, чтобы не пугать читателя большим объемом информации (чем страдают большинство руководств пользователя).

Книга используется в качестве учебного пособия в основном курсе «Практикум на ЭВМ» на механико-математическом факультете МГУ им. М. В. Ломоносова по инициативе и при поддержке академика РАН Н. С. Бахвалова.

Порядок чтения

Книгу можно разделить на 5 достаточно независимых частей:

1. В главах 2, 3, 4 описан параллелизм в работе процессора и оперативной памяти, а также разнообразные приемы, используемые для повышения эффективности их работы. Эту информацию можно использовать для достижения значительного ускорения работы программы даже на однопроцессорном компьютере.
2. В главах 5 и 6 изложены основные понятия, используемые при рассмотрении параллельных программ, а также стандарты на операционные системы UNIX, установленные на подавляющем большинстве параллельных ЭВМ.
3. В главах 7 и 8 описаны основные функции для управления процессами и осуществления межпроцессного взаимодействия. Эти функции можно использовать для запуска многих совместно работающих процессов в системах с общей памятью, а также для разработки параллельного приложения для систем, не поддерживающих задачи (threads).
4. В главах 9 и 10 описаны основные функции для управления задачами (threads) и осуществления межзадачного взаимодействия. Эти функции можно использовать для разработки параллельного приложения в системах с общей памятью.
5. В главе 11 описаны основные функции Message Passing Interface (MPI). Эти функции можно использовать для разработки параллельного приложения в системах как с общей, так и с распределенной памятью.

Части расположены в рекомендуемом порядке чтения. Последние три независимы друг от друга и могут изучаться в произвольной последовательности. Главу 1, адресованную нетерпеливому читателю, при систематическом изучении рекомендуется разбирать по мере ознакомления с материалом основных частей книги.

Книгой можно воспользоваться и в качестве учебника. Для этого в конце приведены программа курса и список типовых экзаменационных задач. Эти материалы будут полезны и для самостоятельной подготовки.

1

Для нетерпеливого читателя

Для нетерпеливого читателя, желающего как можно быстрее научиться писать параллельные приложения, сразу же приведем пример превращения последовательной программы в параллельную. Для простоты рассмотрим задачу вычисления определенного интеграла от заданной функции и будем считать, что все входные параметры (концы отрезка интегрирования и количество точек, в которых вычисляется функция) заданы константами. Все использованные функции будут описаны в последующих главах.

1.1. Последовательная программа

Для вычисления приближения к определенному интегралу от функции f по отрезку $[a, b]$ используем составную формулу трапеций:

$$\int_a^b f(x)\,dx \approx h\big(f(a)/2 + \sum_{j=1}^{n-1} f(a+jh) + f(b)/2\big),$$

где $h = (b-a)/n$, а параметр n задает точность вычислений.

Вначале — файл `integral.c` с текстом последовательной программы, вычисляющей определенный интеграл этим способом:

```
#include "integral.h"
```

```
/* Интегрируемая функция */
double f (double x)
{
  return x;
}

/* Вычислить интеграл по отрезку [a, b] с числом точек
   разбиения n методом трапеций. */
double integrate (double a, double b, int n)
{
  double res;   /* результат */
  double h;     /* шаг интегрирования */
  int i;

  h = (b - a) / n;
  res = 0.5 * (f (a) + f (b)) * h;
  for (i = 1; i < n; i++)
    res += f (a + i * h) * h;
  return res;
}
```

Соответствующий заголовочный файл `integral.h`:

```
double integrate (double a, double b, int n);
```

Файл `sequential.c` с текстом последовательной программы:

```
#include <stdio.h>
#include "integral.h"

/* Все параметры для простоты задаются константами */
static double a = 0.;   /* левый конец интервала */
static double b = 1.;   /* правый конец интервала */
static int n = 100000000; /* число точек разбиения */

int main ()
{
```

```
  double total = 0.;    /* результат: интеграл */

  /* Вычислить интеграл */
  total = integrate (a, b, n);

  printf ("Integral from %lf to %lf = %.18lf\n",
          a, b, total);

  return 0;
}
```

Компиляция этих файлов:

```
cc sequential.c integral.c -o sequential
```

и запуск

```
./sequential
```

1.2. Ускорение работы за счет параллелизма

Для ускорения работы программы на вычислительной установке с p процессорами мы воспользуемся аддитивностью интеграла:

$$\int_a^b f(x)\,dx = \sum_{i=0}^{p-1} \int_{a_i}^{b_i} f(x)\,dx,$$

где $a_i = a + i * l$, $b_i = a_i + l$, $l = (b - a)/p$. Использовав для приближенного определения каждого из слагаемых $\int_{a_i}^{b_i} f(x)\,dx$ этой суммы составную формулу трапеций, взяв n/p в качестве n, и поручив эти вычисления своему процессору, мы получим p-кратное ускорение работы программы. Ниже мы рассмотрим три способа запуска p заданий для исполнения на отдельных процессорах:

1. создание новых процессов (раздел 1.3, с. 13),
2. создание новых задач (threads) (раздел 1.4, с. 18),
3. Message Passing Interface (MPI) (раздел 1.5, с. 21).

Первые два подхода удобно использовать в системах с общей памятью (см. раздел 2.5.1, с. 39). Последний подход применим и в системах с распределенной памятью.

1.3. Параллельная программа, использующая процессы

Рассмотрим пример параллельной программы вычисления определенного интеграла, использующей процессы. Описание ее работы приведено в разделе 8.4.6, с. 133, где рассматривается функция pipe; о функции fork см. раздел 7.1, с. 81. Файл process.c с текстом программы:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include "integral.h"

/* Все параметры для простоты задаются константами */
static double a = 0.;    /* левый конец интервала */
static double b = 1.;    /* правый конец интервала */
static int n = 100000000; /* число точек разбиения */

/* Канал вывода из главного процесса в порожденные.
   from_root[0] - для чтения (в порожденных процессах),
   from_root[1] - для записи (в главном процессе). */
static int from_root[2];

/* Канал вывода из порожденных процессов в главный.
   to_root[0] - для чтения (в порожденных процессах),
   to_root[1] - для записи (в главном процессе). */
static int to_root[2];

/* Функция, работающая в процессе с номером my_rank,
   при общем числе процессов p. */
void process_function (int my_rank, int p)
```

```
{
  char byte;
  /* длина отрезка интегрирования для текущего процесса*/
  double len = (b - a) / p;
  /* число точек разбиения для текущего процесса */
  int local_n = n / p;
  /* левый конец интервала для текущего процесса */
  double local_a = a + my_rank * len;
  /* правый конец интервала для текущего процесса */
  double local_b = local_a + len;
  /* значение интеграла в текущем процессе */
  double integral;

  /* Вычислить интеграл в каждом из процессов */
  integral = integrate (local_a, local_b, local_n);

  /* Ждать сообщения от главного процесса */
  if (read (from_root[0], &byte, 1) != 1)
    {
      /* Ошибка чтения */
      fprintf (stderr,
               "Error reading in process %d, pid = %d\n",
               my_rank, getpid ());
      return;
    }

  /* Передать результат главному процессу */
  if (write (to_root[1], &integral, sizeof (double))
      != sizeof (double))
    {
      /* Ошибка записи */
      fprintf (stderr,
               "Error writing in process %d, pid = %d\n",
               my_rank, getpid ());
      return;
```

```
    }
}

int main (int argc, char * argv[])
{
  /* Идентификатор запускаемого процесса */
  pid_t pid;
  /* Общее количество процессов */
  int p;
  int i;
  char byte;
  double integral = 0.;
  double total = 0.;    /* результат: интеграл */

  if (argc != 2)
    {
      printf ("Usage: %s <instances>\n", argv[0]);
      return 1;
    }

  /* Получаем количество процессов */
  p = (int) strtol (argv[1], 0, 10);

  /* Создаем каналы */
  if (pipe (from_root) == -1 || pipe (to_root) == -1)
    {
      fprintf (stderr, "Cannot pipe!\n");
      return 2;
    }

  /* Запускаем процессы */
  for (i = 0; i < p ; i++)
    {
      /* Клонировать себя */
      pid = fork ();
```

```
    if (pid == -1)
      {
        fprintf (stderr, "Cannot fork!\n");
        return 3 + i;
      }
    else if (pid == 0)
      {
        /* Процесс - потомок */
        /* Закрываем ненужные направления обмена */
        close (from_root[1]);
        close (to_root[0]);

        /* Проводим вычисления */
        process_function (i, p);

        /* Закрываем каналы */
        close (from_root[0]);
        close (to_root[1]);
        /* Завершаем потомка */
        return 0;
      }
    /* Цикл продолжает процесс - родитель */
  }

/* Закрываем ненужные направления обмена */
close (from_root[0]);
close (to_root[1]);

/* Получаем результаты */
for (i = 0; i < p ; i++)
  {
    /* Сигнализируем процессу */
    byte = (char) i;
    if (write (from_root[1], &byte, 1) != 1)
      {
```

```
            /* Ошибка записи */
            fprintf (stderr,
                     "Error writing in root process\n");
            return 100;
          }
        /* Считываем результат */
        if (read (to_root[0], &integral, sizeof (double))
            != sizeof (double))
          {
            /* Ошибка чтения */
            fprintf (stderr,
                     "Error reading in root process\n");
            return 101;
          }
        total += integral;
      }

    /* Закрываем каналы */
    close (from_root[1]);
    close (to_root[0]);

    printf ("Integral from %lf to %lf = %.18lf\n",
            a, b, total);

    return 0;
}
```

Компиляция этих файлов:

```
cc process.c integral.c -o process
```

и запуск

```
./process 2
```

где аргумент программы (в данном случае 2) — количество параллельно работающих процессов (обычно равен количеству имеющихся процессоров).

1.4. Параллельная программа, использующая задачи

Рассмотрим пример параллельной программы вычисления определенного интеграла, использующей задачи (threads). Описание ее работы см. в разделе 10.2, с. 168; описание использованных функций:

- `pthread_create` — см. раздел 9.1, с. 156;
- `pthread_join` — см. раздел 9.2, с. 157;
- `pthread_mutex_lock` — см. раздел 10.1.2, с. 162;
- `pthread_mutex_unlock` — см. раздел 10.1.4, с. 162.

Файл `thread.c` с текстом программы:

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

/* Все параметры для простоты задаются константами */
static double a = 0.;    /* левый конец интервала */
static double b = 1.;    /* правый конец интервала */
static int n = 100000000; /* число точек разбиения */

/* Результат: интеграл */
static double total = 0.;

/* Объект типа mutex для синхронизации доступа к total */
static pthread_mutex_t total_mutex
  = PTHREAD_MUTEX_INITIALIZER;

/* Общее количество процессов */
static int p;

/* Функция, работающая в задаче с номером my_rank */
void * process_function (void *pa)
{
```

```
  /* номер текущей задачи */
  int my_rank = (int) pa;
  /* длина отрезка интегрирования для текущего процесса*/
  double len = (b - a) / p;
  /* число точек разбиения для текущего процесса */
  int local_n = n / p;
  /* левый конец интервала для текущего процесса */
  double local_a = a + my_rank * len;
  /* правый конец интервала для текущего процесса */
  double local_b = local_a + len;
  /* значение интеграла в текущем процессе */
  double integral;

  /* Вычислить интеграл в каждой из задач */
  integral = integrate (local_a, local_b, local_n);

  /* "захватить" mutex для работы с total */
  pthread_mutex_lock (&total_mutex);
  /* сложить все ответы */
  total += integral;
  /* "освободить" mutex */
  pthread_mutex_unlock (&total_mutex);
  return 0;
}

int main (int argc, char * argv[])
{
  /* массив идентификаторов созданных задач */
  pthread_t * threads;
  int i;

  if (argc != 2)
    {
      printf ("Usage: %s <instances>\n", argv[0]);
      return 1;
```

```
  }

/* Получаем количество процессов */
p = (int) strtol (argv[1], 0, 10);

if (!(threads = (pthread_t*)
                malloc (p * sizeof (pthread_t))))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 1;
  }

/* Запускаем задачи */
for (i = 0; i < p; i++)
  {
    if (pthread_create (threads + i, 0,
                        process_function, (void*)i))
      {
        fprintf (stderr, "cannot create thread #%d!\n",
                 i);
        return 2;
      }
  }

/* Ожидаем окончания задач */
for (i = 0; i < p; i++)
  {
    if (pthread_join (threads[i], 0))
      fprintf (stderr, "cannot wait thread #%d!\n", i);
  }

/* Освобождаем память */
free (threads);

printf ("Integral from %lf to %lf = %.18lf\n",
```

```
          a, b, total);

  return 0;
}
```

Компиляция этих файлов:

```
cc thread.c integral.c -o thread
```

и запуск

```
./thread 2
```

где аргумент программы (в данном случае 2) — количество параллельно работающих задач (обычно равен количеству имеющихся процессоров).

1.5. Параллельная программа, использующая MPI

Рассмотрим пример параллельной программы вычисления определенного интеграла, использующей MPI. Описание ее работы см. в разделе 11.9, с. 256; описание использованных функций:

- `MPI_Init` и `MPI_Finalize` — см. раздел 11.1, с. 232;
- `MPI_Comm_rank` и `MPI_Comm_size` — см. раздел 11.3, с. 237;
- `MPI_Reduce` — см. раздел 11.8, с. 250.

Файл `mpi.c` с текстом программы:

```
#include <stdio.h>
#include "mpi.h"
#include "integral.h"

/* Все параметры для простоты задаются константами */
static double a = 0.;    /* левый конец интервала */
static double b = 1.;    /* правый конец интервала */
static int n = 100000000; /* число точек разбиения */

/* Результат: интеграл */
static double total = 0.;
```

```
/* Функция, работающая в процессе с номером my_rank,
   при общем числе процессов p. */
void process_function (int my_rank, int p)
{
  /* длина отрезка интегрирования для текущего процесса*/
  double len = (b - a) / p;
  /* число точек разбиения для текущего процесса */
  int local_n = n / p;
  /* левый конец интервала для текущего процесса */
  double local_a = a + my_rank * len;
  /* правый конец интервала для текущего процесса */
  double local_b = local_a + len;
  /* значение интеграла в текущем процессе */
  double integral;

  /* Вычислить интеграл в каждом из процессов */
  integral = integrate (local_a, local_b, local_n);

  /* Сложить все ответы и передать процессу 0 */
  MPI_Reduce (&integral, &total, 1, MPI_DOUBLE, MPI_SUM,
              0, MPI_COMM_WORLD);
}

int
main (int argc, char **argv)
{
  int my_rank;      /* ранг текущего процесса */
  int p;            /* общее число процессов */

  /* Начать работу с MPI */
  MPI_Init (&argc, &argv);

  /* Получить номер текущего процесса в группе всех
     процессов */
```

```
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  /* Получить общее количество запущенных процессов */
  MPI_Comm_size (MPI_COMM_WORLD, &p);

  /* Вычислить интеграл в каждом из процессов */
  process_function (my_rank, p);

  /* Напечатать ответ в процессе 0 */
  if (my_rank == 0)
    printf ("Integral from %lf to %lf = %.18lf\n",
            a, b, total);

  /* Заканчиваем работу с MPI */
  MPI_Finalize ();
  return 0;
}
```

Компиляция этих файлов:

```
mpicc mpi.c integral.c -o mpi
```

и запуск

```
mpirun -np 2 mpi
```

где опция `-np` задает количество параллельно работающих процессов (в данном случае 2).

2

Пути повышения производительности процессоров

Рассмотрим основные пути, которыми разработчики процессоров пытаются повысить их производительность. Нетерпеливый читатель может сразу заглянуть в раздел 2.7, где приведен пример программы умножения двух матриц, написанный с использованием знаний о внутреннем параллелизме в работе процессора, и дающий на обычном однопроцессорном компьютере почти 14-кратное ускорение работы по сравнению со стандартной программой.

2.1. CISC- и RISC-процессоры

Основной временно́й характеристикой для процессора является время цикла, равное $1/\mathbf{F}$, где $\mathbf{F}$ — тактовая частота процессора. Время, затрачиваемое процессором на задачу, может быть вычислено по формуле $\mathbf{C} * \mathbf{T} * \mathbf{I}$, где $\mathbf{C}$ — число циклов на одну инструкцию, $\mathbf{T}$ — время на один цикл, $\mathbf{I}$ — число инструкций на задачу.

Разработчики «классических» систем (которые теперь называют CISC (Complete Instruction Set Computer)) стремились уменьшить фактор $\mathbf{I}$. В процессорах реализовывались все более сложные инструкции, для выполнения которых внутри него самого запускались специальные процедуры (так называемый микрокод), загружаемые из ПЗУ внутри процессора. Этому пу-

ти способствовало то, что улучшения в технике производства полупроводников делали возможным реализацию все более сложных интегрированных цепей. Однако на этом пути очень трудно уменьшить два других фактора: **C** — поскольку инструкции сложные и требуют *программного декодирования*, и **T** — в силу *аппаратной сложности*.

Концепция RISC (Reduced Instruction Set Computer) возникла из статистического анализа того, как программное обеспечение использует ресурсы процессора. Исследования системных ядер и объектных модулей, порожденных оптимизирующими компиляторами, показали подавляющее доминирование простейших инструкций даже в коде для CISC-машин. Сложные инструкции используются редко, поскольку микрокод обычно не содержит в точности те процедуры, которые нужны для поддержки различных языков высокого уровня и сред исполнения программ. Поэтому разработчики RISC-процессоров убрали реализованные в микрокоде процедуры и передали программному обеспечению низкоуровневое управление машиной. Это позволило заменить процессорный микрокод в ПЗУ на подпрограмму в более быстрой ОЗУ.

Разработчики RISC-процессоров улучшили производительность за счет уменьшения двух факторов: **C** (за счет использования только простых инструкций) и **T** (за счет упрощения процессора). Однако изменения, внесенные для уменьшения числа циклов на инструкцию и времени на цикл, имеют тенденцию к увеличению числа инструкций на задачу. Этот момент был в центре внимания критиков RISC-архитектуры. Однако использование *оптимизирующих компиляторов* и других технических приемов, практически ликвидирует эту проблему.

2.2. Основные черты RISC-архитектуры

У RISC-процессора все инструкции имеют одинаковый формат и состоят из битовых полей, определяющих код инструкции и идентифицирующих ее операнды. В силу этого *декодирование*

инструкций производится аппаратно, т. е. микрокод не требуется. При этом в силу одинакового строения всех инструкций процессор может декодировать несколько полей *одновременно* для ускорения этого процесса.

Инструкции, производящие операции в памяти, обычно либо увеличивают время цикла, либо число циклов на инструкцию. Такие инструкции требуют дополнительного времени для своего исполнения, так как требуется вычислить адреса операндов, считать их из памяти, вычислить результат операции и записать его обратно в память. Для уменьшения негативного влияния таких инструкций разработчики RISC-процессоров выбрали *архитектуру чтение/запись*, в которой все операции выполняются над операндами в регистрах процессора, а основная память доступна только посредством инструкций чтения/записи. Для эффективности этого подхода RISC-процессоры имеют *большое количество регистров*. Архитектура чтение/запись также позволяет *уменьшить количество режимов адресации памяти*, что позволяет упростить декодирование инструкций.

Для CISC-архитектур время исполнения инструкции обычно измеряется в числе циклов на инструкцию. Разработчики RISC-архитектур, однако, стремились получить скорость выполнения инструкции, равную *одной инструкции за цикл*.

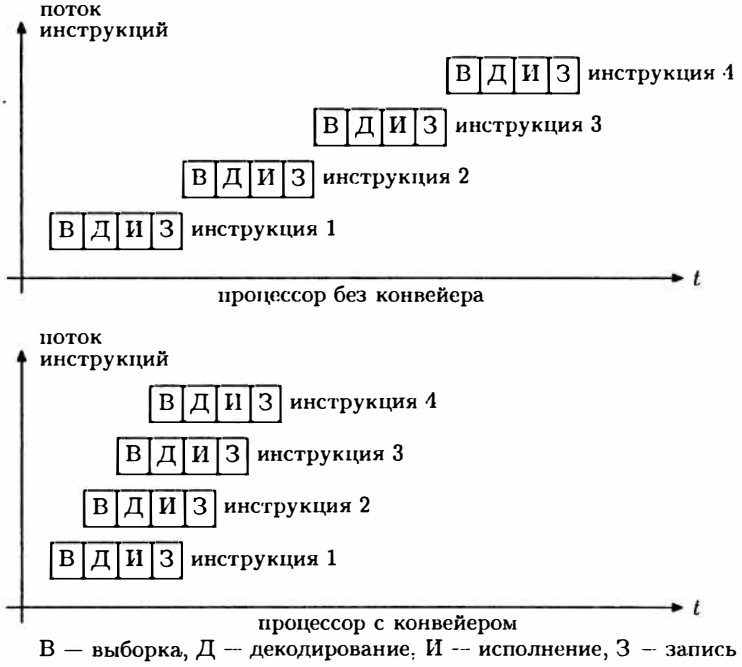
Для RISC-процессора во многих случаях только наличие *оптимизирующего компилятора* позволяет реализовать все его возможности. Отметим, что компилятор может наилучшим образом оптимизировать код именно для RISC-архитектур (в силу их простоты). Программирование на языке ассемблера исчезает для RISC-приложений, так как компиляторы языков высокого уровня могут производить очень сильную оптимизацию.

2.3. Конвейеризация

Конвейеризация является одним из основных способов повышения производительности процессора. Конвейерный процессор принимает новую инструкцию каждый цикл, даже если

предыдущие инструкции не завершены. В результате выполнение нескольких инструкций перекрывается и в процессоре находятся сразу несколько инструкций в разной степени готовности.

Исполнение инструкций может быть разделено на несколько стадий: **выборка**, **декодирование**, **исполнение**, **запись результатов**. **Конвейер инструкций** может уменьшить число циклов на инструкцию посредством одновременного исполнения нескольких инструкций, находящихся на разных стадиях (рис. 2.1). При правильной аппаратной реализации конвейер, имеющий n стадий, может одновременно исполнять n последовательных инструкций. Новая инструкция может приниматься к исполнению на каждом цикле, и эффективная скорость исполнения, таким образом, есть один цикл на инструкцию. Однако



В — выборка, Д — декодирование, И — исполнение, З — запись

Рис. 2.1. Ускорение работы процессора за счет конвейера

это предполагает, что конвейер всегда заполнен полезными инструкциями и нет задержек в прохождении инструкций через конвейер.

Управление конвейером инструкций требует надлежащего эффективного управления такими событиями, как **переходы**, **исключения** или **прерывания**, которые могут полностью нарушить поток инструкций. Например, результат условного перехода известен, только когда эта инструкция будет исполнена. Если конвейер был заполнен инструкциями, следующими за инструкцией условного перехода и переход состоялся, то все эти инструкции должны быть выброшены из конвейера.

Более того, внутри конвейера могут оказаться **взаимозависимые** инструкции. Например, если инструкция в стадии декодирования должна читать из ячейки памяти, значение которой является результатом работы инструкции, находящейся в стадии исполнения, то конвейер будет остановлен на один цикл, поскольку этот результат будет доступен только после стадии записи результатов. Поэтому компилятору необходимо **переупорядочить** инструкции в программе так, чтобы по возможности избежать зависимостей между инструкциями внутри конвейера.

Для уменьшения времени простоя конвейера применяют ряд мер.

- **Таблица регистров (register scoreboarding)** позволяет проследить за использованием регистров. Она имеет бит для каждого регистра процессора. Если этот бит установлен, то регистр находится в состоянии ожидания записи результата. После записи результата этот бит сбрасывается, разрешая использование данного регистра. Если этот бит сброшен для всех регистров, значения которых используются в текущей инструкции, то ее можно выполнять, не дожидаясь завершения исполнения предыдущих инструкций.
- **Переименование регистров (register renaming)** является аппаратной техникой уменьшения конфликтов из-за реги-

стровых ресурсов. Компиляторы преобразуют языки высокого уровня в ассемблерный код, назначая регистрам те или иные значения. В конвейерном процессоре операция может потребовать регистр до того, как предыдущая инструкция закончила использование этого регистра. Такое состояние *не является конфликтом данных*, поскольку этой операции не требуется значение регистра, а только сам регистр. Однако эта ситуация приводит к остановке конвейера до освобождения регистра. Идея разрешения этой проблемы состоит в следующем: берем свободный регистр, переименовываем его для соответствия параметрам инструкции, и даем инструкции его использовать в качестве требуемого ей регистра.

Задержки внутри конвейера могут быть также вызваны временем доступа к оперативной памяти DRAM, которое намного превышает время цикла. Эта проблема в значительной степени снимается при использовании кэш-памяти и буфера предвыборки инструкций (**очереди инструкций**).

Так как поток инструкций в CISC-процессоре **нерегулярный** и время исполнения одной инструкции ($\mathbf{C} * \mathbf{T}$) не постоянно, то конвейеризация в этом случае имеет серьезный недостаток, делающий ее малопригодной к использованию в CISC-процессорах: именно, она приводит к очень сильному усложнению процессора.

RISC-процессоры используют один и тот же формат всех инструкций для того, чтобы ускорить декодирование и упростить управление конвейером, поэтому все инструкции исполняются за **один** цикл.

Одним из способов дальнейшего повышения быстродействия является конвейеризация стадий конвейера. Такие процессоры называются **суперконвейерными**. При таком подходе каждая стадия конвейера, такая как кэш (см. ниже) или АЛУ (арифметическое и логическое устройство), может принимать новую инструкцию каждый цикл, даже если эта стадия не завершила исполнение текущей инструкции. Отметим, что добавление

новых уровней конвейеризации имеет смысл только в случае, если разработчик может **значительно увеличить частоту** процессора. Однако увеличение производительности за счет увеличения внутренней частоты процессора имеет ряд недостатков. Во-первых, это увеличивает потребление энергии процессором, что делает суперконвейерные процессоры малопригодными для встраиваемых (бортовых) систем, а также увеличивает габариты вычислительной установки за счет внушительной системы охлаждения. Во-вторых, это вводит новые трудности в сопряжении процессора с памятью нижнего уровня, такой как DRAM. Быстродействие этой памяти растет не так быстро, как скорость процессоров, поэтому чем быстрее процессор, тем больше разрыв в производительности между ним и основной памятью.

Другим способом увеличения производительности процессоров является выполнение более чем одной операции одновременно. Такие процессоры называются **суперскалярными**. Они имеют **два или более конвейеров инструкций**, работающих параллельно, что значительно увеличивает скорость обработки потока инструкций. Одним из достоинств суперскалярной архитектуры является возможность увеличения производительности без необходимости увеличения частоты процессора. Суперскалярному процессору требуется более **широкий** доступ к памяти, так, чтобы он мог брать сразу группу из нескольких инструкций для исполнения. **Диспетчер** анализирует эти группы и заполняет каждый из конвейеров так, чтобы снизить взаимозависимость данных и конфликты регистров. Выполнение инструкций может быть не по порядку поступления, так, чтобы команды перехода были проанализированы раньше, убирая задержки в случае осуществления перехода. Компилятор должен оптимизировать код для обеспечения заполнения всех конвейеров.

Требование одного и того же ресурса несколькими инструкциями блокирует их продвижение по конвейеру и приводит к вставке циклов ожидания требуемого ресурса. Суперскалярная архитектура с тремя исполняющими устройствами будет полно-

стью эффективной, только если поток инструкций обеспечивает одновременное использование этих трех устройств. Для обеспечения этого условия с минимальными расходами в процессоре выделяют исполняющие устройства, работающие независимо:

- **Целочисленное устройство** (IU, Integer Unit) — выполняет целочисленные операции (арифметические, логические и операции сравнения) в своем АЛУ.
- **Устройство для работы с плавающей точкой** (FPU, Floating Point Unit) — обычно отделено от целочисленного устройства, которое работает только с целыми числами и числами с фиксированной точкой. Большинство FPU совместимо со стандартом ANSI/IEEE для двоичной арифметики с плавающей точкой.
- **Устройство управления памятью** (MMU, Memory Management Unit) — вычисляет реальный физический адрес по виртуальному адресу.
- **Устройство предсказания переходов** (BU, Branch Unit) — занимается предсказанием условных переходов для того, чтобы избежать простоя конвейера в ожидании результата вычисления условия перехода.

Поскольку условные переходы могут свести на нет все преимущества конвейерной организации процессора, остановимся более подробно на приемах, используемых BU для уменьшения их негативного влияния.

- **«Отложенные слоты»** (delay slots). Инструкцию, передающую управление от одной части программы другой, трудно исполнить за один цикл. Обычно загрузка процессорного указателя на следующую инструкцию требует один цикл, предвыборка новой инструкции требует еще один. Для избежания простоя некоторые RISC-процессоры (например, SPARC) позволяют вставить дополнительную инструкцию в так называемый «отложенный слот». Эта инструкция расположена непосредственно после команды перехода, но она будет выполнена до того, как будет совершен переход.

Однако, для суперскалярных RISC-процессоров отложенные слоты работают не очень хорошо. Задержка при переходе может быть равна двум циклам, а суперскалярный процессор, который выполняет за цикл n инструкций, должен найти n инструкций для помещения в конвейеры.

- **«Спекулятивное» исполнение инструкций** (speculative execution). Некоторые RISC-процессоры (например, старшие модели семейств PowerPC и SPARC) используют так называемое «спекулятивное» исполнение инструкций: процессор загружает в конвейер и начинает исполнять инструкции, находящиеся за точкой ветвления, еще не зная, произойдет переход или нет. При этом часто выбирается наиболее вероятная ветвь программы (на основе того или иного подхода, см. ниже). Если после исполнения команды перехода оказалось, что процессор начал исполнять не ту ветвь, то все загруженные в конвейер инструкции из этой ветви и результаты их обработки сбрасываются, и загружается правильная ветвь.
- **Биты предсказания перехода в инструкции**. Некоторые RISC-процессоры (например, PowerPC) используют биты предсказания перехода, устанавливаevst компилятором в инструкции перехода, которые предсказывают, будет или нет совершен переход.
- **Эвристическое предсказание переходов**. Некоторые процессоры уменьшают задержки, вносимые переходами, за счет использования **встроенного предсказателя переходов**. Он предсказывает, что переходы вперед (проверки) произведены не будут, а переходы назад (циклы) — будут.

Для эффективной работы устройства предсказания перехода важно, чтобы код условия для условного перехода был вычислен как можно раньше (за несколько инструкций до самой команды перехода). Этого добиваются несколькими способами.

- **Независимость арифметических операций и кода условия**. В CISC-архитектурах все арифметические операции выставляют код условия по своему результату. Это

сделано для уменьшения фактора **I** — числа инструкций на задачу, поскольку есть вероятность того, что следующая инструкция будет вычислять код условия по результату предыдущей инструкции и, следовательно, может быть удалена. Однако это приводит к тому, что между командой вычисления кода условия и командой перехода очень трудно вставить полезные инструкции, так как они изменят код условия. В RISC-архитектурах арифметические операции *не изменяют код условия* (если противное явно не указано в инструкции, см. ниже). Поэтому возможно между инструкцией, вычисляющей код условия, и командой перехода вставить другие инструкции (переупорядочив их). Это позволит заранее узнать, произойдет или нет переход, и загрузить конвейер инструкциями.

Поскольку возможна ситуация, когда следующая инструкция будет вычислять код условия по результату предыдущей инструкции, то в RISC-архитектурах часть (SPARC) или все (PowerPC) арифметические операции также имеют вторую форму, в которой будет выставляться код условия по их результату. Таким образом, часть или все арифметические операции присутствуют в двух вариантах: один не изменяет код условия (подавляющее большинство случаев использования), а другой вычисляет код условия по результату операции.

- **Использование нескольких равноправных регистров с кодом условия**. Некоторые RISC-процессоры (например, PowerPC) используют несколько равноправных регистров, в которых образуется результат вычисления условия. Над этими регистрами определены логические операции, что иногда позволяет оптимизирующему компилятору заменить команды перехода при вычислении сложных логических выражений на команды логических операций с этими регистрами.
- **Использование кода условия в каждой инструкции**. Некоторые RISC-процессоры (например, ARM) используют код условия в каждой инструкции. В формате каждой

инструкции предусмотрено поле, где компилятором записывается код условия, при котором она будет выполнена. Если в момент исполнения инструкции код условия не такой, как в инструкции, то она игнорируется. Это позволяет вообще обойтись без команд перехода при вычислении результатов условных операций.

2.4. Кэш-память

Время, необходимое для выборки инструкций, в основном зависит от подсистемы памяти и часто является ограничивающим фактором для RISC-процессоров в силу высокой скорости исполнения инструкций. Например, если процессор может брать инструкции только из DRAM с временем доступа 60 нс, то скорость их обработки (при расчете одна инструкция за цикл) будет соответствовать тактовой частоте 16.7 МГц. Эта проблема в значительной степени снимается за счет использования **кэш-памяти**.

Кэш-память (cache) — это быстрое статическое ОЗУ (SRAM), вставленное между исполнительными устройствами и системным ОЗУ (RAM). Она сохраняет последние использованные инструкции и данные, так, что циклы и операции с массивами будут выполняться быстрее. Когда исполняющему устройству нужны данные и они не находятся в кэш-памяти, то это **кэш-промах**: процессор должен обратиться к внешней памяти для выборки данных. Если требуемые данные находятся в кэше, то это **кэш-попадание**: доступ к внешней памяти не требуется.

Таким образом, кэши разгружают внешние шины, уменьшая потребность в них для процессора. Это позволяет нескольким процессорам разделять внешние шины без уменьшения производительности каждого из них.

Кэш содержит строки из нескольких последовательных байтов (обычно 32 байта), которые загружаются процессором, используя так называемый *импульсный* (или блочный) *доступ*

(burst access). Даже если CPU нужен один байт, все равно будет загружена целая строка, так как вероятно, что тем самым будут загружены следующие выполняемые инструкции или используемые данные. Блочные передачи обеспечивают высокие скорости передачи для инструкций или данных в последовательных адресах памяти. При таких передачах только адрес первой инструкции или данного будет послан в подсистему внешней памяти. Все последующие запросы инструкций или данных в последовательных адресах памяти не требуют дополнительной передачи адреса. Например, загрузка 16 байтов требует 5 циклов, если MC68040 делает блочную передачу для загрузки строки кэша, и 8 циклов, если память не поддерживает блочный режим передачи.

Кэш, в котором вместе хранятся данные и инструкции, называется **единым кэшем**. Одним из способов повышения производительности является введение в процессоре *трех* шин: *адреса*, *инструкций* и *данных*. В **Гарвардской архитектуре кэша** разделяют кэши для инструкций и данных для удвоения эффективности кэш-памяти. В типичной Гарвардской архитектуре присутствуют три вида кэш-памяти: специальные кэши (например, **TLB**), внутренние кэши инструкций и данных (**первого уровня** или L1-кэш) и внешний единый кэш (**второго уровня** или L2-кэш). В процессорах, имеющих интегрированные кэши первого и второго уровней (т. е. внутри корпуса процессора), часто дополнительно устанавливают единый кэш **третьего уровня** (L3) вне процессора. Обычно L1-кэш работает на частоте процессора и имеет размер 8...32Кб, L2-кэш работает на частоте процессора или ее половине и имеет размер 128Кб...4Мб, L3-кэш работает на частоте внешней шины и имеет размер 512Кб...8Мб.

Кэши данных в зависимости от их поведения при записи данных в кэш разделяют на два вида.

1. Кэш с прямой записью (write-through cache). Этот вид кэш-памяти при записи в нее сразу инициирует цикл записи во

внешнюю память. Основным достоинством такого кэша является простота и то, что данные в кэше и в памяти всегда идентичны, что упрощает построение многопроцессорных систем.

2. Кэш с обратной записью (write-back cache). Этот вид кэш-памяти при записи в нее не записывает данные сразу во внешнюю память. Запись в память осуществляется при выходе строки из кэша или по запросу системы синхронизации в многопроцессорных системах. Такая организация кэш-памяти может значительно ускорить выполнение циклов, в которых обновляется одна и та же ячейка памяти (будет записано только последнее, а не все промежуточные значения, как в кэше с прямой записью). Другим достоинством является уменьшение потребности процессора во внешней шине, что позволяет разделять ее нескольким процессорам. Недостатком такой организации является усложнение схемы синхронизации кэшей в многопроцессорных системах.

В силу его значительно большей эффективности, большинство современных процессоров используют кэш с обратной записью.

Организация кэш-памяти. Кэш основан на **сравнении адреса**. Для каждой строки кэша хранится адрес ее первого элемента, называемый адресом строки. Для уменьшения объема дополнительно хранимой информации (адресов строк) и ускорения поиска адреса используют несколько технических приемов.

- Пусть длина строки есть 2^b байт. Адреса строк выровнены на границу своего размера, т. е. последние b бит адреса — нулевые и потому не хранятся (т. е. размер адреса уменьшен до $32 - b$ бит).
- Фиксируется некоторое i. Строки хранятся как один или несколько (N) массивов, отсортированными по порядку i младших битов адреса (т. е. младших среди оставшихся $32 - b$). Таким образом, k-й элемент массива имеет адрес, биты которого в позициях от $32 - i - b + 1$ до $32 - b$ образуют чис-

ло, равное k. Это позволяет не хранить эти биты (т. е. размер адреса уменьшен до $32 - b - i$ бит).

- Комбинация чисел b и i подбирается так, чтобы $b+i$ младших битов логического адреса совпадали бы с соответствующими битами физического адреса при страничном преобразовании (т. е. были бы смещением в странице). Это позволяет параллельно производить трансляцию адреса и поиск в кэше (т. е. параллельно работать MMU и кэшу). При типичном размере страницы 4 Кб это означает $b + i = 12$.
- Определение того, содержится ли данный адрес в кэше, производится следующим образом. Берутся биты в позициях от $32-i-b+1$ до $32-b$, образующие число k. Затем берутся элементы с номером k в каждом из N массивов и у полученных N строк сравниваются адреса с $32 - i - b$ битами адреса (которые уже транслированы MMU в физический адрес). Если обнаружено совпадение (т. е. имеет место кэш-попадание), то берется байт с номером b в строке.

Если рассматривается внешний кэш, то согласовывать его работу с MMU не требуется, поскольку внешний кэш работает уже с физическим адресом.

Пример: для PowerPC 603 выбрано $b = 5$ (т. е. длина строки 32 байта), $i = 7$ (т. е. длина массива 127), $N = 2$ (т. е. используются два массива).

Для каждой строки кэша с обратной записью помимо адреса хранится также признак того, что эта строка содержит корректные данные, т. е. данные в кэш-памяти и в основной памяти совпадают. Этот признак используется для записи строки в память при ее выходе из кэш-памяти, а также в многопроцессорных системах.

Алгоритмы замены данных в кэш-памяти. Если все строки кэш-памяти содержат корректные данные, то для обеспечения кэширования новых областей памяти необходимо выбрать строку, которая будет перезаписана. Эта строка выходит из кэ-

ша и, если требуется, ее содержимое будет записано обратно в память. Существуют три алгоритма замены данных в кэше:

- **вероятностный** алгоритм: в качестве номера перезаписываемой строки используется случайное число;
- **FIFO**-алгоритм: первая записанная строка будет первой перезаписана;
- **LRU**-алгоритм (Last Recently Used): наименее используемая строка будет заменена новой.

Для повышения производительности процессора вводится ряд **специальных кэшей**.

- **TLB** (Translation Look-aside Buffers) — это кэш-память, используемая MMU для хранения результатов последних трансляций логического адреса в физический. Содержит пары: логический адрес и соответствующий физический адрес.
- **BTC** (Branch Target Cache) — это кэш-память, используемая BU для хранения адреса предыдущего перехода и первой инструкции, выполненной после перехода. Имеет целью без задержки заполнить конвейер инструкцией, если переход уже ранее состоялся. BTC может значительно повысить производительность процессора, учитывая время, которое он бы простаивал в ожидании заполнения конвейера после перехода.

Согласование кэшей в мультипроцессорных системах. Если несколько процессоров подсоединены к одной и той же шине адреса и данных и разделяют одну и ту же внешнюю память, то должен быть реализован определенный следящий механизм (snooping) для того, чтобы все внутрипроцессорные кэши всегда содержали **одни и те же** данные.

Рассмотрим, например, систему, содержащую два процессора, каждый из которых может брать управление общей шиной. Если процессор 1, управляющий в данный момент шиной, записывает в ячейку памяти, которая кэширована процессором 2, то данные в кэше последнего становятся устаревшими. Следящий механизм позволяет второму процессору отслеживать состоя-

ние шины адреса, даже если он не является в данный момент главным (т. е. управляющим внешней шиной). Если на шине появился адрес кэшированных данных, то эти данные помечаются в кэше как некорректные. Когда второй процессор станет главным, он должен будет выбрать в случае необходимости обновленные данные из разделяемой памяти.

Если процессор 1, управляющий в данный момент шиной, читает из ячейки памяти, которая кэширована процессором 2, то возможно, что реальные данные находятся в кэше процессора 2 (еще не записаны в память, т. е. реализован кэш с обратной записью). Если это так, то следящий механизм инициирует цикл записи строки кэша процессора 2, содержащей затребованные процессором 1 данные, в разделяемую память. После этого эти данные становятся доступными процессору 1 и цикл чтения процессора 1 продолжается.

2.5. Многопроцессорные архитектуры

Одним из самых радикальных способов повышения производительности вычислительной системы является установка нескольких процессоров.

2.5.1. Основные архитектуры

Выделяют несколько типов построения многопроцессорных систем в зависимости от степени связи между отдельными процессорами в системе.

1. **Сильно связанные процессоры** (или симметричные мультипроцессорные системы, symmetrical multiprocessor system, SMP). Все процессоры разделяют общую шину и общую память, могут выполнять одну и ту же задачу, причем задача может переходить от одного процессора к другому. Если один процессор отказывает, он может быть заменен другим. SMP подразумевает наличие аппаратного протокола

синхронизации кэшей всех процессоров (см. выше). Типичный пример: плата с двумя процессорами Pentium.

2. **Слабо связанные процессоры**. Часть системной памяти может быть разделяема, но переход задачи от одного процессора к другому невозможен. Механизмы синхронизации специфичны для каждой системы (почтовые ящики, DPRAM, прерывания). Типичный пример: стойка промышленного стандарта VME с несколькими процессорными платами и разделяемой памятью на одной из плат.
3. **Распределенные процессоры**. Несколько процессоров не разделяют ни одного общего ресурса, за исключением линии связи. Типичный пример: соединенные посредством Ethernet рабочие станции.

Очень часто многопроцессорные системы делят на два класса в зависимости от способа доступа процессов к оперативной памяти (см. раздел 5.1 о разделяемой памяти):

1. **Системы с общей памятью** — процессы могут разделять блок оперативной памяти.
2. **Системы с распределенной памятью** — у каждого из процессов своя оперативная память, которая не может быть сделана разделяемой между ними.

Система на сильно связанных процессорах, конечно, является системой с общей памятью. Отметим, что система на распределенных процессорах с помощью программного обеспечения тоже может быть превращена в систему с общей памятью.

2.5.2. Комбинированные архитектуры

Архитектура SMP является самой дорогой с точки зрения аппаратной реализации и самой дешевой с точки зрения разработки программного обеспечения. И наоборот, распределенные процессоры почти не требуют аппаратных затрат, но являются самым дорогим решением с точки зрения разработки ПО. Для достиже-

ния оптимального компромисса для круга решаемых задач используют различные комбинации описанных выше технологий.

- **Гибридные схемы SMP** используются для ускорения работы программ, требующих интенсивной работы с оперативной памятью. Описанная выше схема SMP ускоряет выполнение задач, которым не нужен постоянный обмен с RAM, поскольку все процессоры разделяют одну шину и в каждый момент времени только один процессор может работать с памятью. Поэтому иногда (это очень дорогое решение) поступают следующим образом: система строится на базе модулей, каждый из которых является самостоятельной процессорной платой с двумя (или четырьмя) процессорами, включенными по схеме SMP. В каждом модуле расположена своя оперативная память, к которой через общую шину имеют доступ процессоры этого модуля. Логически память каждого модуля включена в общую память системы, т. е. память всей системы образована как объединение памяти каждого из модулей. Физически доступ к памяти других модулей осуществляется через быстродействующую коммуникационную шину, которая может быть как общей для всех модулей, так и быть соединением типа «точка-точка» по принципу «все со всеми» (обычно используется комбинация этих способов организации шины). Если программа может быть организована так, что в основном группе процессоров нужна не вся память, а только ее часть (размером с память модуля), и обмены между этими частями идут не часто (это весьма реалистичные предположения, особенно, если память каждого из модулей достаточно велика), то такая программа может эффективно работать на описанной архитектуре.
- **Гибридные схемы слабо связанных (распределенных) процессоров** используются для повышения надежности работы систем на основе архитектуры со слабо связанными (распределенными) процессорами. Используются для ускорения работы систем, в которых требуется обрабатывать

одновременно много процессов и обеспечить отказоустойчивость. Вместо слабо связанных (распределенных) процессоров используют слабо связанные (распределенные) процессорные модули, каждый из которых является самостоятельной процессорной платой с двумя (или четырьмя) процессорами, включенными по схеме SMP. В случае отказа одного из процессоров в модуле, имеется потенциальная возможность передать все процессы оставшимся процессорам.

- **Кластерная организация процессоров** является частным случаем описанных выше архитектур и используется для ускорения работы систем, в которых требуется обрабатывать одновременно много процессов. Физически кластер строится либо на базе распределенных процессоров, либо на базе слабо связанных процессоров (без разделяемой памяти, фактически несколько независимых процессорных систем, объединенных общим корпусом, источниками питания и системами ввода-вывода). Первый способ был более распространен в момент появления понятия «кластер», второй становится популярным в настоящее время в связи с миниатюризацией процессорных систем. Логически кластер представляет собой систему, в которой каждый из процессоров независим и может независимо от других принимать к исполнению задания. Операционная система направляет вновь пришедший процесс на исполнение тому процессору, который в данный момент менее загружен (т. е. для пользователя вся система выглядит как единая многопроцессорная установка).

  Если коммуникационный канал, связывающий отдельные процессорные модули, имеет высокую пропускную способность, то операционная система может эмулировать на такой системе поведение описанной выше системы гибридной SMP, а именно, она может позволить исполнять одно и то же приложение на нескольких процессорах. Это относится не только к кластерам, но и ко всем архитектурам с распределенными процессорами.

2.5.3. Обанкротившиеся архитектуры

В этом разделе мы рассмотрим многопроцессорные архитектуры, которые в чистом виде уже не применяются в новых разработках. Однако они оказали значительное влияние на развитие вычислительной техники, и некоторые их черты, уже в новом технологическом воплощении, присутствуют и в современных архитектурах.

Основными причинами, приведшими к выходу из употребления этих архитектур (и разорению фирм, выпускавших на их основе ЭВМ), являются:

- необходимость разрабатывать программное обеспечение специально под конкретную архитектуру;
- конкуренция с параллельными архитектурами;
- снижение расходов на оборонные разработки (большинство проектов по высокопроизводительным вычислениям финансировалось из военных источников).

Основные архитектуры:

- **Векторные процессоры** — это даже не многопроцессорные архитектуры, а одиночные процессоры, способные выполнять операции с векторами как примитивные инструкции. Это может значительно ускорить выполнение программ вычислительной математики, поскольку в таких задачах основное время работы приходится на обработку векторов и матриц.
- **Матричные процессоры** — это процессоры, способные выполнять операции над матрицами как примитивные инструкции. Обычно представляют собой группу процессоров, разделяющих общую память и выполняющих один поток инструкций.
- **Транспьютеры** — это многопроцессорные архитектуры, состоящие из независимых процессоров (обычно кратных по количеству 4), каждый из которых имеет свою подсистему памяти. Процессоры связаны между собой высокоскорост-

ными соединениями, организованными по принципу «точка-точка». Каждый процессор связан с четырьмя другими.

2.6. Поддержка многозадачности и многопроцессорности

В современных процессорах поддержка многозадачности и многопроцессорности не ограничивается аппаратной частью (вроде рассмотренного выше механизма синхронизации кэшей). Для организации доступа к критическим разделяемым ресурсам необходимо в наборе инструкций процессора предусмотреть специальные инструкции, обеспечивающие доступ к объектам синхронизации. Действительно, сами объекты синхронизации являются *разделяемыми*, а для обеспечения правильного доступа к разделяемым объектам необходимо вводить объекты синхронизации, которые в свою очередь тоже являются разделяемыми и т. д. Для выхода из этого замкнутого круга определяют некоторые «примитивные» объекты синхронизации, к которым возможен одновременный доступ нескольких задач или процессоров за счет использования *специальных инструкций доступа*.

Рассмотрим более подробно случай булевского семафора. Задача или процессор, собирающийся взять управление разделяемым ресурсом, начинают с чтения значения семафора. Если он обнулен, то задача или процессор должны ждать, пока ресурс станет доступным. Если семафор установлен в 1, то задача или процессор немедленно его обнуляют, чтобы показать, что они контролируют ресурс. В процессе изменения семафора можно выделить три фазы: **чтение**, **изменение**, **запись**. Если на стадии чтения возникнет переключение задач или другой процессор станет главным на шине, то может возникнуть ошибка, так как две задачи или два процессора контролируют один и тот же ресурс. Аналогично, если переключение контекста произойдет между циклом чтения и записи, то два процесса могут взять семафор, что тоже приведет к системной ошибке. Для решения этой проблемы большинство процессоров имеют

инструкцию, выполняющую неделимый цикл чтение – изменение – запись. Поскольку это одна инструкция, то переключение задач во время операции с семафором невозможно. Так как она производит неделимый цикл, то процессор, ее выполняющий, остается владельцем шины до окончания операции с семафором.

2.7. Использование параллелизма процессора для повышения эффективности программ

Рассмотрим задачу умножения двух $n \times n$ квадратных матриц: $C = AB$. Элементы матриц в языке C хранятся в оперативной памяти по строкам: вначале элементы первой строки, затем второй и т. д. Например, последовательность элементов матрицы A:

$$a_{1,1}, a_{1,2}, \ldots, a_{1,n}, a_{2,1}, a_{2,2}, \ldots, a_{2,n}, a_{3,1}, \ldots, a_{n,n-1}, a_{n,n}.$$

Таким образом, последовательные элементы строки матрицы лежат друг за другом и доступ к ним будет максимально быстрым, так как стратегия предвыборки элементов в кэш (т. е. чтение каждый раз целого блока оперативной памяти, равного по размеру строке кэша, см. раздел 2.4, с. 34) себя полностью оправдывает. С другой стороны, последовательные элементы столбца матрицы лежат на расстоянии `n * sizeof (double)` байт друг от друга и доступ к ним будет максимально медленным (поскольку строка кэш-памяти существенно меньше расстояния между элементами).

Вычислим произведение матриц несколькими способами:

1. Цикл проведем по строкам матрицы C (стандартный способ):

$$\begin{gathered}\text{для всех} \quad m = 1, 2, \ldots, n \\ \text{для всех} \quad i = 1, 2, \ldots, n \\ \text{вычислять} \quad c_{mi} = \sum_{j=1}^{n} a_{mj} b_{ji}\end{gathered}$$

   В цикле по j (вычисление суммы) элементы одной строки матрицы A (к которым доступ и так быстрый) будут исполь-

зованы многократно (в цикле по i), элементы же столбцов матрицы B (к которым доступ относительно медленный) повторно (в цикле по i) не используются. Тем самым при доступе к элементам B кэш-память практически ничего не дает.

2. Цикл проведем по столбцам матрицы C.

$$\begin{array}{c}\text{для всех} \quad m = 1, 2, \ldots, n \\ \text{для всех} \quad i = 1, 2, \ldots, n \\ \text{вычислять} \quad c_{im} = \sum_{j=1}^{n} a_{ij} b_{jm}\end{array}$$

В цикле по j (вычисление суммы) элементы матрицы A повторно (в цикле по i) не используются. Элементы же столбцов матрицы B используются многократно (в цикле по i). Кэш-память ускоряет доступ к элементам A за счет предвыборки подряд идущих элементов, и ускоряет доступ к элементам B, **если столбец B целиком поместился в кэш-память** (тогда в цикле по i обращение к элементам B в оперативной памяти будет только на первом шаге). Отметим, что если размер кэш-памяти недостаточен (например, n велико), то этот способ может оказаться хуже предыдущего, так как мы ухудшили способ доступа к A и не получили выигрыша при доступе к B.

3. Цикл проведем по $N \times N$, $N = 10$, блокам матрицы C, внутри блока идем по столбцам:

$$\begin{array}{c}\text{для всех} \quad b_m = 1, 1+N, 1+2N \ldots, \quad b_m < n \\ \text{для всех} \quad b_i = 1, 1+N, 1+2N \ldots, \quad b_i < n \\ \text{для всех} \quad m = b_m, b_m+1, \ldots, b_m+N-1, \quad m < n \\ \text{для всех} \quad i = b_i, b_i+1, \ldots, b_i+N-1, \quad i < n \\ \text{вычислять} \quad c_{im} = \sum_{j=1}^{n} a_{ij} b_{jm}\end{array}$$

Теперь мы максимально локализуем количество вовлеченных в самые внутренние циклы элементов, увеличивая тем самым эффективность работы кэш-памяти. Особенно большой выигрыш в скорости работы мы получим, если используемые в циклах по m, i, j N строк матрицы A и N столбцов

матрицы B поместились в кэш-память. Если же n так велико, что даже один столбец B не поместился целиком в кэш, то этот способ может оказаться хуже самого первого.

4. Зададимся параметром N (для простоты ниже предполагается, что n делится на N нацело, хотя программа разработана для произвольного N). Всякая матрица M может быть представлена как составленная из блоков:

$$M = \begin{pmatrix} M_{11} & M_{12} & \dots & M_{1k} \\ M_{21} & M_{22} & \dots & M_{2k} \\ \dots & \dots & \ddots & \dots \\ M_{k1} & M_{k2} & \dots & M_{kk}, \end{pmatrix}$$

где M_{ij} — $N \times N$ матрица, $k = n/N$. Тогда каждый блок C_{im} произведения $C = AB$ матриц A и B может быть вычислен через блоки матриц A и B:

$$C_{im} = \sum_{j=1}^{n} A_{ij} B_{jm}$$

В вычислении произведения $N \times N$ матриц A_{ij} и B_{jm} участвует только подмножество из N^2 элементов матриц A и B. При небольшом N (в нашем тесте $N = 40$) это подмножество полностью поместится в кэш-памяти и каждое слагаемое последней суммы будет вычислено максимально быстро.

5. В предыдущем варианте мы уже практически исчерпали все преимущества, которые дает нам кэш-память. Для получения дальнейшего прироста производительности вспомним, что все современные процессоры имеют конвейер инструкций (см. раздел 2.3, с. 26), причем весьма глубокий (может доходить до полутора десятков стадий). Для его заполнения длина линейного участка программы (т. е. не содержащего команд перехода) должна быть как минимум больше глубины конвейера (желательно — в несколько раз больше). Во всех же предыдущих вариантах в самом внутреннем цикле (по j) находится 1 оператор языка C, который, в зависимости от целевого процессора, транслируется компилятором в

7...12 инструкций (включая операторы обслуживания цикла). Для увеличения длины линейного участка программы «развернем» цикл в предыдущем варианте: за один виток цикла по j будем вычислять сразу 4 элемента матрицы C: $c_{i,m}, c_{i,m+1}, c_{i+1,m}, c_{i+1,m+1}$. Поскольку при их вычислении используются повторяющиеся элементы матриц A и B, то также появляются дополнительные резервы для ускорения работы за счет оптимизации компилятора и кэш-памяти.

Исходный текст этих пяти вариантов приведен ниже.

```
#include "mmult.h"

/* Умножить матрицу a на матрицу b, c = ab,
   цикл по строкам c */
void matrix_mult_matrix_1 (double *a, double *b,
                           double *c, int n)
{
  int i, j, m;
  double *pa, *pb, *pc, s;

  for (m = 0, pc = c; m < n; m++)
    {
      for (i = 0, pb = b; i < n; i++, pb++)
        {
          for (s = 0., j = 0, pa = a + m * n; j < n; j++)
            s += *(pa++) * pb[j * n];
          *(pc++) = s;
        }
    }
}

/* Умножить матрицу a на матрицу b, c = ab,
   цикл по столбцам c */
void matrix_mult_matrix_2 (double *a, double *b,
                           double *c, int n)
```

```
{
  int i, j, m;
  double *pa, *pb, *pc, s;

  for (m = 0, pc = c; m < n; m++, pc++)
    {
      for (i = 0, pa = a, pb = b + m; i < n; i++)
        {
          for (s = 0., j = 0; j < n; j++)
            s += *(pa++) * pb[j * n];
          pc[i * n] = s;
        }
    }
}

/* Размер блока */
#define N       10

/* Умножить матрицу a на матрицу b, c = ab,
   цикл по блокам с */
void matrix_mult_matrix_3 (double *a, double *b,
                           double *c, int n)
{
  int bm, bi, nbm, nbi;
  int i, j, m;
  double *pa, *pb, *pc, s;


  for (bm = 0; bm < n; bm += N)
    {
      nbm = (bm + N <= n ? bm + N : n);
      for (bi = 0; bi < n; bi += N)
        {
          nbi = (bi + N <= n ? bi + N : n);
```

```
          /* Вычислить результирующий блок матрицы c
             с верхним левым углом (bi, bm)
             и правым нижним (nbi-1, nbm-1) */
          for (m = bm, pc = c + bm; m < nbm; m++, pc++)
            {
              for (i = bi, pa = a + bi * n, pb = b + m;
                   i < nbi; i++)
                {
                  for (s = 0., j = 0; j < n; j++)
                    s += *(pa++) * pb[j * n];
                  pc[i * n] = s;
                }
            }
        }
    }
}

#undef  N

/* Размер блока */
#define N       40

/* Умножить матрицу a на матрицу b, c = ab,
   блочное умножение матриц */
void matrix_mult_matrix_4 (double *a, double *b,
                           double *c, int n)
{
  int bm, bi, nbm, nbi;
  int l, nl;
  int i, j, m;
  double *pa, *pb, *pc, s;

  for (bm = 0; bm < n; bm += N)
    {
      nbm = (bm + N <= n ? bm + N : n);
```

```
      for (bi = 0; bi < n; bi += N)
        {
          nbi = (bi + N <= n ? bi + N : n);

          /* Вычислить результирующий блок матрицы c
             с верхним левым углом (bi, bm)
             и правым нижним (nbi-1, nbm-1) */
          /* Вычисляем как произведения блоков матриц
             a и b */
          for (m = bm, pc = c + bm; m < nbm; m++, pc++)
            for (i = bi; i < nbi; i++)
              pc[i * n] = 0.;

          for (l = 0; l < n; l += N)
            {
              nl = (l + N <= n ? l + N : n);
              /* Вычисляем произведение
                 блока матрицы a [(bi,l) x (nbi-1,nl-1)]
                 на блок матрицы b [(l,bm)x(nl-1,nbm-1)]
                 и прибавляем к блоку
                 матрицы c [(bi,bm) x (nbi-1,nbm-1)] */
              for (m = bm, pc = c+bm; m < nbm; m++, pc++)
                for (i = bi, pb = b + m; i < nbi; i++)
                  {
                    pa = a + l + i * n;
                    for (s = 0., j = l; j < nl; j++)
                      s += *(pa++) * pb[j * n];
                    pc[i * n] += s;
                  }
            }
        }
    }
}

#undef  N
```

```
/* Размер блока, должен делиться на 2 */
#define N       40

/* Умножить матрицу a на матрицу b, c = ab,
   блочное умножение матриц, внутренний цикл развернут */
/* Работает только для четных n (для всех n придется
   усложнять разворачивание цикла) */
void matrix_mult_matrix_5 (double *a, double *b,
                           double *c, int n)
{
  int bm, bi, nbm, nbi;
  int l, nl;
  int i, j, m;
  double *pa, *pb, *pc;
  double s00, s01, s10, s11;

  for (bm = 0; bm < n; bm += N)
    {
      nbm = (bm + N <= n ? bm + N : n);
      for (bi = 0; bi < n; bi += N)
        {
          nbi = (bi + N <= n ? bi + N : n);

          /* Вычислить результирующий блок матрицы c
             с верхним левым углом (bi, bm)
             и правым нижним (nbi-1, nbm-1) */
          /* Вычисляем как произведения блоков матриц
             a и b */
          for (m = bm, pc = c + bm; m < nbm; m++, pc++)
            for (i = bi; i < nbi; i++)
              pc[i * n] = 0.;

          for (l = 0; l < n; l += N)
            {
```

```
                nl = (l + N <= n ? l + N : n);
                /* Вычисляем произведение
                   блока матрицы a [(bi,l) x (nbi-1,nl-1)]
                   на блок матрицы b [(l,bm)x(nl-1,nbm-1)]
                   и прибавляем к блоку
                   матрицы c [(bi,bm) x (nbi-1,nbm-1)] */
                for (m = bm, pc = c + bm; m < nbm;
                     m += 2, pc += 2)
                  for (i = bi, pb = b + m; i < nbi; i += 2)
                    {
                      pa = a + l + i * n;
                      s00 = s01 = s10 = s11 = 0.;
                      for (j = l; j < nl; j++, pa++)
                        {
                          /* элемент (i, m) */
                          s00 += pa[0] * pb[j * n];
                          /* элемент (i, m + 1) */
                          s01 += pa[0] * pb[j * n + 1];
                          /* элемент (i + 1, m) */
                          s10 += pa[n] * pb[j * n];
                          /* элемент (i + 1, m + 1) */
                          s11 += pa[n] * pb[j * n + 1];
                        }
                      pc[i * n]           += s00;
                      pc[i * n + 1]       += s01;
                      pc[(i + 1) * n]     += s10;
                      pc[(i + 1) * n + 1] += s11;
                    }
            }
        }
    }
}
```

Отношение времени работы первого варианта (стандартного) к времени работы каждого из 5 вариантов для матриц

Таблица 2.1. Соотношение скорости работы различных алгоритмов умножения матриц

Процессор	Pentium III	Pentium III	PowerPC 603e
Размерность	1000	2000	1000
Алгоритм 1	1.0	1.0	1.0
Алгоритм 2	1.3	1.0	1.0
Алгоритм 3	1.9	1.0	1.0
Алгоритм 4	2.3	7.6	5.6
Алгоритм 5	4.3	13.8	11.0

1000×1000 и 2000×2000, вычисленное на процессорах Intel Pentium III и IBM/Motorola PowerPC 603e, приведено в таблице 2.1.

Отметим, что просто переписав программу с учетом наличия кэш-памяти и конвейера инструкций, мы получили более чем 10-кратное ускорение на одном процессоре, без использования параллельной вычислительной установки!

По просьбам читателей ниже приводится текст описанных выше вариантов умножения матриц на языке FORTRAN-77.

```
C Умножить матрицу a на матрицу b, c = ab,
C цикл по строкам c
        subroutine matrix_mult_matrix_1 (a, b, c, n)
        real*8 a (n, n), b (n, n), c (n, n)
        integer n

        integer i, j, m
        real*8 s

        do m = 1, n
          do i = 1, n
            s = 0.d0
            do j = 1, n
              s = s + a (m, j) * b (j, i)
```

```
            end do
            c (m, i) = s
          end do
        end do
        end subroutine

C Умножить матрицу a на матрицу b, c = ab,
C цикл по столбцам c
        subroutine matrix_mult_matrix_2 (a, b, c, n)
        real*8 a (n, n), b (n, n), c (n, n)
        integer n

        integer i, j, m
        real*8 s

        do m = 1, n
          do i = 1, n
            s = 0.d0
            do j = 1, n
              s = s + a (i, j) * b (j, m)
            end do
            c (i, m) = s
          end do
        end do
        end subroutine

C Умножить матрицу a на матрицу b, c = ab,
C цикл по блокам c
        subroutine matrix_mult_matrix_3 (a, b, c, n)
        real*8 a (n, n), b (n, n), c (n, n)
        integer n

        integer bm, bi, nbm, nbi
        integer i, j, m
        real*8 s
```

```
      parameter (nn = 10)

      do bm = 1, n, nn
        if (bm + nn .le. n + 1) then
          nbm = bm + nn
        else
          nbm = n + 1
        endif
        do bi = 1, n, nn
          if (bi + nn .le. n + 1) then
            nbi = bi + nn
          else
            nbi = n + 1
          endif
C         Вычислить результирующий блок матрицы c
C         с верхним левым углом (bi, bm)
C         и правым нижним (nbi-1, nbm-1)
          do m = bm, nbm - 1
            do i = bi, nbi - 1
              s = 0.d0
              do j = 1, n
                s = s + a (i, j) * b (j, m)
              end do
              c (i, m) = s
            end do
          end do
        end do
      end do
      end subroutine

C Умножить матрицу a на матрицу b, c = ab,
C блочное умножение матриц
      subroutine matrix_mult_matrix_4 (a, b, c, n)
      real*8 a (n, n), b (n, n), c (n, n)
      integer n
```

```
      integer bm, bi, nbm, nbi
      integer l, nl
      integer i, j, m
      real*8 s
      parameter (nn = 40)

      do bm = 1, n, nn
        if (bm + nn .le. n + 1) then
          nbm = bm + nn
        else
          nbm = n + 1
        endif
        do bi = 1, n, nn
          if (bi + nn .le. n + 1) then
            nbi = bi + nn
          else
            nbi = n + 1
          endif
C         Вычислить результирующий блок матрицы c
C         с верхним левым углом (bi, bm)
C         и правым нижним (nbi-1, nbm-1)
C         Вычисляем как произведения блоков матриц
C         a и b
          do m = bm, nbm - 1
            do i = bi, nbi - 1
              c (i, m) = 0.d0
            end do
          end do

          do l = 1, n, nn
            if (l + nn .le. n + 1) then
              nl = l + nn
            else
              nl = n + 1
```

```
              endif
C             Вычисляем произведение
C             блока матрицы a [(bi,l) x (nbi-1,nl-1)]
C             на блок матрицы b [(l,bm)x(nl-1,nbm-1)]
C             и прибавляем к блоку
C             матрицы c [(bi,bm) x (nbi-1,nbm-1)]
              do m = bm, nbm - 1
                do i = bi, nbi - 1
                  s = 0.d0
                  do j = l, nl - 1
                    s = s + a (i, j) * b (j, m)
                  end do
                  c (i, m) = c (i, m) + s
                end do
              end do
            end do
          end do
        end do
        end subroutine

C Умножить матрицу a на матрицу b, c = ab,
C блочное умножение матриц, внутренний цикл развернут
C Работает только для четных n (для всех n придется
C усложнять разворачивание цикла)
        subroutine matrix_mult_matrix_5 (a, b, c, n)
        real*8 a (n, n), b (n, n), c (n, n)
        integer n

        integer bm, bi, nbm, nbi
        integer l, nl
        integer i, j, m
        real*8 s00, s01, s10, s11
        parameter (nn = 40)

        do bm = 1, n, nn
```

```
            if (bm + nn .le. n + 1) then
              nbm = bm + nn
            else
              nbm = n + 1
            endif
            do bi = 1, n, nn
              if (bi + nn .le. n + 1) then
                nbi = bi + nn
              else
                nbi = n + 1
              endif
C             Вычислить результирующий блок матрицы c
C             с верхним левым углом (bi, bm)
C             и правым нижним (nbi-1, nbm-1)
C             Вычисляем как произведения блоков матриц
C             a и b
              do m = bm, nbm - 1
                do i = bi, nbi - 1
                  c (i, m) = 0.d0
                end do
              end do

              do l = 1, n, nn
                if (l + nn .le. n + 1) then
                  nl = l + nn
                else
                  nl = n + 1
                endif
C               Вычисляем произведение
C               блока матрицы a [(bi,l) x (nbi-1,nl-1)]
C               на блок матрицы b [(l,bm)x(nl-1,nbm-1)]
C               и прибавляем к блоку
C               матрицы c [(bi,bm) x (nbi-1,nbm-1)]
                do m = bm, nbm - 1, 2
                  do i = bi, nbi - 1, 2
```

```
                    s00 = 0.d0
                    s01 = 0.d0
                    s10 = 0.d0
                    s11 = 0.d0
                    do j = l, nl - 1
C                     элемент (i, m)
                      s00 = s00 + a (i, j) * b (j, m)
C                     элемент (i, m + 1)
                      s01 = s01 + a (i, j) * b (j, m + 1)
C                     элемент (i + 1, m)
                      s10 = s10 + a (i + 1, j) * b (j, m)
C                     элемент (i + 1, m + 1)
                      s11 = s11 + a (i + 1, j) * b (j, m+1)
                    end do
                    c (i, m) = c (i, m) + s00
                    c (i, m + 1) = c (i, m + 1) + s01
                    c (i + 1, m) = c (i + 1, m) + s10
                    c (i + 1, m + 1) = c (i + 1, m+1) + s11
                  end do
                end do
              end do
            end do
          end do
          end subroutine
```

Отметим, что из-за хранения матриц по столбцам при $n = 1000$ на Intel Pentium III скорости работы первого и второго алгоритмов меняются местами; при $n = 2000$ на Intel Pentium III алгоритм 5 в 16 раз превосходит по скорости алгоритм 1.

3

Пути повышения производительности оперативной памяти

К сожалению, разрыв в производительности процессора и оперативной памяти увеличивается с каждым годом. Дело в том, что в процессор не только внедряют все более изощренные механизмы ускорения его работы, но и поднимают его тактовую частоту. Последнее не годится для оперативной памяти (по крайней мере, для ЭВМ общего назначения), поскольку это приводит к значительному повышению выделения тепла, которое не удается отводить обычными способами. (Заметим, что существуют ЭВМ, где использована значительная частота оперативной памяти, расплатой за что является необходимость охлаждения модулей памяти водяным либо ультразвуковым воздушным потоком.) Поэтому в настоящий момент частота процессора может более, чем в 10 раз превосходить частоту оперативной памяти. С учетом параллелизма в работе процессора это означает, что за время одного обращения к памяти он может выполнить более 20 инструкций (!), таких как перемножение пары вещественных чисел.

Для повышения производительности оперативной памяти применяют несколько приемов, идеологически схожих с методами ускорения работы процессора. Именно, оперативная память должна параллельно обрабатывать (т. е. принимать (за-

пись) или выдавать (чтение)) как можно большее количество данных:

- **Увеличение ширины шины данных**: переход от SIMM (32-битная шина) к модулям DIMM, DDR (64-, 128- или 256-битная шина) при построении подсистемы памяти. Это дает прирост в 2, 4 или 8 раз при доступе к последовательным данным (т. е. когда стратегия предвыборки строки кэша себя оправдывает).
- **Введение небольшой статической памяти SRAM для буферизации DRAM-модулей**: буферизованные DIMM-модули. Этот способ аналогичен введению кэша в процессор.
- **Введение конвейера в модули DRAM**. Используется та же идея, что и при введении конвейера в процессоры. Внутри модуля памяти находится несколько обрабатываемых обращений к памяти в разной степени готовности (формирование адреса, выбор банка, выборка данных, запись их на внешнюю шину и т. д.). Это позволяет модулю памяти принимать/выдавать данные каждый цикл шины (при условии оптимального функционирования конвейера). В силу этого такая память получила название synchronous DRAM (SDRAM). Для такой памяти в качестве времени доступа производители в рекламных целях пишут минимальный цикл шины, что дает фантастические времена доступа менее 10 нс (т. е. частота шины более 100 МГц). На самом же деле в модулях SDRAM использованы обычные микросхемы памяти, время доступа к которым — около 60 нс.
- **«Расслоение» оперативной памяти**. Поскольку процессор обменивается с памятью только блоками размером со строку кэша, то можно разделить этот блок на N частей (N обычно 2, 4, 8) и передать каждую из частей своей подсистеме памяти. В результате получаются N подсистем памяти, работающих параллельно. Если программа требует последовательные адреса памяти (т. е. стратегия предвыборки стро-

ки кэша себя оправдывает), то этот подход может в N раз увеличить производительность подсистемы памяти.

Подчеркнем еще раз, что все описанные приемы (как и в ситуации с процессором) дают выигрыш только для «правильно» устроенной программы, т. е. той, которая в основном обращается к последовательно расположенным данным и за каждое обращение читает/записывает блок памяти, не меньший ширины шины.

4

Организация данных во внешней памяти

Многопроцессорные системы иногда строятся на процессорах разных производителей. Эти процессоры решают общую задачу и обмениваются между собой данными через разделяемую память или коммуникационный канал. В этой ситуации необходимо согласование представления данных в памяти каждым из участвующих в обмене процессоров.

При доступе к данным (целочисленным или с плавающей точкой) основным вопросом является способ нумерации байтов в слове.

Если бы в 32-битной архитектуре минимальным адресуемым элементом оперативной памяти являлось 32-битное слово, то вопрос о нумерации байтов в слове не вставал бы. В реальной ситуации, когда минимальным адресуемым элементом оперативной памяти является байт (8-битное слово), данные большего размера образуются как объединение подряд идущих байт. Выбор нумерации байт в 32-битном (4 байта) слове может быть произвольным, что дает $24 = 4!$ способа. На практике используются только два: ABCD (называемый big-endian) и DCBA (называемый little-endian).

В модели **big-endian** байты в слове нумеруются от наиболее значимого к наименее значимому. В модели **little-endian** байты в слове нумеруются от наименее значимого к наиболее значимому. Примеры big-endian-процессоров: Motorola 68xxx, PowerPC

(по умолчанию), SPARC; пример little-endian-процессора: Intel 80x86. Процессоры PowerPC, Intel 80960x, ARM, SPARC (64-битные модели) могут работать как в big-endian-режиме, так и в little-endian.

Если процессоры, осуществляющие обмен данными, используют разный режим нумерации байтов, то потребуется преобразовывать все полученные или переданные данные. Практически все современные процессоры имеют для этой цели специальную инструкцию.

Все современные процессоры поддерживают в организации данных с плавающей точкой стандарт ANSI/IEEE 754-1985. Поэтому дополнительных проблем в этом случае не возникает. Необходимо только учитывать, что данные с плавающей точкой содержат несколько байт, поэтому на них также оказывает влияние способ нумерации байтов в слове.

5

Основные положения

В этой главе мы введем основные понятия, используемые при рассмотрении любой программной системы.

5.1. Основные определения

Определение. Программа — это описание на некотором формализованном языке алгоритма, решающего поставленную задачу. Программа является статической единицей, т. е. неизменяемой с точки зрения операционной системы, ее выполняющей.

Определение. Процесс — это динамическая сущность программы, ее код в процессе своего выполнения. Имеет

- собственные области памяти под код и данные,
- собственный стек,
- (в системах с виртуальной памятью) собственное отображение виртуальной памяти на физическую,
- собственное **состояние**.

Процесс может находиться в одном из следующих типичных **состояний** (точное количество и свойства того или иного состояния зависят от операционной системы):

1. «остановлен» — процесс остановлен и не использует процессор; например, в таком состоянии процесс находится сразу после создания;

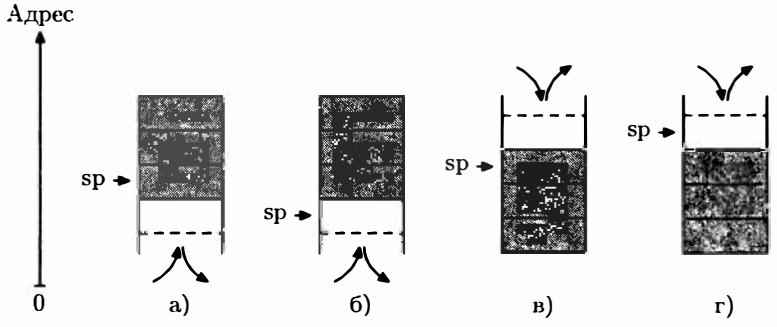


Рис. 5.1. Виды стеков

2. «терминирован» — процесс терминирован и не использует процессор; например, процесс закончился, но еще не удален операционной системой;
3. «ждет» — процесс ждет некоторого события (которым может быть аппаратное или программное прерывание, сигнал или другая форма межпроцессного взаимодействия);
4. «готов» — процесс не остановлен, не терминирован, не ожидает, не удален, но и не работает; например, процесс может не получать доступа к процессору, если в данный момент выполняется другой, более приоритетный процесс;
5. «выполняется» — процесс выполняется и использует процессор.

Определение. Стек (stack) — это область памяти, в которой размещаются локальные переменные, аргументы и возвращаемые значения функций. Вместе с областью статических данных полностью задает текущее состояние процесса. По логическому устройству стеки бывают четырех видов (рис. 5.1):

а) **full descending** — растет вниз (к младшим адресам памяти), указатель стека (SP, Stack Pointer) указывает на последнюю занятую ячейку стека (называемую вершиной стека);
б) **empty descending** — растет вниз, SP указывает на первую свободную ячейку стека за его вершиной;

в) **full ascending** — растет вверх, SP указывает на первую занятую ячейку стека;
г) **empty ascending** — растет вверх, SP указывает на первую свободную ячейку стека за его вершиной.

В ряде процессоров тип стека закреплен аппаратно (например, в Motorola 68xxx и Intel 80x86 используется стек 5.1а, в Intel 80960 — 5.1в), для других процессоров тип стека может выбираться разработчиками операционных систем произвольно (практически всегда выбирается стек 5.1а).

Определение. Виртуальная память — это «память», в адресном пространстве которой работает процесс. Виртуальная память:

1. позволяет увеличить объем памяти, доступной процессам за счет дисковой памяти;
2. обеспечивает выделение каждому из процессов виртуально непрерывного блока памяти, начинающегося (виртуально) с одного и того же адреса;
3. обеспечивает изоляцию одного процесса от другого.

Трансляцией виртуального адреса в физический занимается операционная система. Для ускорения этого процесса многие компьютерные системы имеют поддержку со стороны аппаратуры, которая может быть либо прямо в процессоре, либо в специальном устройстве управления памятью. Среди механизмов трансляции виртуального адреса преобладает *страничный*, при котором виртуальная и физическая память разбиваются на куски равного размера, называемые страницами (типичный размер — 4 Кб), между страницами виртуальной и физической памяти устанавливается взаимно-однозначное (для каждого процесса) отображение.

Определение. Межпроцессное взаимодействие — это тот или иной способ передачи информации из одного процесса в другой. Наиболее распространенными формами взаимодействия

являются (не все системы поддерживают перечисленные ниже возможности):

1. *Разделяемая память* — два (или более) процесса имеют доступ к одному и тому же блоку памяти. В системах с виртуальной памятью организация такого вида взаимодействия требует поддержки со стороны операционной системы, поскольку необходимо отобразить соответствующие блоки виртуальной памяти процессов на один и тот же блок физической памяти.
2. *Семафоры* — два (или более) процесса имеют доступ к одной переменной, принимающей значение 0 или 1. Сама переменная часто находится в области данных операционной системы и доступ к ней организуется посредством специальных функций.
3. *Сигналы* — это сообщения, доставляемые процессу посредством операционной системы. Процесс должен зарегистрировать обработчик этого сообщения у операционной системы, чтобы получить возможность реагировать на него. Часто операционная система извещает процесс сигналом о наступлении какого-либо сбоя, например, делении на 0, или о каком-либо аппаратном прерывании, например, прерывании таймера.
4. *Почтовые ящики* — это очередь сообщений (обычно — тех или иных структур данных), которые помещаются в почтовый ящик процессами и/или операционной системой. Несколько процессов могут ждать поступления сообщения в почтовый ящик и активизироваться по его поступлении. Требует поддержки со стороны операционной системы.

Определение. Событие — это оповещение процесса со стороны операционной системы о той или иной форме межпроцессного взаимодействия, например, о принятии семафором нужного значения, о наличии сигнала, о поступлении сообщения в почтовый ящик.

Создание, обеспечение взаимодействия, разделение процессорного времени требует от операционной системы значительных вычислительных затрат, особенно в системах с виртуальной памятью. Это связано прежде всего с тем, что каждый процесс имеет свое отображение виртуальной памяти на физическую, которое надо менять при переключении процессов и при обеспечении их доступа к объектам взаимодействия (общей памяти, семафорам, почтовым ящикам). Очень часто бывает так, что требуется запустить несколько копий одной и той же программы, например, для использования всех процессоров системы. В этом случае мы несем двойные накладные расходы: держим в оперативной памяти несколько копий кода одной программы и еще тратим дополнительное время на обеспечение их взаимодействия. Улучшает ситуацию введение задач.

Определение. Задача (или **поток**, или **нить**, **thread**) — это как бы одна из ветвей исполнения процесса:

- разделяет с процессом область памяти под код и данные,
- имеет собственный стек,
- (в системах с виртуальной памятью) разделяет с процессом отображение виртуальной памяти на физическую,
- имеет собственное состояние.

Таким образом, у двух задач в одном процессе вся память является разделяемой, и дополнительные расходы, связанные с разным отображением виртуальной памяти на физическую, сведены к нулю. Для задач так же, как для процессов, определяются понятия состояния задачи и межзадачного взаимодействия. Отметим, что для двух процессов обычно требуется организовать что-то общее (память, канал и т. д.) для их взаимодействия, в то время как для двух потоков часто требуется организовать что-то (например, область памяти), имеющее свое значение в каждом из них.

Определение. Ресурс — это объект, необходимый для работы процессу или задаче.

Определение. Приоритет — это число, приписанное операционной системой каждому процессу и задаче. Чем больше это число, тем важнее этот процесс или задача и тем больше процессорного времени он или она получит.

Если в операционной системе могут одновременно существовать несколько процессов или/и задач, находящихся в состоянии «выполняется», то говорят, что это многозадачная система, а эти процессы называют **параллельными**. Отметим, что если процессор один, то в каждый момент времени на самом деле реально выполняется только один процесс или задача. Система разделяет время между такими «выполняющимися» процессами/задачами, давая каждому из них квант времени, пропорциональный его приоритету.

Определение. Связывание (линковка, linkage) — это процесс превращения скомпилированного кода (объектных модулей) в загрузочный модуль (т. е. то, что может исполняться процессором при поддержке операционной системы). Различают:

- **статическое связывание**, когда код необходимых для работы программы библиотечных функций физически добавляется к коду объектных модулей для получения загрузочного модуля;
- **динамическое связывание**, когда в результирующем загрузочном модуле проставляются лишь ссылки на код необходимых библиотечных функций; сам код будет реально добавлен к загрузочному модулю только при его исполнении.

При статическом связывании загрузочные модули получаются очень большого размера. Поэтому подавляющее большинство современных операционных систем использует динамическое связывание, несмотря на то, что при этом начальная загрузка процесса на исполнение медленнее, чем при статическом связывании из-за необходимости поиска и загрузки кода нужных библиотечных функций (часто только тех из них, которые не были загружены для других процессов).

5.2. Виды ресурсов

По своей природе ресурсы можно разделить на

- **аппаратные**:
  - процессор,
  - область памяти,
  - периферийные устройства,
  - прерывания,
- **программные**:
  - программа,
  - данные,
  - файлы,
  - сообщения.

По своим характеристикам ресурсы разделяют на:

- **активные**:
  - способны изменять информацию (процессор),
- **пассивные**:
  - способны хранить информацию,
- **локальные**:
  - принадлежат одному процессу; время жизни совпадает с временем жизни процесса,
- **разделяемые**:
  - могут быть использованы несколькими процессами; существуют, пока есть хоть один процесс, который их использует,
- **постоянные**:
  - используются посредством операций «захватить» и «освободить»,
- **временные**:
  - используются посредством операций «создать» и «удалить».

Разделяемые ресурсы бывают (рис. 5.2):

- **некритичными**:

Рис. 5.2. Виды разделяемых ресурсов

  - могут быть использованы *одновременно* несколькими процессами (например, жесткий диск или канал Ethernet);
- **критичными**:
  - могут быть использованы *только одним* процессом, и пока этот процесс не завершит работу с ресурсом, последний не доступен другим процессам (например, разделяемая память, доступная на запись).

5.3. Типы взаимодействия процессов

По типу взаимодействия различают

- **независимые процессы** (рис. 5.3);

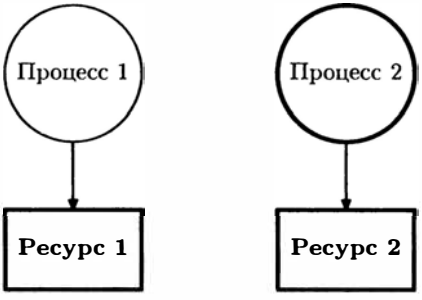


Рис. 5.3. Типы взаимодействия процессов: независимые процессы

- **сотрудничающие процессы** (рис. 5.4):
  - процессы, разделяющие только коммуникационный канал, по которому один передает данные, а другой их получает;
  - процессы, осуществляющие взаимную синхронизацию: когда работает один, другой ждет окончания его работы;
- **конкурирующие процессы** (рис. 5.5):
  - процессы, использующие совместно разделяемый ресурс;
  - процессы, использующие критические секции;
  - процессы, использующие взаимные исключения.

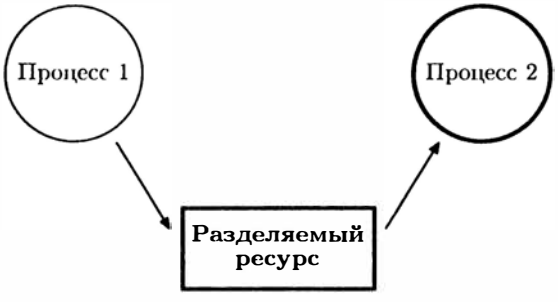


Рис. 5.4. Типы взаимодействия процессов: синхронизирующиеся процессы

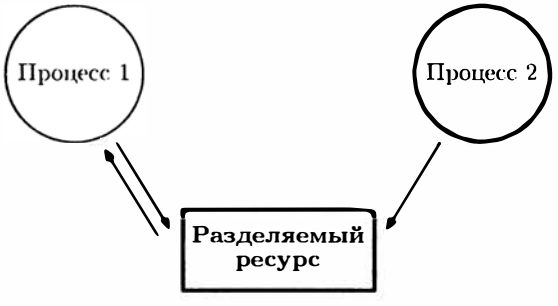


Рис. 5.5. Типы взаимодействия процессов: конкурирующие процессы

Определение. Критическая секция — это участок программы, на котором запрещается переключение задач для обеспечения исключительного использования ресурсов текущим процессом (задачей). Большинство операционных систем общего назначения не предоставляют такой возможности пользовательским программам.

Определение. Взаимное исключение (mutual exclusion, mutex) — это способ синхронизации параллельно работающих процессов (задач), использующих разделяемый постоянный критичный ресурс. Если ресурс занят, то системный вызов «захватить ресурс» переводит процесс (задачу) из состояния выполнения в состояние ожидания. Когда ресурс будет освобожден посредством системного вызова «освободить ресурс», то этот процесс (задача) вернется в состояние выполнения и продолжит свою работу. Ресурс при этом перейдет в состояние «занят».

Если процессы независимы (не имеют совместно используемых ресурсов), то синхронизация их работы не требуется. Если же процессы используют разделяемый ресурс, то их деятельность необходимо синхронизировать. Например, при использовании общего блока памяти проблемы могут возникнуть даже если один процесс (задача) только читает данные, а другой — только пишет. На рис. 5.6 показан пример, в котором два процесса P и Q увеличивают на 1 значение разделяемой ячейки памяти VAL. Результат работы будет различным в случае планирования задач 5.6 а и 5.6 б.

При синхронизации задач необходимо бороться с тремя проблемами:

1. «блокировка» («lockout»):
   - процесс (задача) ожидает ресурс, который никогда не освободится,
2. «тупик» («deadlock»):
   - два процесса (задачи) владеют каждый по ресурсу и ожидают освобождения ресурса, которым владеет другой процесс (задача),

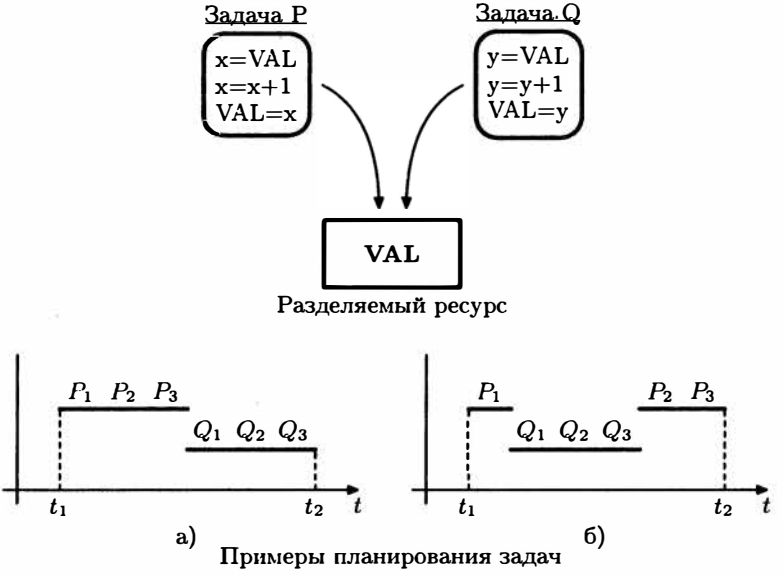


Рис. 5.6. Работа с разделяемым ресурсом

3. «застой» («starvation»):
   - процесс (задача) монополизировал процессор.

Для минимизации этих проблем используются следующие идеи.

- Количество ресурсов ограничено, поэтому нельзя допускать создания задач, для которых недостаточно ресурсов для выполнения.
- Задачи делятся на группы:
  - **неактивные** задачи, которым не хватило даже пассивных ресурсов, и **ожидающие** событий задачи; таким задачам активный ресурс (процессор) не дается вообще;
  - **готовые** задачи, у которых есть все необходимые пассивные ресурсы, но нет процессора; являются кандидатами на получение процессора в случае его освобождения;

– **выполняющиеся** задачи, у которых есть все необходимые пассивные ресурсы и процессор.

5.4. Состояния процесса

Рассмотрим более детально состояния процесса и переходы из одного состояния в другое (рис. 5.7). Состояния:

1. не существует,
2. не обслуживается,
3. готов,
4. выполняется,
5. ожидает ресурс,
6. ожидает назначенное время,
7. ожидает события.

Переходы из состояния в состояние:

1. переход 1–2: создание процесса
2. переход 2–1: уничтожение процесса
3. переход 2–3: активизация процесса диспетчером

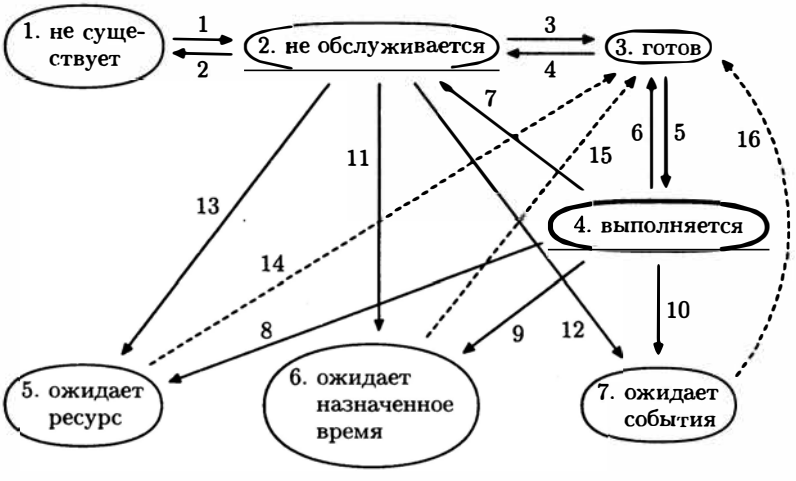


Рис. 5.7. Состояния процесса

4. переход 3–2: дезактивизация процесса
5. переход 3–4: загрузка на выполнение процесса диспетчером
6. переход 4–3: требование обслуживания от процессора другим процессом
7. переход 4–2: завершение процесса
8. переход 4–5: блокировка процесса до освобождения требуемого ресурса
9. переход 4–6: блокировка процесса до истечения заданного времени
10. переход 4–7: блокировка процесса до прихода события
11. переход 2–6: активизация процесса приводит к ожиданию временной задержки
12. переход 2–7: активизация процесса приводит к ожиданию события
13. переход 2–5: активизация процесса приводит к ожиданию освобождения ресурса
14. переход 5–3: активизация процесса из-за освобождения ожидавшегося ресурса
15. переход 6–3: активизация процесса по истечении заданного времени
16. переход 7–3: активизация процесса из-за прихода ожидавшегося события

6

Стандарты на операционные системы UNIX

На подавляющем большинстве параллельных вычислительных установок используется тот или иной вариант операционной системы UNIX. В этом разделе мы рассмотрим несколько существующих стандартов на UNIX-системы. Как и во многих других областях, стандарты стали появляться лишь *после* того, как уже был создан ряд диалектов UNIX. Основной целью введения стандартов является облегчение переноса программного обеспечения из одной системы в другую.

Мы кратко рассмотрим несколько стандартов. Подробно будет рассмотрен только наиболее развитый из них — POSIX.

6.1. Стандарт BSD 4.3

BSD 4.3 (Berkley Software Distribution версии 4.3) — это даже не стандарт на UNIX-системы, а эталонная реализация UNIX, выполненная в Калифорнийском университете г. Беркли в США в 1980-е годы. Она оказала очень большое влияние на дальнейшее развитие UNIX.

6.2. Стандарт UNIX System V Release 4

UNIX System V Release 4 — это тоже не стандарт на UNIX-системы, а эталонная реализация UNIX, выполненная фирмой

AT&T и включающая в себя особенности большинства существовавших UNIX-систем. Она оказала очень большое влияние на дальнейшее развитие UNIX.

6.3. Стандарт POSIX 1003

Стандарт POSIX (Portable Operating System Interface), разработанный IEEE (Institute of Electrical and Electronical Engineers), состоит из следующих частей.

1. POSIX 1003.1 — определяет стандарт на основные компоненты операционной системы, API (application interface) для процессов, файловой системы, устройств и т. д.
2. POSIX 1003.2 — определяет стандарт на основные утилиты.
3. POSIX 1003.1b — определяет стандарт на основные расширения «реального времени».
4. POSIX 1003.1c — определяет стандарт на задачи (threads).
5. POSIX 1003.1d — определяет стандарт на дополнительные расширения «реального времени» (такие, как, например, поддержка обработчиков прерываний).

Стандарт POSIX 1003 является наиболее всеобъемлющим из всех рассмотренных ранее. Большинство существующих UNIX-систем были адаптированы для удовлетворения этому стандарту.

6.4. Стандарт UNIX X/Open

Стандарт UNIX X/Open Portability Guide 3 — это стандарт на UNIX-системы, разработанный группой фирм-разработчиков программного обеспечения. Этот стандарт сокращенно называют XPG3. В 2000 г. уже вышел XPG6.

7

Управление процессами

В этой главе мы рассмотрим основные функции, позволяющие запускать новые процессы и управлять ими.

7.1. Функция fork

Функция `fork`, описанная в заголовочном файле `<unistd.h>` (тип возвращаемого значения описан в файле `<sys/types.h>`), позволяет «клонировать» текущий процесс. Прототип:

```
pid_t fork (void);
```

Эта функция создает процесс-потомок, полностью идентичный текущему процессу (отличие только в том, что у них разные идентификаторы и потомок не наследует файловые блокировки и очередь сигналов). Фактически создается копия виртуальной памяти процесса-родителя. Во многих системах физическая память реально не копируется, а просто создается новое отображение виртуальной памяти на ту же самую физическую. И только при записи в память создается копия изменяемой страницы для потомка. Функция возвращает идентификатор потомка в родительском процессе и 0 в порожденном процессе.

Рассмотрим задачу создания «демона» — процесса, работающего в фоновом режиме даже после выхода запустившего его пользователя из системы. Прежде всего необходимо запу-

стить фоновый процесс. Для этого используем функцию `fork` для создания процесса–потомка и завершим родительский процесс. Процесс–потомок продолжает работать в фоновом режиме, но он унаследовал от родителя связь с терминалом, с которого он был запущен, и будет терминирован операционной системой при окончании работы пользователя на этом терминале. Для создания не зависимого от терминала «демона» мы фактически повторим описанные выше действия еще раз: используем функцию `fork` для создания процесса–потомка и завершим родительский процесс. Процесс–потомок продолжает работать в фоновом режиме и вызывает функцию `daemon_process`, являющуюся телом «демона». В нашем примере эта функция печатает 10 раз с интервалом 2 секунды приветствие и завершает свою работу. Текст программы:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

void daemon_process ()
{
  int i;

  fprintf (stderr, "Daemon started!\n");
  /* Используем для тестирования конечный цикл */
  for (i = 0; i < 10; i++)
    {
      fprintf (stderr, "Hello %d!\n", i);
      sleep (2);
    }
  fprintf (stderr, "Daemon finished!\n");
}

int main ()
{
```

```
  pid_t pid;

  /* Клонировать себя */
  pid = fork ();
  if (pid == -1)
    {
      fprintf (stderr, "Cannot fork!\n");
      return 1;
    }
  else if (pid != 0)
    {
      /* Завершить родительский процесс */
      return 0;
    }

  /* Процесс-потомок продолжает работу */
  /* Клонировать себя */
  pid = fork ();
  if (pid == -1)
    {
      fprintf (stderr, "Cannot fork!\n");
      return 1;
    }
  else if (pid != 0)
    {
      /* Завершить родительский процесс */
      return 0;
    }

  /* Процесс-потомок продолжает работу */
  daemon_process ();

  return 0;
}
```

7.2. Функции execl, execv

Функции `execl`, `execv`, описанные в файле `<unistd.h>`, позволяют заменить текущий процесс новым. Прототипы:

```
int execl (const char *path, const char *arg, ...);
int execv (const char *path, char *const argv[]);
```

где входные параметры:

- `path` — имя исполнимого файла процесса,
- `arg`, ... — список аргументов (завершаемый нулевым указателем),
- `argv` — указатель на список аргументов (завершаемый нулевым указателем).

В случае успеха эта функция не возвращает управления (поскольку текущий процесс больше не существует), в случае ошибки она возвращает -1.

Пример использования для запуска программы см. в разделе 7.3.

7.3. Функция waitpid

Функция `waitpid`, описанная в файле `<sys/wait.h>` (тип возвращаемого значения описан в файле `<sys/types.h>`), позволяет ждать окончания порожденного процесса. Прототип:

```
pid_t waitpid (pid_t pid, int *status, int options);
```

Эта функция останавливает выполнение текущего процесса до завершения процесса-потомка, указанного аргументом `pid`:

- если `pid` > 0 (наиболее используемая форма), то ожидать окончания потомка с идентификатором `pid`;
- если `pid` $= 0$, то ожидать окончания любого потомка, принадлежащего к той же группе, что и текущий процесс;
- если `pid` < -1, то ожидать окончания любого потомка, принадлежащего к группе $|$`pid`$|$;
- если `pid` $= -1$, то ожидать окончания любого потомка.

Причину завершения процесса можно узнать через переменную `status` с помощью специальных макросов (см. примеры ниже). Аргумент `options` задает поведение функции в случае, если процесс-потомок уже завершился или остановлен. В случае успеха функция возвращает идентификатор закончившегося процесса, в случае ошибки — -1.

Рассмотрим в качестве примера следующую задачу: требуется запустить заданную программу (мы используем `/bin/ls`) с заданным параметром (`-l`) в виде отдельного процесса и дождаться его окончания. Для этого мы с помощью функции `fork` создадим процесс-потомок, в котором сразу выполним `execl` для его замены на указанный процесс. Напомним, что первым аргументом любой программы (в нашем случае `/bin/ls`) является ее собственное путевое имя, поэтому список аргументов `execl` выглядит следующим образом:

1. `"/bin/ls"` — имя исполнимого файла процесса,
2. `"/bin/ls"` — первый аргумент,
3. `"-l"` — второй аргумент,
4. `0` — нулевой указатель, обозначающий конец списка аргументов `execl`.

В родительском процессе мы будем ожидать окончания процесса-потомка с помощью `waitpid`. После этого можно проверить причину окончания процесса, используя содержимое переменной `status` и специальные макросы:

- `WIFEXITED (status)` — возвращает истинное значение, если программа завершилась нормально (например, с помощью оператора `return` в функции `main` или посредством вызова функции `exit`). О второй возможной причине окончания процесса см. раздел 7.5, с. 88.
- `WEXITSTATUS (status)` — дает значение, которое вернула программа операционной системе, т. е. аргумент оператора `return` в функции `main` или аргумент функции `exit`. Этот макрос можно использовать только в случае, если значение `WIFEXITED (status)` истинно. В UNIX-системах признаком

нормального завершения программы является нулевое значение WEXITSTATUS (status).

Текст программы:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main ()
{
  pid_t pid;
  int status;

  /* Клонировать себя */
  pid = fork ();
  if (pid == -1)
    {
      fprintf (stderr, "Cannot fork!\n");
      return 1;
    }
  if (pid == 0)
    {
      /* Запускаем программу в порожденном процессе */
      execl ("/bin/ls", "/bin/ls", "-l", 0);
      /* Если этот код выполняется, значит
         ошибка при запуске */
      fprintf (stderr, "Cannot exec!\n");
      return 2;
    }

  /* В родительском процессе: ожидаем завершения
     потомка */
  if (waitpid (pid, &status, 0) <= 0)
    {
```

```
      fprintf (stderr,
        "Cannot waitpid child, terminating...\n");
      return 3;
    }

  /* Проверяем успешность завершения потомка */
  if (!WIFEXITED (status))
    {
      fprintf (stderr,
        "Child finished abnormally, terminating...\n");
      return 4;
    }

  /* Проверяем код завершения потомка */
  if (WEXITSTATUS (status))
    {
      fprintf (stderr, "Child finished with non-zero \
return code, terminating...\n");
      return 5;
    }

  /* завершаем работу */
  return 0;
}
```

7.4. Функция kill

Функция `kill`, описанная в заголовочном файле `<signal.h>` (тип первого аргумента описан в файле `<sys/types.h>`), позволяет послать сигнал процессу. Прототип:

```
int kill (pid_t pid, int sig);
```

Поведение функции различно в зависимости от `pid`:

- если `pid` > 0 (наиболее используемая форма), то сигнал с номером `sig` будет послан процессу с идентификатором `pid`;

- если pid = 0, то сигнал с номером sig будет послан всем процессам, принадлежащим к той же группе, что и текущий процесс;
- если pid < −1, то сигнал с номером sig будет послан всем процессам, принадлежащим к группе |pid|;
- если pid = −1, то сигнал с номером sig будет послан всем процессам (за исключением процесса с номером 1, которому часто вообще нельзя послать сигнал).

В случае успеха функция возвращает 0, в случае ошибки — −1. Процесс-получатель сигнала должен установить обработчик для сигнала (например, с помощью функции signal, см. раздел 7.5). В противном случае будет вызван стандартный обработчик, который во многих операционных системах терминирует процесс.

7.5. Функция signal

Функция signal, описанная в заголовочном файле <signal.h>, позволяет указать функцию, которую следует вызывать при поступлении сигнала с заданным номером. Прототип:

```
sighandler_t signal (int signum,
  sighandler_t handler);
```

где тип обработчика сигнала есть

```
typedef void (*sighandler_t) (int);
```

Функция возвращает старый обработчик сигнала.

Рассмотрим задачу создания «отказоустойчивого» фонового процесса («демона»). Требуется разработать программу, осуществляющую запуск и мониторинг состояния заданного процесса (в нашем примере -- "./daemon"). В случае (нормального или аварийного) завершения последнего программа должна осуществлять его перезапуск. Приведем вначале тексты программ, а затем дадим необходимые пояснения. Программа, осуществляющая мониторинг состояния демона:

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

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <errno.h>

/* Номер запущенного процесса */
static pid_t pid;

int daemon_process ()
{
  static const char * program = "./daemon";
  static char * const program_args[] = { "./daemon", 0};
  pid_t res;
  int status;
  int i;

  fprintf (stderr, "Daemon started!\n");
  /* Используем для тестирования конечный цикл */
  for (i = 0; i < 10; i++)
    {
      /* Клонировать себя */
      pid = fork ();
      if (pid == -1)
        {
          fprintf (stderr, "Cannot fork!\n");
          return 1;
        }
      if (pid == 0)
        {
          /* Запускаем программу в порожденном процессе*
          execv (program, program_args);
          /* Если этот код выполняется, значит
             ошибка при запуске */
```

```
        fprintf (stderr, "Cannot exec!\n");
        return 2;
      }

    /* В родительском процессе: ожидаем завершения
       потомка; при этом игнорируем все прерывания
       от сигналов */
    while (((res = waitpid (pid, &status, 0)) <= 0)
           && (errno == EINTR));

    if (res != pid)
      {
        fprintf (stderr,
          "Cannot waitpid child, terminating...\n");
        return 3;
      }

    /* Проверяем причину завершения потомка */
    if (WIFEXITED (status))
      {
        fprintf (stderr,
          "Child process %d exited with code %d\n",
          pid, WEXITSTATUS (status));
      }
    else if (WIFSIGNALED (status))
      {
        fprintf (stderr,
          "Child process %d killed by signal %d\n",
          pid, WTERMSIG (status));
      }
    else
      {
        /* никогда! */
        fprintf (stderr, "Child process %d terminanted\
by unknown reason\n",
```

```
                    pid);
        }
    }
  fprintf (stderr, "Daemon finished!\n");
  return 0;
}

/* Обработчик сигналов */
void signal_handler (int signum)
{
  switch (signum)
    {
      case SIGTERM:
      case SIGKILL:
        /* Ликвидируем потомка */
        fprintf (stderr,
          "Killing child process %d\n", pid);
        if (kill (pid, 9))
          fprintf (stderr,
            "Error while killing child process %d\n",
            pid);
        fprintf (stderr, "Daemon finished!\n");
        exit(0);
        break ;

      default :
        break;
  }
}

int main ()
{
  pid_t pid;

  /* Игнорируем сигнал SIGTTOU - остановка фонового
```

```
   процесса при попытке вывода на свой управляющий
   терминал */
signal (SIGTTOU, SIG_IGN);
/* Игнорируем сигнал SIGTTIN - остановка фонового
   процесса при попытке ввода со своего управляющего
   терминала */
signal (SIGTTIN, SIG_IGN);
/* Игнорируем сигнал SIGTSTP - разрыв связи с
   управляющим терминалом */
signal (SIGHUP, SIG_IGN);
/* Игнорируем сигнал SIGTSTP - остановка процесса
   нажатием на клавиатуре Ctrl+Z */
signal (SIGTSTP, SIG_IGN);

/* Устанавливаем свой обработчик сигнала SIG_TERM
   - завершение процесса */
signal (SIGTERM,  signal_handler);

/* Клонировать себя */
pid = fork ();
if (pid == -1)
  {
    fprintf (stderr, "Cannot fork!\n");
    return 1;
  }
else if (pid != 0)
  {
    /* Завершить родительский процесс */
    return 0;
  }

/* Процесс-потомок продолжает работу */
/* Клонировать себя */
pid = fork ();
if (pid == -1)
```

```
    {
      fprintf (stderr, "Cannot fork!\n");
      return 1;
    }
  else if (pid != 0)
    {
      /* Завершить родительский процесс */
      return 0;
    }

  /* Процесс-потомок продолжает работу */
  return daemon_process ();
}
```

Программа–демон:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main ()
{
  pid_t pid;
  int * p = 0;

  pid = getpid ();
  fprintf (stderr, "Program started, pid = %d\n", pid);
  /* подождать указанное время */
  sleep (10);
  fprintf (stderr, "Program died, pid = %d\n", pid);
  /* Сделать segmentation fault */
  * p = 0;

  return 0;
}
```

Рассмотрим вначале подробнее программу-демон. Она печатает свой идентификатор, ждет 10 секунд, печатает сообщение о своем окончании и производит запись по нулевому адресу. Это приводит к посылке ей операционной системой сигнала с номером 11 (SIGSEGV – segmentation fault), стандартный обработчик которого терминирует программу.

Рассмотрим подробнее программу, осуществляющую мониторинг состояния демона. Функция `main` представляет собой улучшенный вариант описанной в разделе 7.1, с. 81 программы, запускающей фоновый процесс. Отличие состоит в том, что мы с помощью функции `signal` с предопределенным значением `SIG_IGN` заказываем игнорирование ряда сигналов, связывающих фоновый процесс с терминалом, с которого его запустили. Также посредством `signal` мы регистрируем новый обработчик `signal_handler` для сигнала 15 (SIGTERM). Это делается для того, чтобы при получении этого сигнала программа, осуществляющая мониторинг состояния демона, завершала его работу.

Функция `daemon_process`, работающая в фоновом процессе, в цикле (конечном в нашем примере и обычно бесконечном в реальном применении) осуществляет следующие действия. В начале цикла с помощью функций `fork` и `execv` осуществляется запуск нового процесса `./daemon` Затем с помощью функции `waitpid` программа переходит в режим ожидания завершения этого процесса. Процесс ожидания может быть прерван поступившим сигналом. Эта ситуация проверяется по состоянию переменной `errno`, содержащей код ошибки, возникшей при работе последней вызванной библиотечной функции. Если значение `errno` равно `EINTR`, то ожидание было прервано сигналом, и мы его возобновляем. Признаком успешного окончания цикла ожидания является возврат функцией `waitpid` идентификатора ожидавшегося процесса. После этого `daemon_process` печатает информацию о причине окончания процесса, используя содержимое переменной `status` и специальные макросы:

- `WIFEXITED (status)` и `WEXITSTATUS (status)` — см. раздел 7.3, с. 84.
- `WIFSIGNALED (status)` — возвращает истинное значение, если программа завершилась вследствие получения сигнала.
- `WTERMSIG (status)` — дает номер сигнала, полученного процессом. Этот макрос можно использовать только в случае, если значение `WIFSIGNALED (status)` истинно.

Функция `signal_handler`, регистрируемая в функции `main` как обработчик сигнала SIGTERM, посылает с помощью функции `kill` сигнал 9 (SIGKILL) запущенному процессу. Это позволяет терминировать процесс–демон, послав сигнал SIGTERM (например, с помощью программы `kill`) процессу, осуществляющему его мониторинг.

8

Синхронизация и взаимодействие процессов

Доступ процессов (задач) к различным ресурсам (особенно разделяемым) в многозадачных системах требует синхронизации действий этих процессов (задач). Способы осуществления взаимодействия подразделяют на:

- **безопасное взаимодействие**, когда обмен данными осуществляется посредством «объектов» взаимодействия, предоставляемых системой; при этом целостность информации и неделимость операций с нею (т. е. отсутствие нежелательного переключения задач) неявно обеспечиваются системой; примерами таких «объектов» взаимодействия являются семафоры, сигналы и почтовые ящики;
- **небезопасное взаимодействие**, когда обмен данными осуществляется посредством разделяемых ресурсов (например, общих переменных), не зависимых от системных объектов взаимодействия; при этом целостность информации и неделимость явно обеспечиваются самим приложением (в подавляющем большинстве случаев — посредством того или иного системного объекта синхронизации и взаимодействия).

Поскольку для любого типа взаимодействия требуются системные объекты синхронизации, то все имеющиеся операционные системы предоставляют приложениям некоторый набор таких

объектов. Ниже мы рассмотрим самые распространенные из них.

8.1. Разделяемая память

Определение. Разделяемая память — это область памяти, к которой имеют доступ несколько процессов. Взаимодействие через разделяемую память является базовым механизмом взаимодействия процессов, к которому сводятся все остальные. Оно, с одной стороны, является самым быстрым видом взаимодействия, поскольку процессы напрямую (т. е. без участия операционной системы) передают данные друг другу. С другой стороны, оно является небезопасным, и для обеспечения правильности передачи информации используются те или иные объекты синхронизации.

В системах с виртуальной памятью существуют два подхода к логической организации разделяемой памяти:

1. Разделяемая память находится в адресном пространстве операционной системы, а виртуальные адресные пространства процессов отображаются на нее. В зависимости от реализации операционной системы это может приводить к переключению задач при работе с разделяемой памятью (поскольку она принадлежит ядру, а не процессу) и фиксации разделяемой памяти в физической памяти (поскольку само ядро не участвует в страничном обмене).
2. Разделяемая память логически представляется как файл, отображенный на память (т. е. файл, рассматриваемый как массив байтов в памяти). При этом разделяемая память полностью находится в пользовательском адресном пространстве и отсутствуют дополнительные задержки при доступе к ней.

В силу большей эффективности рекомендуется использовать второй способ работы с разделяемой памятью.

В системах с виртуальной памятью над разделяемой памятью определены следующие элементарные операции.

- создать (или открыть) разделяемую память, при этом разделяемая память появляется в процессе как объект, но доступ к ее содержимому еще невозможен;
- подсоединить разделяемую память к адресному пространству процесса, при этом происходит отображение разделяемой памяти на виртуальное адресное пространство процесса; после этой операции разделяемая память доступна для использования;
- отсоединить разделяемую память от адресного пространства процесса, после этой операции доступ к содержимому разделяемой памяти невозможен;
- удалить (или закрыть) разделяемую память, реально разделяемая память будет удалена, когда с ней закончит работать последний из процессов.

В системах без виртуальной памяти эти операции тривиальны.

Отметим, что в разных процессах разделяемая память может быть отображена в разные адреса виртуальной памяти. Следовательно, в разделяемой памяти нельзя хранить указатели на другие элементы разделяемой памяти (например, классическую реализацию однонаправленного списка нельзя без изменений хранить в разделяемой памяти).

8.1.1. Функция shmget

Функция `shmget`, описанная в файле `<sys/shm.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет создать (или открыть существующую) область разделяемой памяти указанного размера и с указанными правами доступа. Прототип:

```
int shmget (key_t key, size_t size, int shmflg);
```

где входные параметры:

- `key` — числовой идентификатор области разделяемой памяти;
- `size` — размер области разделяемой памяти (округляется вверх до величины, кратной размеру страницы виртуальной памяти);
- `shmflg` — атрибуты, используемые при создании области разделяемой памяти (флаги и права доступа).

Функция возвращает идентификатор области разделяемой памяти (не путать с `key`) или -1 в случае ошибки.

8.1.2. Функция shmat

Функция `shmat`, описанная в файле `<sys/shm.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет подсоединить указанную область разделяемой памяти к адресному пространству процесса. Прототип:

```
void *shmat (int shmid, const void *shmaddr,
  int shmflg);
```

где входные параметры:

- `shmid` — идентификатор области разделяемой памяти;
- `shmaddr` — пожелания процесса об адресе, начиная с которого следует подсоединить область разделяемой памяти, или 0, если адрес безразличен (операционная система может игнорировать этот аргумент);
- `shmflg` — права доступа к области разделяемой памяти.

Функция возвращает адрес, начиная с которого подсоединена область разделяемой памяти, или -1 в случае ошибки.

8.1.3. Функция shmctl

Функция `shmctl`, описанная в файле `<sys/shm.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет управлять областью разделяемой памяти (в частности, удалять ее). Прототип:

```
int shmctl (int shmid, int cmd,
  struct shmid_ds *buf);
```

где входные параметры:

- `shmid` — идентификатор области разделяемой памяти;
- `cmd` — задает операцию над областью разделяемой памяти;
- `buf` — аргументы операции (также служит для возврата информации о результате операции).

Функция возвращает 0 в случае успеха или -1 в случае ошибки.

8.2. Семафоры

Определение. Семафор — это объект синхронизации, задающий количество пользователей (задач, процессов), имеющих одновременный доступ к некоторому ресурсу. С каждым семафором связаны счетчик (значение семафора) и очередь ожидания (процессов, задач, ожидающих принятие счетчиком определенного значения). Различают:

- *двоичные (булевские) семафоры* — это механизм взаимного исключения для защиты критичного разделяемого ресурса; начальное значение счетчика такого семафора равно 1;
- *счетные семафоры* — это механизм взаимного исключения для защиты ресурса, который может быть одновременно использован не более, чем ограниченным фиксированным числом задач n; начальное значение счетчика такого семафора равно n.

Над семафорами определены следующие элементарные операции (ниже $k = 1$ для булевских семафоров):

- взять k единиц из семафора, т. е. уменьшить счетчик на k (если в счетчике нет k единиц, то эта операция переводит задачу в состояние ожидания наличия как минимум k единиц в семафоре, и добавляет ее в конец очереди ожидания этого семафора);
- вернуть k единиц в семафор, т. е. увеличить счетчик на k (если семафор ожидается другой задачей и ей требуется не

более, чем новое текущее значение счетчика единиц, то она может быть активизирована, удалена из очереди ожидания и может вытеснить текущую задачу, например, если ее приоритет выше);

- попробовать взять k единиц из семафора (если в счетчике $\geq k$ единиц, то взять k единиц из него, иначе вернуть признак занятости семафора без перевода задачи в состояние ожидания);
- проверить семафор, т. е. получить значение счетчика;
- блокировать семафор, т. е. взять из него столько единиц, сколько в нем есть (при этом иногда бывают две разновидности этой операции: взять столько, сколько есть в данный момент, или взять столько, сколько есть в начальный момент, т. е. максимально возможное количество, именно последнее обычно называют блокировкой, поскольку такая задача будет монопольно владеть ресурсом);
- разблокировать семафор, т. е. вернуть столько единиц, сколько всего было взято данной задачей по команде блокировать.

Логическая структура двоичных семафоров особенно проста. Счетчик семафора s инициализируется 1 при создании. Для доступа к нему определены две примитивные операции:

- $Get(s)$ — взять (или закрыть) семафор s, т. е. запросить ресурс; эта операция вычитает из счетчика 1;
- $Put(s)$ — вернуть (или открыть) семафор, т. е. освободить ресурс; эта операция прибавляет к счетчику 1.

Эти операции **неделимы**, т. е. переключение задач во время их исполнения запрещено. В процессе работы состояние счетчика может быть:

- 1 — ресурс свободен,
- 0 — ресурс занят, очередь ожидания пуста,
- $m < 0$ — ресурс занят, в очереди ожидания находятся $|m|$ задач.

Рассмотрим пример (рис. 8.1).

1. *A.Get* — задача *A* завладела ресурсом,
2. *C.Get* — задача *C* запросила ресурс, который занят, следовательно, задача *C* заблокирована и помещена в очередь ожидания,
3. *B.Get* — задача *B* запросила ресурс, который занят, следовательно, задача *B* заблокирована и помещена в очередь ожидания,
4. *A.Put* — задача *A* освободила ресурс, который был передан первой задаче в очереди ожидания, т. е. *C* (которая в свою очередь активизирована),
5. *C.Put* — задача *C* освободила ресурс, который был передан первой задаче в очереди ожидания, т. е. *B* (которая в свою очередь активизирована),

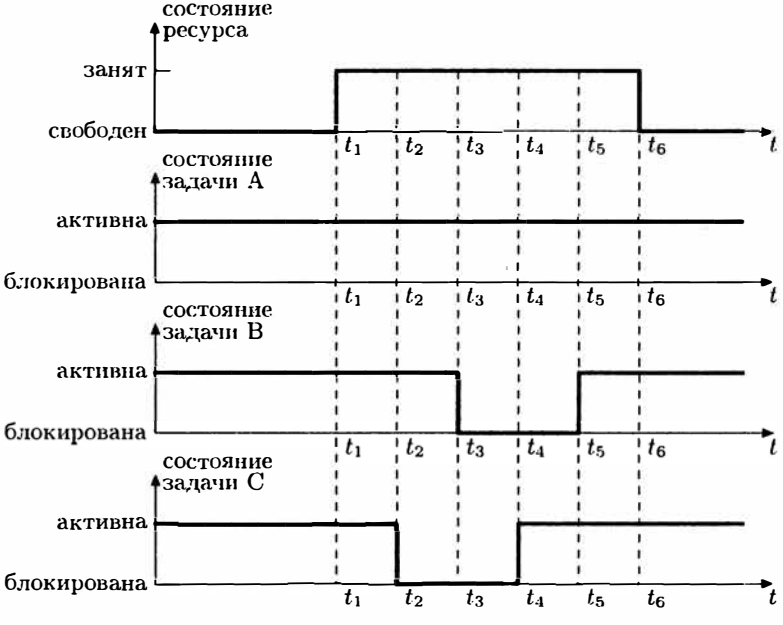


Рис. 8.1. Пример работы с семафором

6. *B.Put* — задача *B* освободила ресурс, который будет передан первой задаче, которая вызовет *Get*.

8.2.1. Функция semget

Функция `semget`, описанная в файле `<sys/sem.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет создать (или открыть существующий) семафор (или группу семафоров) с указанными правами доступа. Прототип:

```
int semget (key_t key, size_t semnum, int semflg);
```

где входные параметры:

- `key` — числовой идентификатор набора семафоров;
- `size` — количество семафоров в наборе;
- `semflg` — атрибуты, используемые при создании набора семафоров (флаги и права доступа).

Функция возвращает идентификатор набора семафоров (не путать с `key`) или -1 в случае ошибки.

8.2.2. Функция semop

Функция `semop`, описанная в файле `<sys/sem.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет изменять значения семафора. Прототип:

```
int semop (int semid, struct sembuf *sops,
  size_t nsops);
```

где входные параметры:

- `semid` — идентификатор набора семафоров;
- `sops` — указатель на массив структур типа `sembuf`, задающих операцию над семафорами из набора;
- `nsops` — количество элементов в массиве, на который указывает `sops`.

Функция возвращает 0 в случае успеха или -1 в случае ошибки.

8.2.3. Функция semctl

Функция `semctl`, описанная в файле `<sys/sem.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет управлять семафором (в частности, инициализировать и удалять его). Прототип:

```
int semctl (int semid, int semnum, int cmd, ...);
```

где входные параметры:

- `semid` — идентификатор набора семафоров;
- `semnum` — номер семафора из набора, над которым надо произвести операцию;
- `cmd` — задает операцию над семафором `semnum`.

Функция возвращает неотрицательное число, зависящее от операции `cmd`, или -1 в случае ошибки.

8.2.4. Пример использования семафоров и разделяемой памяти

Рассмотрим пример типичного клиент-серверного приложения. Процесс-клиент (текст которого находится в файле `reader.c`) считывает строку со стандартного ввода и передает ее через разделяемую память процессу-серверу (текст которого находится в файле `writer.c`) выводящему ее на стандартный вывод. Для организации взаимного исключения при доступе к критическим разделяемым ресурсам (переменным `done`, `msglen`, `msgbuf`, находящимся в разделяемой памяти), используется семафор, идентификатор которого тоже содержится в разделяемой памяти. Как клиент, так и сервер используют одни и те же функции по работе с разделяемой памятью и семафором, находящиеся в файлах `shdata.h` и `shdata.c`.

Общая часть — заголовочный файл `shdata.h`, в котором описаны структура разделяемой памяти (`SHDATA`) и функции работы с ней:

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

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define BUF_LEN 256

typedef struct _shdata_
{
  int sem_id;           /* идентификатор семафора */
  int done;             /* признак окончания работы */
  int msglen;           /* длина сообщения */
  char msgbuf[BUF_LEN]; /* буфер для хранения сообщения*/
} SHDATA;

/* Начать работу с разделяемой памятью */
SHDATA * shdata_open (void);
/* Закончить работу с разделяемой памятью */
int shdata_close (SHDATA * data);
/* Обеспечить исключительную работу с разделяемой памятью
   текущему процессу.
   Возвращает не 0 в случае ошибки. */
int shdata_lock (SHDATA * data);
/* Разрешить другим процессам работать с разделяемой
   памятью. Возвращает не 0 в случае ошибки. */
int shdata_unlock (SHDATA * data);
```

Общая часть (для клиента и сервера) находится в файле `shdata.c` (см. пояснения ниже):

```
#include "shdata.h"
#include <errno.h>
```

```
/* Идентификатор разделяемой памяти */
#define SHM_KEY 1221
/* Идентификатор семафора */
#define SEM_KEY (1221+1)

/* Идентификатор разделяемой памяти */
static int shm_id;

/* Идентификатор семафора */
static int sem_id;

/* "Открыть" разделяемую память с ключом key и размером
   size. Возвращает указатель на память, 0 в случае
   ошибки. Значение *pcreated будет истинным, если
   текущий процесс создал разделяемую память. */
static void *
shm_open (int key, int size, int *pcreated)
{
  void * res = 0;/* результат */
  *pcreated = 0; /* память не создали */

  /* Попытаться открыть разделяемую память */
  if ((shm_id = shmget (key, size, 0)) == -1)
    {
      /* Не удалось, рассмотрим причину */
      if (errno == ENOENT)
        {
          /* Запрошенная разделяемая память не существует.
             Создаем ее. */
          if ((shm_id = shmget (key, size,
                                IPC_CREAT | 0664)) == -1)
            {
              fprintf (stderr,
                       "Cannot create shared memory\n");
              return 0;
```

```
            }
          *pcreated = 1; /* память была создана */
        }
      else
        {
          fprintf (stderr, "Cannot get shared memory\n")
          return 0;
        }
    }

  /* Подсоединяем разделяемую память к адресному
     пространству процесса */
  if ((res = (void *)shmat (shm_id, 0, 0)) == (void *)-1
    {
      fprintf (stderr, "Cannot attach shared memory\n");
      return 0;
    }
  return res;
}

/* "Закрыть" разделяемую память.
   Возвращает не 0 в случае ошибки. */
static int
shm_close (void * mem)
{
  /* Отсоединяем разделяемую память от адресного
     пространства процесса */
  if (shmdt (mem) == -1)
    {
      fprintf (stderr, "Cannot detach shared memory\n")
      return 1;
    }

  /* Удаляем разделяемую память. Реально удаление
     произойдет, когда закончится последний процесс,
```

```
      использующий разделяемую память. Если несколько
      процессов вызывают удаление, то успешным будет
      только первый вызов, поэтому возвращаемое
      значение функции не проверяется. */
  shmctl (shm_id, IPC_RMID, 0);
  return 0;
}

/* "Открыть" семафор с ключом key
   Возвращает идентификатор семафора, -1 в случае ошибки.
   Значение *pcreated будет истинным, если текущий
   процесс создал семафор. */
static int
sem_open (int key, int *pcreated)
{
  int id;         /* результат */
  *pcreated = 0; /* семафор не создан */

  /* Попытаться получить семафор */
  if ((id = semget (key, 1, 0)) == -1)
    {
      /* Не удалось, рассмотрим причину */
      if (errno == ENOENT)
        {
          /* Запрошенный семафор не существует.
             Создаем его. */
          if ((id = semget (key, 1, IPC_CREAT | 0664))
              == -1)
            {
              fprintf (stderr,
                       "Cannot create semaphore\n");
              return id;
            }
          *pcreated = 1; /* семафор был создан */
        }
```

```
      else
        {
          fprintf (stderr, "Cannot get semaphore\n");
          return id;
        }
    }

  /* Инициализировать семафор, если текущий процесс
     его создал */
  if (*pcreated)
    {
      /* Структура данных для управления семафором */
#ifdef _SEM_SEMUN_UNDEFINED
       union semun
       {
         /* Значение для команды SETVAL */
         int val;
         /* Буфер для команд IPC_STAT и IPC_SET */
         struct semid_ds *buf;
         /* Массив для команд GETALL и SETALL */
         unsigned short int *array;
         /* Буфер для команды IPC_INFO */
         struct seminfo *__buf;
       };
#endif
      union semun s_un;

      /*  Начальное значение семафора */
      s_un.val = 1;
      /*  Установить начальное значение семафора */
      if (semctl (id, 0, SETVAL, s_un) == -1)
        {
          fprintf (stderr, "Cannot init semaphore\n");
          /* С семафором работать нельзя, заканчиваем */
          return -1;
```

```
        }
    }

  return id;
}

/* "Закрыть" семафор с идентификатором id.
   Возвращает не 0 в случае ошибки. */
static int
sem_close (int id)
{
  /* Удаляем семафор. Удаление происходит сразу. Все
     ожидающие семафор процессы активизируются,
     ожидание завершается с ошибкой EIDRM -
     Error IDentifier ReMoved. У них вся последующая
     работа с этим семафором будет давать ошибку,
     поэтому возвращаемое значение функции
     не проверяется. */
  semctl (id, 0, IPC_RMID, 0);
  return 0;
}

/* Заблокировать семафор с идентификатором id.
   Возвращает не 0 в случае ошибки. */
static int
sem_lock (int id)
{
  /* Структура данных для операций с семафором */
  struct sembuf s_buf = { 0, -1, 0};

  if (semop (id, &s_buf, 1) == -1)
    {
      /* Проверяем причину */
      if (errno == EIDRM)
        {
```

```
          /* Семафор был удален другим процессом. */
          return -1;
        }
      /* Еще может быть EINTR - ожидание прервано
         сигналом */

      fprintf (stderr, "Cannot lock semaphore\n");
      return 1; /* ошибка */
    }
  return 0;
}

/* Разблокировать семафор с идентификатором id.
   Возвращает не 0 в случае ошибки. */
static int
sem_unlock (int id)
{
  /* Структура данных для операций с семафором */
  struct sembuf s_buf = { 0, 1, 0};

  if (semop (id, &s_buf, 1) == -1)
    {
      fprintf (stderr, "Cannot unlock semaphore\n");
      return 1; /* ошибка */
    }
  return 0;
}


/* Начать работу с разделяемой памятью */
SHDATA * shdata_open (void)
{
  /* Результат */
  SHDATA * res = 0;
  /* Истинно, если текущий процесс создал разделяемую
```

```
   память */
int shm_created;
/* Истинно, если текущий процесс создал семафор */
int sem_created;

/* "Открыть" разделяемую память */
res = (SHDATA*) shm_open (SHM_KEY, sizeof (SHDATA),
                          &shm_created);
if (!res)
  return 0;   /* ошибка */

/* "Открыть" семафор */
sem_id = sem_open (SEM_KEY, &sem_created);
if (sem_id == -1)
  {
    /* Ошибка, удаляем разделяемую память и
       заканчиваем */
    shm_close (res);
    return 0;   /* ошибка */
  }

/* Заблокировать семафор */
if (sem_lock (sem_id))
  {
    /* Ошибка, удаляем все и заканчиваем */
    shdata_close (res);
    return 0;   /* ошибка */
  }

/* Теперь только один процесс работает с памятью */
/* Пусть инициализацией структур данных занимается
   тот процесс, который создал разделяемую память */
if (shm_created)
  {
    /* обнулить область разделяемой памяти */
```

```
      memset ((char*)res, 0, sizeof (SHDATA));
      /* занести идентификатор семафора в разделяемую
         память */
      res->sem_id = sem_id;
    }

  /* Разблокировать семафор */
  if (sem_unlock (sem_id))
    {
      /* Ошибка, удаляем все и заканчиваем */
      shdata_close (res);
      return 0;   /* ошибка */
    }

  return res;
}

/* Закончить работу с разделяемой памятью
   Возвращает не 0 в случае ошибки. */
int shdata_close (SHDATA * data)
{
  /* Удаляем семафор */
  if (sem_close (sem_id))
    return 1; /* ошибка */
  /* Удаляем разделяемую память */
  if (shm_close (data))
    return 2; /* ошибка */
  return 0;
}

/* Обеспечить исключительную работу с разделяемой памятью
   текущему процессу.
   Возвращает не 0 в случае ошибки. */
int shdata_lock (SHDATA * data)
{
```

```
  /* Берем идентификатор семафора из разделяемой памяти*/
  int id = data->sem_id;
  return sem_lock (id);
}

/* Разрешить другим процессам работать с разделяемой
   памятью. Возвращает не 0 в случае ошибки. */
int shdata_unlock (SHDATA * data)
{
  /* Берем идентификатор семафора из разделяемой памяти*/
  int id = data->sem_id;
  return sem_unlock (id);
}
```

Для облегчения понимания код, работающий с блоком разделяемой памяти, разбит в файле `shdata.c` на ряд подпрограмм:

- `shm_open` — пытается открыть существующую разделяемую память, а в случае неудачи создает новую; информация о том, что был создан новый блок разделяемой памяти, возвращается в вызвавшую процедуру для проведения его инициализации;
- `shm_close` — закрывает разделяемую память и посылает запрос на ее удаление (который будет выполнен, когда все процессы вызовут эту функцию); после завершения `shm_close` любое обращение к раздсляемой памяти вызовет ошибку;
- `sem_open` — пытается открыть существующий семафор, а в случае неудачи создает новый; информация о том, что был создан новый семафор, используется для проведения его инициализации и возвращается в вызвавшую процедуру;
- `sem_close` — удаляет семафор; после этого активизируются все ожидавшие его процессы и любое последующее обращение к нему вызовет ошибку;
- `sem_lock` — «закрыть» семафор; после этого текущий процесс становится владельцем семафора;

- `sem_unlock` — «открыть» семафор; после этого другие процессы могут «закрывать» семафор;
- `shdata_open` — открывает разделяемую память и семафор, затем инициализирует разделяемую память (если она была создана, исключительный доступ к памяти текущему процессу обеспечивается с помощью семафора);
- `shdata_close` — удаляет разделяемую память и семафор;
- `shdata_lock` — обеспечить исключительный доступ к разделяемой памяти текущему процессу;
- `shdata_unlock` — разрешить другим процессам работать с разделяемой памятью.

Программа-клиент находится в файле `reader.c` (см. пояснения ниже):

```
#include "shdata.h"

/* Считать сообщение с терминала и записать в
   разделяемую память */
void reader (SHDATA * data)
{
  int ret;

  for (;;)
    {
      /* Обеспечить исключительную работу с разделяемой
         памятью текущему процессу */
      ret = shdata_lock (data);
      if (!ret)
        {
          if (!data->msglen)
            {
              /* Выход по Ctrl+D */
              if (!fgets (data->msgbuf, BUF_LEN, stdin))
                break;
              data->msglen = strlen (data->msgbuf) + 1;
```

```
            }

          /* Разрешить другим процессам работать с
             разделяемой памятью. */
          shdata_unlock (data);
        }
      /* Дать поработать другим */
      sleep (1);
    }

  data->done = 1;
  fprintf (stderr, "reader process %d exits\n", getpid());
  /* Разрешить другим процессам работать с разделяемой
     памятью. */
  shdata_unlock (data);
}

int main ()
{
  SHDATA * data = shdata_open ();

  if (data)
    {
      reader (data);
      shdata_close (data);
    }
  return 0;
}
```

Процедура `reader` в бесконечном цикле, прерываемом по ошибке ввода:

1. обеспечивает себе исключительный доступ к разделяемой памяти с помощью `shdata_lock`;
2. если предыдущее сообщение обработано сервером (т. е. длина сообщения `data->msglen` равна 0), то считывает строку со стандартного ввода с помощью `fgets`;

3. в случае успешного чтения устанавливает длину сообщения `data->msglen`, иначе: прерывает цикл, устанавливает переменную `data->done` в 1, разрешает другим процессам работать с разделяемой памятью с помощью `shdata_unlock` и завершает работу;
4. разрешает другим процессам работать с разделяемой памятью с помощью `shdata_unlock`;
5. для обеспечения переключения задач к процессу–серверу останавливает на 1 секунду выполнение текущего процесса с помощью `sleep` (иначе бы этот цикл работал вхолостую до окончания кванта времени, выделенного текущему процессу); это крайне плохое (исключительно модельное) решение, но у нас еще нет механизма, позволяющего ожидать, пока сервер не обработал запрос; см. «правильное» решение в разделе 8.4.5.

Программа–клиент находится в файле `writer.c` (см. пояснения ниже):

```
#include "shdata.h"

/* Взять сообщение из разделяемой памяти и вывести
   на экран */
void writer (SHDATA * data)
{
  int ret;

  for (;;)
    {
      /* Обеспечить исключительную работу с разделяемой
         памятью текущему процессу */
      ret = shdata_lock (data);
      /* Проверяем признак конца работы в любом случае */
      if (data->done)
        {
          /* Конец работы */
          fprintf (stderr,  "writer process %d exits\n",
```

```
                    getpid ());
          /* Разрешить другим процессам работать
             с разделяемой памятью. */
          if (!ret)
            shdata_unlock (data);
          return;
        }
      if (!ret)
        {
          if (data->msglen)
            {
              /* есть сообщение для вывода */
              printf ("-> %s\n", data->msgbuf);
              data->msglen = 0;
            }
          /* Разрешить другим процессам работать с
             разделяемой памятью. */
          shdata_unlock (data);
        }
      /* Дать поработать другим */
      sleep (1);
    }
}

int main ()
{
  SHDATA * data = shdata_open ();

  if (data)
    {
      writer (data);
      shdata_close (data);
    }
  return 0;
}
```

Процедура `writer` в бесконечном цикле, прерываемом в случае ненулевого значения `data->done`:

1. обеспечивает себе исключительный доступ к разделяемой памяти с помощью `shdata_lock`;
2. если `data->done` не равно 0, то разрешает другим процессам работать с разделяемой памятью с помощью `shdata_unlock` и завершает работу;
3. если есть сообщение для обработки сервером (т. е. длина сообщения `data->msglen` не равна 0), то выводит строку с помощью `printf` и устанавливает `data->msglen` в 0;
4. разрешает другим процессам работать с разделяемой памятью с помощью `shdata_unlock`;
5. для обеспечения переключения задач к процессам-клиентам останавливает на 1 секунду выполнение текущего процесса с помощью `sleep` (иначе бы этот цикл работал вхолостую до окончания кванта времени, выделенного текущему процессу).

Сделаем **полезное замечание об отладке** программ, использующих описанные в разделах 8.1, 8.2 и 8.4 функции. В случае аварийного завершения программы созданные ею объекты межпроцессного взаимодействия (блоки разделяемой памяти, семафоры и очереди сообщений) не удаляются. Следовательно, при повторном старте программы она получит объекты, оставшиеся от предыдущего запуска. При этом, например, семафор может оказаться занятым уже не существующим процессом и потому никогда не освободится. С большинством UNIX-систем поставляются утилиты, позволяющие посмотреть состояние таких объектов и удалить их:

- `ipcs` (InterProcess Communication Status) — выводит список блоков разделяемой памяти, семафоров и очередей сообщений с указанием их идентификатора, владельца, прав доступа и т. д.

- `ipcrm` (InterProcess Communication ReMove) — позволяет удалить:
  - `ipcrm shm` *id* — блок разделяемой памяти,
  - `ipcrm sem` *id* — семафор,
  - `ipcrm msg` *id* — очередь сообщений

  с идентификатором *id*, который можно получить с помощью команды `ipcs`.

8.3. События

Определение. Событие — это логический сигнал (оповещение), приходящий асинхронно по отношению к течению процесса. С каждым событием связаны булевская переменная E, принимающая два значения (0 — событие не пришло, и 1 — событие пришло), и очередь ожидания (процессов, задач, ожидающих прихода события). Над событиями определены следующие элементарные операции:

- $Signal(E)$ — послать событие, т. е. установить переменную E в 1, при этом первая в очереди ожидающая задача активизируется (в некоторых системах активизируется одна из задач, т. е. очередность не гарантируется);
- $Broadcast(E)$ — послать событие, т. е. установить переменную E в 1, при этом все задачи из очереди ожидания активизируются;
- $Wait(E)$ — ожидать события (если события нет, т. е. переменная E равна 0, то эта операция переводит задачу в состояние ожидания прихода события и добавляет ее в конец очереди ожидания этого события; как только событие придет, задача будет активизирована);
- $Reset(E)$ — очистить (удалить поступившее событие), т. е. установить переменную E в 0;
- $Test(E)$ — проверить (поступление) — получить значение переменной E.

Рассмотрим пример (рис. 8.2).

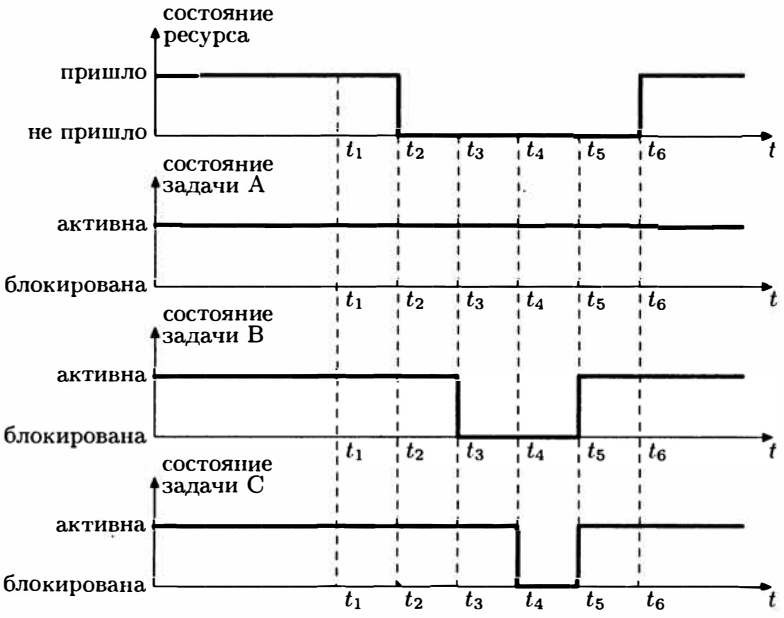


Рис. 8.2. Пример работы с событием

1. $B.Wait$ — задача B вызвала $Wait$; поскольку $E = 1$, то это не имеет никакого эффекта, B продолжает исполнение;
2. $Reset$ — была вызвана функция установки $E = 0$ (одной из задач A, B или C, или другой задачей);
3. $B.Wait$ — поскольку $E = 0$, то задача B заблокирована и помещена в очередь ожидания;
4. $C.Wait$ — поскольку $E = 0$, то задача C заблокирована и помещена в очередь ожидания;
5. $A.Broadcast$ — установить $E = 1$, активизировать все задачи из очереди ожидания, т. е. активизировать B и C.

С помощью событий легко организовать обмен между двумя задачами по типу клиент-сервер. Пусть есть два события E_1 и E_2. Задача T_1 (сервер) сразу после старта вызывает функцию $E_1.Wait()$. Тем самым она блокируется до получения события E_1. Задача T_2 (клиент) сразу после старта подготавливает дан-

ные в разделяемой с задачей T_1 памяти и вызывает функцию $E_1.Send()$. Это приводит к активизации задачи T_1. Задача T_2 продолжает работу и вызывает функцию $E_2.Wait()$, блокируясь до получения события E_2. Задача T_1 (сервер) по окончании обработки данных вызывает функции $E_2.Send()$, $E_1.Wait()$, активизируя задачу T_2 (клиента) и блокируя себя.

Ниже (см. раздел 10.3) мы рассмотрим реализацию событий для задач. Для процессов общепринятая реализация событий в UNIX-системах отсутствует.

8.4. Очереди сообщений (почтовые ящики)

Определение. Очереди сообщений (почтовые ящики) — это объект обмена данными между задачами, устроенный в виде очереди FIFO (рис. 8.3). С каждым почтовым ящиком связаны:

1. очередь сообщений (образующая содержимое почтового ящика),
2. очередь задач, ожидающих сообщений в почтовом ящике,
3. очередь задач, ожидающих освобождения места в почтовом ящике,
4. механизм взаимного исключения, обеспечивающий правильный доступ нескольких задач к одним и тем же сообщениям в почтовом ящике.

Количество задач, ожидающих сообщения в почтовом ящике, обычно не ограничено. Количество же сообщений в ящике обычно ограничено параметром, указанным при создании ящика. Это

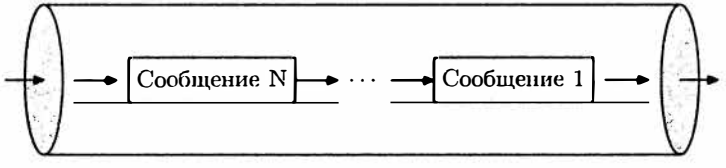


Рис. 8.3. Почтовый ящик

связано с тем, что размер каждого сообщения указывается при создании ящика и может быть значительным. Если ящик полон и задача пытается поместить в него новое сообщение, то она блокируется до тех пор, пока в ящике не появится свободное место. Над почтовыми ящиками определены следующие элементарные операции:

- положить сообщение в почтовый ящик, при этом задача, вызвавшая эту операцию, может быть блокирована и помещена в очередь задач, ожидающих освобождения места в ящике, если в ящике нет свободного места (активизация наступит сразу после взятия первого же сообщения из очереди сообщений); если очередь ожидающих сообщения задач непуста, то активизируются все задачи из этой очереди;
- попробовать положить сообщение в почтовый ящик — если в ящике есть свободное место, то эта операция эквивалентна положить, иначе вернуть признак отсутствия места;
- положить в начало — то же, что положить, только сообщение помещается в голову очереди;
- попробовать положить в начало — то же, что попробовать положить, только сообщение помещается в голову очереди;
- взять сообщение из почтового ящика, при этом задача, вызвавшая эту операцию, может быть блокирована и помещена в очередь задач, ожидающих сообщения, если в ящике нет сообщений; при поступлении сообщения она будет активизирована; при этом, если в очереди ожидающих сообщения задач находится более одной задачи, то при выполнении операции взять обеспечивается механизм взаимного исключения задач (т. е. пока выполняется эта операция одной задачей, все другие задачи, запросившие ту же операцию, заблокированы);
- попробовать взять сообщение из почтового ящика — если в ящике есть сообщения, то эта операция эквивалентна взять, иначе вернуть признак отсутствия сообщений;
- очистить ящик — удалить все сообщения из очереди.

Фактически описанный механизм представляет собой взаимодействие клиент–сервер, когда задачи–клиенты помещают запросы в почтовый ящик, а задачи–серверы их оттуда забирают.

8.4.1. Функция msgget

Функция `msgget`, описанная в файле `<sys/msg.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет создать (или открыть существующую) очередь сообщений с указанными правами доступа. Прототип:

```
int msgget (key_t key, int msgflg);
```

где входные параметры:

- `key` — числовой идентификатор очереди сообщений;
- `msgflg` — атрибуты, используемые при создании очереди сообщений (флаги и права доступа).

Функция возвращает идентификатор очереди сообщений (не путать с `key`) или -1 в случае ошибки.

8.4.2. Функция msgsnd

Функция `msgsnd`, описанная в файле `<sys/msg.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет положить сообщение в очередь. Прототип:

```
int msgsnd (int msqid, const void *msgp,
  size_t msgsz, int msgflg);
```

где входные параметры:

- `msqid` — идентификатор очереди сообщений;
- `msgp` — указатель на сообщение, которое должно иметь следующий тип (формат):

```
struct msgbuf
  {
    /* Тип (тег) сообщения */
    long int mtype;
```

```
    /* Данные сообщения */
    char mtext[1];
};
```

- `msgsz` — размер сообщения;
- `msgflg` — дополнительные флаги, определяющие поведение процесса в случае, если в очереди сообщений нет свободного места (можно указать: ожидать появления достаточного места или сигнализировать об ошибке).

Функция возвращает 0 в случае успеха или -1 в случае ошибки.

8.4.3. Функция msgrcv

Функция `msgrcv`, описанная в файле `<sys/msg.h>` (используемые типы описаны в заголовочном файле `<sys/ipc.h>`), позволяет взять сообщение с указанным типом (тегом) из очереди. Прототип:

```
int msgrcv (int msqid, void *msgp, size_t msgsz,
  long int msgtyp, int msgflg);
```

где входные параметры:

- `msqid` — идентификатор очереди сообщений;
- `msgp` — указатель на буфер для сообщения, которое должно иметь тип (формат) `struct msgbuf`;
- `msgsz` — размер буфера для сообщения;
- `msgtyp` — тип (тег) сообщения:
  - если `msgtyp` равно 0, то взять первое сообщение из очереди,
  - если `msgtyp` < 0, то взять первое сообщение из очереди с типом $<$ |`msgtyp`|,
  - если `msgtyp` > 0, то взять первое сообщение из очереди с типом $\neq$ |`msgtyp`|;
- `msgflg` — дополнительные флаги, определяющие поведение процесса в случае, если очередь сообщений пуста (можно указать: ожидать появления сообщений или сигнализировать об ошибке) или выбранное сообщение имеет длину боль-

ше msgsz (можно указать: обрезать сообщение или сигнализировать об ошибке).

Функция возвращает длину полученного сообщения в случае успеха или −1 в случае ошибки.

8.4.4. Функция msgctl

Функция msgctl, описанная в файле <sys/msg.h> (используемые типы описаны в заголовочном файле <sys/ipc.h>), позволяет управлять очередью сообщений (в частности, удалять ее). Прототип:

```
int msgctl (int msgid, int cmd,
  struct msqid_ds *buf);
```

где входные параметры:

- msqid — идентификатор очереди сообщений;
- cmd — задает операцию над очередью сообщений;
- buf — аргументы операции (также служит для возврата информации о результате операции).

Функция возвращает 0 в случае успеха или −1 в случае ошибки.

8.4.5. Пример использования очередей

Рассмотрим ту же задачу, что и в разделе 8.2.4. Приведенная в разделе 8.2.4 реализация нехороша тем, что для избежания бесполезного цикла опроса переменных используется функция sleep. С помощью очередей сообщений мы можем избежать этой проблемы и при этом сократить сами программы.

Общая часть — заголовочный файл msgdata.h, в котором описаны структура сообщения (MSGDATA) и функции работы с ним:

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

```
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define BUF_LEN 256

/* Первые два поля должны быть именно такими и в таком
   порядке для образования msgbuf */
typedef struct _msgdata_
{
  long int mtype;        /* тип сообщения */
  char msgbuf[BUF_LEN]; /* буфер для хранения сообщения*/
  int done;              /* признак окончания работы */
  int msglen;            /* длина сообщения */
} MSGDATA;

/* Длина данных сообщения (без первого поля) */
#define MSGDATA_LEN     (BUF_LEN + 2 * sizeof (int))

/* Идентификатор очереди сообщений */
#define MSG_KEY 1223

int msg_open (int key);
int msg_close (int id);
```

Общая часть (для клиента и сервера) находится в файле `msgdata.c` (см. пояснения ниже):

```
#include "msgdata.h"

/* Идентификатор разделяемой памяти */
#define SHM_KEY 1221
/* Идентификатор разделяемой памяти */
#define SEM_KEY (1221+1)

/* Начать работу с очередью сообщений с ключом key.
```

```
   Возвращает идентификатор очереди,
   -1 в случае ошибки.*/
int
msg_open (int key)
{
  int msg_id;   /* результат */

  /* Попытаться открыть очередь сообщений */
  if ((msg_id = msgget (key, 0)) == -1)
    {
      /* Не удалось, рассмотрим причину */
      if (errno == ENOENT)
        {
          /* Запрошенная очередь сообщений не существует.
             Создаем ее. */
          if ((msg_id = msgget (key, IPC_CREAT | 0664))
              == -1)
            {
              fprintf (stderr,
                       "Cannot create mesage queue\n");
              return 0;
            }
        }
      else
        {
          fprintf (stderr, "Cannot get mesage queue\n");
          return 0;
        }
    }

  return msg_id;
}

/* Закончить работу с очередью сообщений с
   идентификатором id.
```

```
   Возвращает не 0 в случае ошибки. */
int
msg_close (int id)
{
  /* Удаляем очередь сообщений. Удаление происходит сразу.
     Все ожидающие сообщений процессы активизируются,
     ожидание завершается с ошибкой EIDRM - Error
     IDentifier ReMoved. У них вся последующая работа с
     этой очередью будет давать ошибку, поэтому
     возвращаемое значение функции не проверяется. */
  msgctl (id, IPC_RMID, 0);
  return 0;
}
```

Функции, работающие с сообщением:

- `msg_open` — пытается открыть существующую очередь сообщений, а в случае неудачи создает новую;
- `msg_close` — удаляет очередь сообщений; после этого активизируются все процессы, ожидавшие сообщений в очереди, и любое последующее обращение к ней вызовет ошибку.

Программа-клиент находится в файле `reader.c` (см. пояснения ниже):

```
#include "msgdata.h"

/* Считать сообщение с терминала и записать в очередь */
void reader (int msg_id)
{
  MSGDATA data;

  for (;;)
    {
      /* Выход по Ctrl+D */
      if (!fgets (data.msgbuf, BUF_LEN, stdin))
        break;
      data.msglen = strlen (data.msgbuf) + 1;
```

```
      data.done = 0;
      data.mtype = 1;
      /* Посылаем сообщение в очередь */
      if (msgsnd (msg_id, &data, MSGDATA_LEN, 0) == -1)
        {
          perror ("send");
          fprintf (stderr, "Cannot send message\n");
          return;
        }
    }

  data.mtype = 1;
  data.done = 1;
  data.msglen = 0;
  /* Посылаем сообщение в очередь */
  if (msgsnd (msg_id, &data, MSGDATA_LEN, 0) == -1)
    {
      fprintf (stderr, "Cannot send message\n");
      return;
    }
  fprintf (stderr, "reader process %d exits\n", getpid())
}

int main ()
{
  /* Идентификатор очереди сообщений */
  int msg_id;

  if ((msg_id = msg_open (MSG_KEY)) == -1)
    return 1;

  reader (msg_id);
  msg_close (msg_id);
  return 0;
}
```

Процедура `reader` в бесконечном цикле, прерываемом по ошибке ввода:

1. считывает строку со стандартного ввода с помощью `fgets`;
2. в случае успешного чтения устанавливает реальную длину сообщения `data.msglen` и посылает его, иначе: прерывает цикл, устанавливает переменную `data.done` в 1, а реальную длину сообщения `data.msglen` — в 0, посылает это сообщение и завершает работу.

Программа-клиент находится в файле `writer.c` (см. пояснения ниже):

```
#include "msgdata.h"

/* Взять сообщение из очереди и вывести на экран */
void writer (int msg_id)
{
  MSGDATA data;

  for (;;)
    {
      /* Получаем сообщение из очереди */
      if (msgrcv (msg_id, &data, MSGDATA_LEN, 0, 0) == -1
        {
          /* Проверяем причину */
          if (errno == EIDRM)
            {
              /* Очередь была удалена другим процессом.*/
              break;     /* Заканчиваем работу */
            }

          fprintf (stderr, "Cannot recieve message\n");
          return;
        }

      /* Проверяем признак конца работы */
```

```
      if (data.done)
        {
          /* Конец работы */
          break;
        }
      if (data.msglen)
        {
          /* есть сообщение для вывода */
          printf ("-> %s\n", data.msgbuf);
          data.msglen = 0;
        }
    }
  fprintf (stderr,  "writer process %d exits\n",
           getpid ());
}

int main ()
{
  /* Идентификатор очереди сообщений */
  int msg_id;

  if ((msg_id = msg_open (MSG_KEY)) == -1)
    return 1;

  writer (msg_id);
  msg_close (msg_id);
  return 0;
}
```

Процедура `writer` в бесконечном цикле, прерываемом в случае ненулевого значения поля `done` в полученном сообщении:

1. получает сообщение;
2. если поле `done` в сообщении равно 0, то завершает работу;
3. если поле `msglen` в сообщении не равно 0 (т. е. есть строка для вывода), выводит строку из сообщения.

По поводу отладки программ, использующих очереди сообщений, см. замечание в разделе 8.2.4.

8.4.6. Функция pipe

Функция `pipe`, описанная в заголовочном файле `<unistd.h>`, предоставляет альтернативный по отношению к описанным выше функциям `msgget`, `msgsnd`, `msgrcv`, `msgctl` механизм создания очереди сообщений. Прототип:

```
int pipe (int filedesc[2]);
```

где `filedesc` — результат работы функции, представляющий собой массив из двух дескрипторов файлов. В случае успешного выполнения эта функция создает очередь сообщений, из которой можно читать данные обычными функциями файлового ввода (например, `read`), используя дескриптор `filedesc[0]`, и в которую можно записывать данные обычными функциями файлового вывода (например, `write`), используя дескриптор `filedesc[1]`, Функция возвращает -1 в случае ошибки.

Основные отличия функции `pipe` от интерфейса, предоставляемого функциями `msgget`, `msgsnd`, `msgrcv`, `msgctl`:

- Созданная `pipe` очередь не имеет идентификатора, позволяющего выделить конкретную очередь (ср. с `msgget`, где есть параметр `key`). Поэтому типичным использованием `pipe` является организация очереди между процессами–потомками одного родителя: `pipe` вызывается до функции `fork` и потому дескрипторы `filedesc` доступны потомку и родителю.
- Использование обычных функций файлового ввода-вывода (вроде `read` и `write`) вместо специализированных (`msgsnd` и `msgrcv`) приводит к тому, что отсутствует гарантия целостности получаемых и передаваемых сообщений (т. е. возможно одновременное чтение или запись от нескольких процессов с перемешиванием их данных). Поэтому необходимо применять тот или иной механизм взаимного исключения процессов. Поскольку по определению чтение из пустой очереди

и запись в заполненную очередь блокируют процесс, то это можно использовать в качестве возможного механизма синхронизации.

- Функция удаления созданной `pipe` очереди отсутствует (ср. с `msgctl`). Очередь удаляется как только будут закрыты (с помощью обычной файловой функции `close`) все дескрипторы файлов `filedesc` в каждом из процессов. При этом, если закрыты все дескрипторы `filedesc[1]` (для записи), то по окончании данных в очереди все операции чтения будут давать ошибку «конец файла». Если же закрыты все дескрипторы `filedesc[0]` (для чтения), то любая операция записи в очередь будет генерировать сигнал `SIGPIPE`, стандартный обработчик которого терминирует процесс.

Пример использования `pipe` см. в разделе 1.3. В этой программе очередь `from_root` служит для взаимного исключения процессов (передаваемые данные не используются), очередь `to_root` служит для передачи результата от потомка к родителю. Очереди создаются в основном (т. е. изначально запущенном пользователем) процессе, и только затем `p` раз вызывается функция `fork`, создавая `p` процессов-потомков. Следовательно, все процессы имеют одни и те же значения в `from_root` и `to_root`. Процессы-потомки сразу после старта закрывают свои копии дескрипторов для записи в очередь `from_root` (`from_root[1]`) и для чтения из очереди `to_root` (`to_root[0]`). Основной процесс после запуска потомков закрывает свои копии дескрипторов для записи в очередь `to_root` (`to_root[1]`) и для чтения из очереди `from_root` (`from_root[0]`). Тем самым формируются две очереди сообщений:

- `from_root`, в которую основной процесс может только записывать данные, а процессы-потомки — только их считывать,
- `to_root`, из которой основной процесс может только считывать данные, а процессы-потомки — только их записывать.

Сразу после запуска процесс-потомок вычисляет свою часть интеграла с помощью функции `integrate` и переходит в режим ожидания появления сообщения (из одного байта) в очереди `from_root`. Один байт является минимальной величиной, неделимой между процессами, поэтому при отправке основным процессом такого сообщения будет активизирован ровно один из его потомков. Дождавшись сообщения, процесс-потомок записывает результат в очередь `to_root` и заканчивает свою работу. Основной процесс после запуска потомков `p` раз последовательно записывает один байт в очередь `from_root` (тем самым активизируя один из процессов-потомков) и переходит в режим ожидания появления сообщения в очереди `to_root`. Получив сообщение — число типа `double`, основной процесс прибавляет его к результату `total`, формируя окончательный ответ.

8.5. Пример многопроцессной программы, вычисляющей произведение матрицы на вектор

Рассмотрим задачу вычисления произведения матрицы на вектор. Для повышения скорости работы на многопроцессорной вычислительной установке с общей памятью создадим несколько процессов (по числу процессоров) и разделим работу между ними, разместив все данные в разделяемой памяти. Основным элементом синхронизации задач будет являться функция `synchronize`, которая дает возможность процессу ожидать, пока эту же функцию вызовут все другие процессы. Файлы проекта:

- `synchronize.c`, `synchronize.h` — исходный текст и соответствующий заголовочный файл для функций синхронизации и взаимного исключения процессов;
- `get_time.c`, `get_time.h` — исходный текст и соответствующий заголовочный файл для функций работы со временем;
- `main.c` — запуск процессов и вывод результатов;
- `matrices.c`, `matrices.h` — исходный текст и соответствующий заголовочный файл для функций, работающих с матрицами;

- `Makefile` — для сборки проекта.

Заголовочный файл `synchronize.h`:

```
int init_synchronize_lock (int total_processes);
int destroy_synchronize_lock (void);
int lock_data (void);
int unlock_data (void);
void synchronize (int process_num, int total_processes);
```

В файле `synchronize.c` находятся функции:

- `sem_open` — создает новый набор из указанного количества семафоров и устанавливает каждый из них в указанное начальное значение; для создания уникального (т. е. отличного от уже имеющихся в каком-либо процессе) набора в качестве ключа для функции `semget` использовано значение `IPC_PRIVATE`, для инициализации всех созданных семафоров в функции `semctl` использовано значение `SETALL`;
- `sem_close` — удаляет указанный набор семафоров;
- `init_synchronize_lock` — создает все используемые объекты синхронизации и взаимного исключения процессов:
  - 1 семафор с начальным значением 1 для взаимного исключения процессов при доступе к разделяемой памяти;
  - 2 набора по `total_processes` семафоров с начальным значением 0 для синхронизации процессов;
  - массив `lock_all`, используемый для операций с последними двумя наборами;
- `destroy_synchronize_lock` — удаляет все созданные в предыдущей функции объекты;
- `lock_data` — обеспечить исключительную работу с разделяемой памятью текущему процессу;
- `unlock_data` — разрешить другим процессам работать с разделяемой памятью;
- `synchronize_internal` — основная подпрограмма функции синхронизации; блокирует вызвавший ее процесс до тех пор, пока ее не вызовут все `total_processes` процессов; схема работы:

1. добавить `total_processes` к значению семафора с номером `process_num` из указанного набора, состоящего из `total_processes` семафоров; эта операция не приводит к блокировке процесса;
2. вычесть 1 из значения каждого из `total_processes` семафоров в указанном наборе; эта операция приводит к блокировке процесса до тех пор, пока значения **всех** семафоров не станут положительными;

другими словами, каждый из `total_processes` процессов прибавляет `total_processes` к одному семафору, и вычитает 1 из всех, поэтому, например, перед приходом в эту функцию последнего процесса, имеющего номер `k`, `k`-й семафор в наборе имеет значение `1-total_processes` (и потому блокирует все остальные `total_processes-1` процессов), а остальные `total_processes-1` семафоров имеют значение 1; последний пришедший процесс описанной выше операцией 1 разрешает работать всем процессам, а затем операцией 2 приводит все семафоры в начальное значение 0;

- `synchronize` — основная функция синхронизации; блокирует вызвавший ее процесс до тех пор, пока ее не вызовут все `total_processes` процессов; дважды вызывает предыдущую функцию для двух разных наборов по `total_processes` семафоров для того, чтобы дождаться, пока в эту функцию все процессы **войдут** и **выйдут**, не допуская ситуации, когда один из процессов, выйдя из `synchronize`, войдет в нее раньше, чем остальные вышли при предыдущей синхронизации, и нарушит тем самым ее работу.

Файл `synchronize.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/sem.h>

#include "synchronize.h"

/* Идентификатор семафора взаимного исключения */
static int lock_data_sem_id;
/* Идентификаторы наборов семафоров, используемых для
   синхронизации. */
static int processes_in_sem_id;
static int processes_out_sem_id;
/* Структура данных для операций с набором семафоров */
struct sembuf * lock_all;

/* "Открыть" новый набор семафоров длины n и установить
   начальное значение каждого семафора val. Возвращает
   идентификатор набора, -1 в случае ошибки. */
static int
sem_open (int n, int val)
{
  int id;       /* результат */
  /* Структура данных для управления семафором */
#ifdef _SEM_SEMUN_UNDEFINED
  union semun
  {
    /* Значение для команды SETVAL */
    int val;
    /* Буфер для команд IPC_STAT и IPC_SET */
    struct semid_ds *buf;
    /* Массив для команд GETALL и SETALL */
    unsigned short int *array;
    /* Буфер для команды IPC_INFO */
    struct seminfo *__buf;
  };
#endif
  union semun s_un;
```

```
  int i;

  /* Создаем новый набор семафоров */
  if ((id = semget (IPC_PRIVATE, n,
                    IPC_CREAT | 0664)) == -1)
    {
      fprintf (stderr, "Cannot create semaphore\n");
      return -1;
    }

  /* Выделяем память под массив начальных значений */
  if (!(s_un.array = (unsigned short int*)
        malloc (n * sizeof (unsigned short int))))
    {
      /* Мало памяти */
      fprintf (stderr, "Not enough memory\n");
      return -1;
    }

  /*  Начальное значение набора семафоров */
  for (i = 0; i < n; i++)
    s_un.array[i] = val;

  /*  Установить начальное значение каждого семафора */
  if (semctl (id, 0, SETALL, s_un) == -1)
    {
      fprintf (stderr, "Cannot init semaphore\n");
      /* С семафором работать нельзя, заканчиваем */
      free (s_un.array);
      return -1;
    }

  free (s_un.array);
  return id;
}
```

```
/* "Закрыть" семафор с идентификатором id.
   Возвращает не 0 в случае ошибки. */
static int
sem_close (int id)
{
  /* Удаляем семафор. */
  if (semctl (id, 0, IPC_RMID, 0) == -1)
    {
      fprintf (stderr, "Cannot delete semaphore\n");
      return -1;
    }
  return 0;
}

/* Инициализировать все объекты синхронизации и
   взаимного исключения. Возвращает -1 в случае ошибки.*/
int
init_synchronize_lock (int total_processes)
{
  int i;

  /* Создаем 1 семафор с начальным значением 1 */
  lock_data_sem_id = sem_open (1, 1);
  /* Создаем total_processes семафоров с начальным
     значением 0 */
  processes_in_sem_id = sem_open (total_processes, 0);
  /* Создаем total_processes семафоров с начальным
     значением 0 */
  processes_out_sem_id = sem_open (total_processes, 0);
  /* Проверяем успешность создания */
  if (lock_data_sem_id == -1 || processes_in_sem_id == -1
      || processes_out_sem_id == -1)
    return -1; /* семафоры не созданы */
  /* Выделяем память под массив */
```

```
  if (!(lock_all = (struct sembuf *)
      malloc (total_processes * sizeof (struct sembuf))))
    return -2; /* нет памяти */
  /* Инициализируем массив */
  for (i = 0; i < total_processes; i++)
    {
      lock_all[i].sem_num = i;
      lock_all[i].sem_op = -1;
      lock_all[i].sem_flg = 0;
    }

  return 0;
}

/* Закончить работу со всеми объектами синхронизации и
   взаимного исключения. Возвращает -1 в случае ошибки.*/
int
destroy_synchronize_lock ()
{
  if (sem_close (lock_data_sem_id) == -1
      || sem_close (processes_in_sem_id) == -1
      || sem_close (processes_out_sem_id) == -1)
    return -1; /* ошибка */
  free (lock_all);
  return 0;
}

/* Обеспечить исключительную работу с разделяемой памятью
   текущему процессу. Возвращает не 0 в случае ошибки. */
int
lock_data ()
{
  /* Структура данных для операций с семафором */
  struct sembuf s_buf = { 0, -1, 0};
```

```
  if (semop (lock_data_sem_id, &s_buf, 1) == -1)
    {
      fprintf (stderr, "Cannot lock semaphore\n");
      return -1; /* ошибка */
    }
  return 0;
}

/* Разрешить другим процессам работать с разделяемой
   памятью. Возвращает не 0 в случае ошибки. */
int
unlock_data ()
{
  /* Структура данных для операций с семафором */
  struct sembuf s_buf = { 0, 1, 0};

  if (semop (lock_data_sem_id, &s_buf, 1) == -1)
    {
      fprintf (stderr, "Cannot unlock semaphore\n");
      return -1; /* ошибка */
    }
  return 0;
}

/* Подпрограмма для synchronize */
static int
synchronize_internal (int id, int process_num,
                      int total_processes)
{
  /* Добавить total_processes единиц к семафору с номером
     process_num из набора с идентификатором id. */
  /* Структура данных для операций с семафором */
  struct sembuf s_buf = {process_num, total_processes, 0};

  if (semop (id, &s_buf, 1) == -1)
```

```
    return -1; /* ошибка */

  /* Вычесть 1 из каждого из total_processes семафоров
     из набора с идентификатором id. */
  if (semop (id, lock_all, total_processes) == -1)
    return -2; /* ошибка */

  return 0;
}

/* Дождаться в текущем процессе с номером process_num
  остальных процессов (из общего числа total_processes)*/
void
synchronize (int process_num, int total_processes)
{
  /* Дождаться, пока все процессы войдут в эту функцию */
  synchronize_internal (processes_in_sem_id, process_num,
                        total_processes);
  /* Дождаться, пока все процессы выйдут из этой функции*/
  synchronize_internal (processes_out_sem_id, process_num,
                        total_processes);
}
```

Заголовочный файл get_time.h:

```
long int get_time ();
long int get_full_time ();
```

Файл get_time.c:

```
#include <sys/time.h>
#include <sys/resource.h>
#include "get_time.h"

/* Вернуть процессорное время, затраченное на текущий
   процесс, в сотых долях секунды. Берется время только
   самого процесса, время работы системных вызовов не
   прибавляется. */
```

```
long int get_time ()
{
  struct rusage buf;

  getrusage (RUSAGE_SELF, &buf);
           /* преобразуем время в секундах
              в сотые доли секунды */
  return   buf.ru_utime.tv_sec * 100
           /* преобразуем время в микросекундах
              в сотые доли секунды */
         + buf.ru_utime.tv_usec / 10000;
}

/* Возвращает астрономическое (!) время
   в сотых долях секунды */
long int get_full_time ()
{
  struct timeval buf;

  gettimeofday (&buf, 0);
           /* преобразуем время в секундах
              в сотые доли секунды */
  return   buf.tv_sec * 100
           /* преобразуем время в микросекундах
              в сотые доли секунды */
         + buf.tv_usec / 10000;
}
```

Результат, выдаваемый `get_full_time`, имеет смысл лишь в том случае, если в системе нет других выполняющихся процессов, кроме запущенной этой программой группы.

В файле `main.c` находится главная функция `main`, которая:

- создает блок разделяемой памяти, достаточный для размещения:

  1. 1-й матрицы $n \times n$,

2. 2-х векторов длины n,
3. 1-го целого числа, в котором будет суммироваться время работы всех процессов;

для создания уникального (т. е. отличного от уже имеющихся в каком-либо процессе) блока разделяемой памяти в качестве ключа для функции `shmget` использовано значение `IPC_PRIVATE`;

- подсоединяет созданный блок разделяемой памяти к адресному пространству процесса;
- с помощью `init_synchronize_lock` создает все используемые объекты синхронизации и взаимного исключения процессов;
- размещает матрицу, 2 вектора, счетчик времени в разделяемой памяти и инициализирует их;
- запускает `total_processes-1` копий текущего процесса с помощью `fork`, при этом созданный блок разделяемой памяти и все семафоры наследуются порожденными процессами;
- каждый из `total_processes` процессов вызывает функцию `matrix_mult_vector_process`, которая:
  - вычисляет компоненты ответа с индексами в диапазоне

    ```
    n * process_num / total_processes, ...,
    n * (process_num + 1)  / total_processes - 1
    ```

    с помощью функции `matrix_mult_vector`;
  - вычисляет суммарное процессорное время, затраченное на все запущенные процессы; для взаимного исключения при доступе процессов к одной и той же ячейке разделяемой памяти с адресом `p_total_time` используются функции `lock_data`, `unlock_data` (см. выше);
  - для целей отладки и более точного замера времени работы функция вычисления произведения матрицы на вектор вызывается в цикле `N_TESTS` раз.
- каждый из `total_processes` процессов вызывает функцию `synchronize` для ожидания завершения всех процессов;

- total_processes-1 порожденных процессов завершаются; главный процесс (с номером 0) выводит результаты работы и удаляет все объекты синхронизации и разделяемую память.

Файл main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "matrices.h"
#include "get_time.h"
#include "synchronize.h"

/* Количество тестов (для отладки) */
#define N_TESTS 10

/* Умножение матрицы на вектор для одного процесса */
void
matrix_mult_vector_process (
  double * matrix,      /* матрица */
  double * vector,      /* вектор */
  double * result,      /* результат */
  int n,                /* размерность */
  long * p_total_time,  /* суммарное время */
  int process_num,      /* номер процесса */
  int total_processes)  /* всего процессов */
{
  long int t;
  int i;

  printf ("Process %d started\n", process_num);
```

```
  t = get_time ();       /* время начала работы */
  for (i = 0; i < N_TESTS; i++)
  {
    matrix_mult_vector (matrix, vector, result, n,
                        process_num, total_processes);
    printf ("Process %d mult %d times\n",
            process_num, i);
  }
  t = get_time () - t;  /* время конца работы */

  /* Суммируем времена работы */
  /* Обеспечить исключительный доступ к памяти */
  lock_data ();
  *p_total_time += t;
  /* Разрешить другим работать с разделяемой памятью */
  unlock_data ();
  printf ("Process %d finished, time = %ld\n",
          process_num, t);
}


int main ()
{
  int total_processes;  /* число создаваемых процессов *
  int process_num;      /* номер процесса */
  /* астрономическое время работы всего процесса */
  long int t_full;

  void *shmem;  /* указатель на разделяемую память   */
  int shm_id;   /* идентификатор разделяемой памяти  */
  int n;                /* размер матрицы и векторов */
  double *matrix;       /* матрица */
  double *vector;       /* вектор */
  double *result;       /* результирующий вектор */
  long *p_total_time;   /* общее время всех процессов*/
```

```
int l;
pid_t pid;

printf ("Input processes number: ");
scanf ("%d", &total_processes);

printf ("matrix size = ");
scanf ("%d", &n);

/* Общий размер требуемой памяти:
   под массивы:
     1 матрица n x n типа double
     2 вектора длины n типа double
   для счетчика времени:
     1 число типа long int */
l = (n * n + 2 * n) * sizeof (double) + sizeof (long);

/* Создаем новый блок разделяемой памяти */
if ((shm_id = shmget (IPC_PRIVATE, l,
                      IPC_CREAT | 0664)) == -1)
  {
    fprintf (stderr, "Cannot create shared memory\n");
    return 1; /* завершаем работу */
  }

/* Подсоединяем разделяемую память к адресному
   пространству процесса */
if ((shmem = (void *)shmat (shm_id, 0, 0)) == (void*)-1)
  {
    fprintf (stderr, "Cannot attach shared memory\n");
    return 3; /* завершаем работу */
  }

/* Устанавливаем указатели на данные
   в разделяемой памяти */
```

```
matrix = (double*) shmem;
vector = matrix + n * n;
result = vector + n;
p_total_time = (long int*) (result + n);
*p_total_time = 0; /* начальное значение */

/* Инициализируем все объекты взаимодействия */
if (init_synchronize_lock (total_processes))
  return 4; /* завершаем работу в случае ошибки */

/* Инициализация массивов */
init_matrix (matrix, n);
init_vector (vector, n);
printf ("Matrix:\n");
print_matrix (matrix, n);
printf ("Vector:\n");
print_vector (vector, n);

printf ("Allocated %d bytes (%dKb or %dMb) of memory\n"
        l, l >> 10, l >> 20);

/* Засекаем астрономическое время начала работы */
t_full = get_full_time ();
/* Запускаем total_processes - 1 процессов,
   процесс с номером 0 - текущий */
for (process_num = total_processes - 1;
     process_num > 0; process_num--)
  {
    /* Клонировать себя */
    pid = fork ();
    if (pid == -1)
      {
        fprintf (stderr, "Cannot fork!\n");
        return 5; /* завершаем работу */
      }
```

```
    if (pid == 0)
      {
        /* Выполняем работу в порожденном процессе */
        matrix_mult_vector_process (matrix, vector,
          result, n, p_total_time, process_num,
          total_processes);
        /* Дождаться остальных процессов */
        synchronize (process_num, total_processes);
        return 0; /* завершить процесс */
      }
  }

/* Здесь работает главный процесс с номером 0. */

/* Выполняем работу в процессе process_num = 0 */
matrix_mult_vector_process (matrix, vector, result, n,
  p_total_time, process_num, total_processes);
/* Дождаться остальных процессов */
synchronize (process_num, total_processes);

t_full = get_full_time () - t_full;
if (t_full == 0)
  t_full = 1; /* очень быстрый компьютер... */

/* Здесь можно работать с результатом */
print_vector (result, n);

printf ("Total full time = %ld, \
total processes time = %ld (%ld%%), per process = %ld\n",
        t_full, *p_total_time,
        (*p_total_time * 100) / t_full,
        *p_total_time / total_processes);

/* Ликвидируем созданные объекты синхронизации */
destroy_synchronize_lock ();
```

```
  /* Отсоединяем разделяемую память от адресного
     пространства процесса */
  if (shmdt (shmem) == -1)
    {
      fprintf (stderr, "Cannot detach shared memory\n");
      return 10;
    }

  /* Удаляем разделяемую память. */
  if (shmctl (shm_id, IPC_RMID, 0) == -1)
    {
      fprintf (stderr, "Cannot delete shared memory\n");
      return 11;
    }

  return 0;
}
```

Заголовочный файл matrices.h:

```
void init_matrix (double * matrix, int n);
void init_vector (double * vector, int n);
void print_matrix (double * matrix, int n);
void print_vector (double * vector, int n);
void matrix_mult_vector (double *a, double *b, double *c
                         int n, int process_num,
                         int total_processes);
```

Файл matrices.c:

```
#include <stdio.h>
#include "matrices.h"
#include "synchronize.h"

/* Инициализация матрицы */
void init_matrix (double * matrix, int n)
{
```

```
  int i, j;
  double *a = matrix;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      *(a++) = (i > j) ? i : j;
}

/* Инициализация вектора */
void init_vector (double * vector, int n)
{
  int i;
  double *b = vector;

  for (i = 0; i < n; i++)
    *(b++) = 1.;
}

#define N_MAX   5

/* Вывод матрицы */
void print_matrix (double * matrix, int n)
{
  int i, j;
  int m = (n > N_MAX ? N_MAX : n);

  for (i = 0; i < m; i++)
    {
      for (j = 0; j < m; j++)
        printf (" %12.6lf", matrix[i * n + j]);
      printf ("\n");
    }
}

/* Вывод вектора */
```

```
void print_vector (double * vector, int n)
{
  int i;
  int m = (n > N_MAX ? N_MAX : n);

  for (i = 0; i < m; i++)
      printf (" %12.6lf", vector[i]);
  printf ("\n");
}

/* Умножить матрицу a на вектор b, c = ab для процесса с
   номером process_num из общего количества
   total_processes. */
void matrix_mult_vector (double *a, double *b, double *c,
                         int n, int process_num,
                         int total_processes)
{
  int i, j;
  double *p, s;
  int first_row, last_row;

  /* Первая участвующая строка матрицы */
  first_row = n * process_num;
  first_row /= total_processes;
  /* Последняя участвующая строка матрицы */
  last_row = n * (process_num + 1);
  last_row = last_row / total_processes - 1;

  for (i = first_row, p = a + i * n; i <= last_row; i++)
    {
      for (s = 0., j = 0; j < n; j++)
        s += *(p++) * b[j];
      c[i] = s;
    }
  ::ynchronize (process_num, total_processes);
```

```
}
```

Файл Makefile:

```
NAME    = process_mult

DEBUG   = -g
CC      = gcc -c
LD      = gcc
CFLAGS  = $(DEBUG) -W -Wall
LIBS    = -lm
LDFLAGS = $(DEBUG)

OBJS = main.o matrices.o synchronize.o get_time.o

all : $(NAME)

$(NAME) : $(OBJS)
        $(LD) $(LDFLAGS) $^ $(LIBS) -o $@

.c.o:
        $(CC) $(CFLAGS) $< -o $@

clean:
        rm -f $(OBJS) $(NAME)

main.o       : main.c matrices.h get_time.h synchronize.h
matrices.o   : matrices.c matrices.h synchronize.h
synchronize.o: synchronize.c synchronize.h
get_time.o   : get_time.c get_time.h
```

Сделаем **замечание об отладке** приведенной выше программы. В случае ее аварийного завершения созданные объекты межпроцессного взаимодействия (блок разделяемой памяти и наборы семафоров) не удаляются. При повторном запуске программа будет требовать создания **новых** объектов, что может оказаться невозможным из-за исчерпания объема доступ-

ной разделяемой памяти или количества семафоров. Поэтому оставшиеся после ошибки в программе объекты IPC необходимо удалить внешней процедурой, см. описание утилит `ipcs` и `ipcrm` в разделе 8.2.4 (с. 119).

Заметим также, что максимальный объем разделяемой памяти, выделяемой процессу, является важным параметром любой операционной системы, и обычно существенно меньше объема доступной процессу виртуальной памяти. Часто этот параметр нельзя изменить динамически (во время работы системы), а только путем перекомпиляции ядра. Поэтому для описанной выше задачи, требующей при больших n значительных объемов памяти, предпочтительнее использовать механизм задач (threads), где объемы разделяемой и виртуальной памяти совпадают.

9

Управление задачами (threads)

В этой главе мы рассмотрим основные функции, позволяющие запускать новые задачи и управлять ими.

9.1. Функция pthread_create

Функция `pthread_create`, описанная в заголовочном файле `<pthread.h>`, позволяет создать новую задачу и начать ее выполнение с указанной функции. Прототип:

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

где

- `thread` — возвращаемое значение — идентификатор созданной задачи;
- `attr` — указатель на структуру с атрибутами создаваемой задачи (может быть 0 для использования значений по умолчанию);
- `start_routine` — указатель на функцию, с которой надо начать выполнение задачи;
- `arg` — аргумент этой функции.

Функция возвращает 0 в случае успеха.

9.2. Функция pthread_join

Функция `pthread_join`, описанная в файле `<pthread.h>`, позволяет текущей задаче ожидать окончания указанной задачи (этого же процесса). Прототип:

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

где

- `thread` — идентификатор задачи, завершения которой надо ожидать,
- `thread_return` — указатель на переменную, где будет сохранено возвращаемое значение задачи (может быть 0, если это не требуется).

Функция возвращает 0 в случае успеха.

Любую задачу установкой специального атрибута при создании или с помощью специальной функции (`pthread_detach`) можно перевести в такое состояние, в котором ее нельзя будет ожидать посредством `pthread_join`.

9.3. Функция sched_yield

Функция `sched_yield`, описанная в файле `<sched.h>`, позволяет поставить текущую задачу (процесс) в конец очереди готовых задач (процессов) с тем же приоритетом. Прототип:

```
int sched_yield (void);
```

Функция возвращает 0 в случае успеха.

10

Синхронизация и взаимодействие задач

Вся память у работающих задач является разделяемой между ними, поэтому для совместного использования любой глобальной переменной необходимо использовать тот или иной объект синхронизации. Для повышения эффективности работы программ и удобства программирования для задач введены специальные средства синхронизации.

10.1. Объекты синхронизации типа mutex

Объекты синхронизации типа mutex фактически представляют собой некоторое развитие булевских семафоров в плане повышения безопасности и эффективности работы программы.

Типичный цикл работы с разделяемым ресурсом следующий: взять семафор, работать с ресурсом, вернуть семафор. Однако, если в результате ошибки в программе она вначале вызовет функцию вернуть семафор (не взяв его!), а затем выполнит приведенный выше цикл работы с разделяемым ресурсом, то функция взять семафор не блокирует задачу, если ресурс занят.

Другой проблемой при работе с семафорами является необходимость переключения задач при каждом вызове функций, работающих с семафором. Дело в том, что сам счетчик семафора s (фактически являющийся разделяемой между процессами

памятью) находится в области данных операционной системы, и любая функция, работающая с семафором, является системным вызовом. Это особенно плохо при синхронизации между задачами, поскольку в этом случае сам обмен данными не требует переключения задач (так как вся память у задач общая).

Для борьбы с этими явлениями вводится объект mutex, который фактически состоит из пары: булевского семафора и идентификатора задачи — текущего владельца семафора (т. е. той задачи, которая успешно вызвала функцию взять и стала владельцем разделяемого ресурса). При этом сама эта пара хранится в разделяемой между задачами памяти (в случае threads — в любом месте их общей памяти). Для доступа к объекту mutex m определены три примитивные операции:

- $Lock(m)$ — блокировать mutex m, если m уже заблокирован другой задачей, то эта операция переводит задачу в состояние ожидания разблокирования m;
- $Unlock(m)$ — разблокировать mutex m, (если m ожидается другой задачей, то она может быть активизирована, удалена из очереди ожидания и может вытеснить текущую задачу, например, если ее приоритет выше); если вызвавшая эту операцию задача не является владельцем m, то операция не имеет никакого эффекта;
- $TryLock(m)$ — попробовать блокировать mutex m, если m не блокирован, то эта операция эквивалентна $Lock(m)$, иначе возвращается признак неудачи.

Эти операции **неделимы**, т. е. переключение задач во время их исполнения запрещено.

Объекты mutex бывают:

- **локальными** — доступны для синхронизации между задачами (threads) одного процесса; размещаются в любом месте их общей памяти;
- **глобальными** — доступны для синхронизации между задачами (threads) разных процессов; размещаются в разделяемой между процессами памяти.

Отметим, что глобальные mutex появились значительно позже локальных и предоставляются не всеми операционными системами. Причиной их появления явилось удобство локальных mutex для синхронизации задач, и глобальные mutex являются обобщением последних на процессы.

Некоторые операционные системы предоставляют объекты mutex со специальными свойствами, полезными в ряде случаев. Рассмотрим следующую ситуацию. Функции `f1` и `f2` работают с разделяемыми ресурсами и используют mutex `m` для синхронизации между задачами:

```
f1()
{
  Lock(m);
  ...
  Unlock(m);
}

f2()
{
  Lock(m);
  ...
  Unlock(m);
}
```

В результате развития программы нам потребовалось вызвать функцию `f1` из `f2`:

```
f2()
{
  Lock(m);
  <операторы 1>
  f1();
  <операторы 2>
  Unlock(m);
}
```

Однако это приводит к ситуации **deadlock**: задача переходит в вечный цикл ожидания освобождения mutex `m` в функции `f1`, поскольку она уже является владельцем этого mutex. Поэтому некоторые операционные системы предоставляют объекты mutex дополнительных типов:

- **error check** — вызов *Lock* владельцем mutex, а также *Unlock* не владельцем не производят никакого действия; отметим, что mutex этого типа решит проблему, описанную выше, но появляется новая: `<операторы 2>` (может быть, работающие с разделяемым ресурсом) выполняются, когда задача уже не является владельцем mutex;
- **recurcive** — вызов *Lock* владельцем mutex увеличивает счетчик таких вызовов, вызов *Unlock* владельцем — уменьшает счетчик; реально разблокирование mutex происходит при значении счетчика, равном 0.

10.1.1. Функция pthread_mutex_init

Функция `pthread_mutex_init`, описанная в заголовочном файле `<pthread.h>`, позволяет инициализировать mutex. Прототип:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
  pthread_mutexattr_t *attr);
```

где

- `mutex` — указатель на mutex,
- `attr` — указатель на структуру с атрибутами mutex (может быть 0 для использования значений по умолчанию).

Функция возвращает 0 в случае успеха.

Для mutex (объекта типа `pthread_mutex_t`) вместо функции `pthread_mutex_init` можно использовать статическую инициализацию при объявлении:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

где макрос PTHREAD_MUTEX_INITIALIZER определен в заголовочном файле <pthread.h>. Последняя запись эквивалентна последовательности:

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, 0);
```

но, в отличие от нее, может быть использована для инициализации глобальных объектов.

10.1.2. Функция pthread_mutex_lock

Функция pthread_mutex_lock, описанная в заголовочном файле <pthread.h>, позволяет блокировать mutex; если он уже заблокирован другой задачей, то эта функция переводит задачу в состояние ожидания разблокирования mutex. Прототип:

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

где mutex — указатель на mutex. Функция возвращает 0 в случае успеха.

10.1.3. Функция pthread_mutex_trylock

Функция pthread_mutex_trylock, описанная в заголовочном файле <pthread.h>, позволяет блокировать mutex; если он уже заблокирован другой задачей, то эта функция возвращает ошибку, не меняя состояния mutex. Прототип:

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

где mutex — указатель на mutex. Функция возвращает 0 в случае успеха.

10.1.4. Функция pthread_mutex_unlock

Функция pthread_mutex_unlock, описанная в заголовочном файле <pthread.h>, позволяет разблокировать mutex. Прототип:

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

где mutex — указатель на mutex. Функция возвращает 0 в случае успеха.

10.1.5. Функция pthread_mutex_destroy

Функция pthread_mutex_destroy, описанная в заголовочном файле <pthread.h>, позволяет удалить mutex, который должен быть в этот момент разблокированным. Прототип:

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

где mutex — указатель на mutex. Функция возвращает 0 в случае успеха.

10.1.6. Пример использования mutex

Рассмотрим ту же задачу, что и в разделе 8.2.4, только вместо процессов будем использовать задачи. Приводимая ниже программа считывает строку со стандартного ввода и выводит ее на стандартный вывод. Для чтения строки и для ее вывода создаются две задачи (thread), кодом для которых являются функции reader и writer соответственно. Поскольку задачи работают в том же адресном пространстве, что и создавший их процесс, то они автоматически разделяют все переменные, включая буфер сообщений msgbuf. Для организации взаимного исключения при доступе к критическим разделяемым ресурсам (переменным done, msglen, msgbuf) используется объект синхронизации mutex mutex.

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

#define BUF_LEN 256

/* Объект синхронизации типа mutex */
pthread_mutex_t mutex;
```

```
/* Признак окончания работы */
int done;
/* Длина сообщения */
int msglen;
/* Буфер для сообщения */
char msgbuf[BUF_LEN];

/* Считать сообщение и поместить его в буфер.
   Функция работает как независимая задача, вызываемая
   операционной системой, поэтому прототип фиксирован.
   Аргумент argp не используется. */
void * reader (void *argp)
{
  for (;;)
    {
      /* "захватить" mutex */
      pthread_mutex_lock (&mutex);
      if (!msglen)
        {
          /* Считать сообщение. Выход по Ctrl+D */
          if (!fgets (msgbuf, BUF_LEN, stdin))
            break;
          msglen = strlen (msgbuf) + 1;
        }
      /* "освободить" mutex  */
      pthread_mutex_unlock (&mutex);
      /* Поместить задачу в конец очереди готовых задач
         с тем же приоритетом */
      sched_yield ();
    }
  /* Напечатать идентификатор текущей задачи */
  printf ("Thread %x exits\n", (int)pthread_self ());
  done = 1;
  /* "освободить" mutex */
  pthread_mutex_unlock (&mutex);
```

```
  /*завершить задачу */
  return 0;
}

/*Ждать сообщения, по его поступлении вывести его на
   экран. Функция работает как независимая задача,
   вызываемая операционной системой, поэтому прототип
   фиксирован. Аргумент argp не используется. */
void * writer (void * argp)
{
  for (;;)
    {
      /*"захватить" mutex */
      pthread_mutex_lock (&mutex);
      if (done)
        {
          /*Напечатать идентификатор текущей задачи */
          printf ("Thread %x exits\n",
                  (int)pthread_self ());
          /*"освободить" mutex */
          pthread_mutex_unlock (&mutex);
          /*завершить задачу */
          return 0;
        }
      if (msglen)
        {
          /*вывести на экран */
          printf ("-> %s\n", msgbuf);
          msglen = 0;
        }
      /*"освободить" mutex */
      pthread_mutex_unlock (&mutex);
      /*Поместить задачу в конец очереди готовых задач
         с тем же приоритетом */
      sched_yield ();
```

```
    }
}

int main ()
{
  /* дескрипторы задач */
  pthread_t writer_thread, reader_thread;

  /* создать mutex */
  if (pthread_mutex_init (&mutex, 0))
    {
      fprintf (stderr, "Cannot init mutex\n");
      return 1;
    }

  /* создать задачу reader */
  if (pthread_create (&reader_thread, 0, reader, 0))
    {
      fprintf (stderr, "Cannot create reader thread\n");
      return 2;
    }
  /* создать задачу writer */
  if (pthread_create (&writer_thread, 0, writer, 0))
    {
      fprintf (stderr, "Cannot create writer thread\n");
      return 3;
    }

  /* ждать окончания задачи reader_thread */
  pthread_join (reader_thread, 0);

  /* ждать окончания задачи writer_thread */
  pthread_join (writer_thread, 0);

  /* удалить mutex */
```

```
  if (pthread_mutex_destroy (&mutex))
    {
      fprintf (stderr, "Cannot destroy mutex\n");
      return 4;
    }

  return 0;
}
```

Процедура `reader` в бесконечном цикле, прерываемом по ошибке ввода:

1. обеспечивает себе исключительный доступ к общим переменным с помощью блокировки `mutex`;
2. если предыдущее сообщение обработано сервером (т. е. длина сообщения `msglen` равна 0), то считывает строку со стандартного ввода с помощью `fgets`;
3. в случае успешного чтения устанавливает длину сообщения `msglen`, иначе: прерывает цикл, устанавливает переменную `done` в 1, разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex` и завершает работу;
4. разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex`;
5. для обеспечения переключения задач к задаче-серверу использует функцию `sched_yield` (иначе бы этот цикл работал вхолостую до окончания кванта времени, выделенного текущей задаче); более надежное и универсальное решение см. в разделе 10.3.6.

Процедура `writer` в бесконечном цикле, прерываемом в случае ненулевого значения `done`:

1. обеспечивает себе исключительный доступ к общим переменным с помощью блокировки `mutex`;

2. если `done` не равно 0, то разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex` и завершает работу;
3. если есть сообщение для обработки сервером (т. е. длина сообщения `msglen` не равна 0), то выводит строку с помощью `printf` и устанавливает `msglen` в 0;
4. разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex`;
5. для обеспечения переключения задач к задачам-клиентам использует функцию `sched_yield`.

10.2. Пример multithread-программы, вычисляющей определенный интеграл

Рассмотрим в качестве примера программу, вычисляющую определенный интеграл. Идея ускорения работы описана в разделе 1.2, с. 12. Текст программы см. в разделе 1.4, с. 18. Задачи (threads) создаются в основном (т. е. изначально запущенном пользователем) процессе. Идентификаторы задач, полученные от `pthread_create`, запоминаются в массиве `threads` и используются основным процессом для ожидания завершения запущенных задач с помощью функции `pthread_join`. В каждой из запущенных задач работает функция `process_function` с аргументом — номером этой задачи. Эта функция вычисляет свою часть интеграла с помощью функции `integrate` и прибавляет его к ответу `total`, обеспечивая взаимное исключение при доступе к переменной `total` с помощью mutex `total_mutex`.

10.3. Объекты синхронизации типа condvar

Объект синхронизации типа condvar дает возможность задаче ожидать выполнения некоторых условий. Фактически является событием E (см. раздел 8.3) с дополнительными функциями по работе с указанным объектом mutex. Как и mutex, объект condvar явно размещается в разделяемой между задачами (про-

цессами) памяти, что позволяет обойтись без системных вызовов при синхронизации между задачами (threads). Для доступа к объекту condvar E определены три примитивные операции:

- $Wait(E, m)$ (где m типа mutex) — производит следующие действия (первые два из них **неделимы**, т. е. переключение задач во время их исполнения запрещено):
  - вызвать $Unlock(m)$ для текущей задачи (т. е. вызвавшей эту операцию),
  - вызвать $Wait(E)$,
  - вызвать $Lock(m)$ (произойдет, когда текущая задача «дождется» события E),

  mutex m перед вызовом этой функции должен быть **заблокированным**;
- $Signal(E)$ — вызвать $Signal(E)$, если нет ожидающих задач, то ничего не происходит;
- $Broadcast(E)$ — вызвать $Broadcast(E)$, если нет ожидающих задач, то ничего не происходит.

Рассмотрим пример использования объектов condvar. Пусть есть задачи $T_0, T_1, \ldots, T_n$, разделяющие общую область памяти X. Для синхронизации доступа к X используется mutex m. Пусть задача T_0 активизируется, только если выполнено некоторое условие $P(X)$. Для обеспечения этого взаимодействия используем объект condvar E. Тогда алгоритм работы T_0 может быть схематически записан так:

```
T_0 ()
{
  for (;;)
    {
      Lock (m);
      Wait (E, m);
      /* Можем работать с разделяемой памятью X,
         поскольку являемся владельцем mutex m */
      if (P(X))
        {
```

```
            /* Условие выполнено:
               исполняем необходимые действия */
            ...
          }
        Unlock (m);
    }
}
```

Тогда алгоритм работы T_i ($i = 1, \ldots, n$ может быть схематически записан так:

```
T_i ()
{
  for (;;)
    {
      Lock (m);
      /* Можем работать с разделяемой памятью X,
         поскольку являемся владельцем mutex m */
      /* Записываем данные в X */
      ...

      Signal (E);
      Unlock (m);
    }
}
```

10.3.1. Функция pthread_cond_init

Функция `pthread_cond_init`, описанная в заголовочном файле `<pthread.h>`, позволяет инициализировать condvar. Прототип:

```
int pthread_cond_init (pthread_cond_t *cond,
  pthread_condattr_t *attr);
```

где

- `cond` — указатель на condvar,
- `attr` — указатель на структуру с атрибутами condvar (может быть 0 для использования значений по умолчанию).

Функция возвращает 0 в случае успеха.

Для condvar (объекта типа `pthread_cond_t`) вместо функции `pthread_cond_init` можно использовать статическую инициализацию при объявлении:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

где макрос `PTHREAD_COND_INITIALIZER` определен в `<pthread.h>`. Последняя запись эквивалентна последовательности:

```
pthread_cond_t cond;
pthread_cond_init (&cond, 0);
```

но, в отличие от нее, может быть использована для инициализации глобальных объектов.

10.3.2. Функция pthread_cond_signal

Функция `pthread_cond_signal`, описанная в заголовочном файле `<pthread.h>`, позволяет возобновить работу одной из задач, ожидающей condvar (какой именно, не определено), если таких задач нет, то ничего не происходит. Прототип:

```
int pthread_cond_signal (pthread_cond_t *cond);
```

где `cond` — указатель на condvar. Функция возвращает 0 в случае успеха.

10.3.3. Функция pthread_cond_broadcast

Функция `pthread_cond_broadcast`, описанная в заголовочном файле `<pthread.h>`, позволяет возобновить работу всех задач, ожидающих condvar, если таких задач нет, то ничего не происходит. Прототип:

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

где `cond` — указатель на condvar. Функция возвращает 0 в случае успеха.

10.3.4. Функция pthread_cond_wait

Функция `pthread_cond_wait`, описанная в заголовочном файле `<pthread.h>`, разблокирует mutex (как при использовании `pthread_unlock_mutex`) и останавливает работу текущей задачи, пока какая-нибудь другая задача не вызовет функцию `pthread_cond_signal` или функцию `pthread_cond_broadcast`, причем эти операции **неделимы** (не допускается переключение задач между ними). После возвращения из состояния ожидания эта функция блокирует mutex (как при `pthread_lock_mutex`). Прототип:

```
int pthread_cond_wait (pthread_cond_t *cond,
  pthread_mutex_t *mutex);
```

где `cond` – указатель на condvar, `mutex` – указатель на mutex. Функция возвращает 0 в случае успеха.

10.3.5. Функция pthread_cond_destroy

Функция `pthread_cond_destroy`, описанная в заголовочном файле `<pthread.h>`, позволяет удалить condvar, который не должен в этот момент ожидаться (с помощью `pthread_cond_wait`) ни одной задачей. Прототип:

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

где `cond` – указатель на condvar. Функция возвращает 0 в случае успеха.

10.3.6. Пример использования condvar

Рассмотрим ту же задачу, что и в разделах 8.2.4 и 10.1.6.

Приведенная в разделе 10.1.6 программа является не совсем корректной в следующем смысле. Каждая из функций `reader/writer` делает предположение о том, что вторая функция `writer/reader` в момент вызова функции `sched_yield` ждет только освобождения mutex `mutex`. Для данной программы это предположение верно, но в общем случае требуется обеспечить,

чтобы после выполнения функции reader/writer следующей выполнялась writer/reader, если же последняя занята (не обработала предыдущее сообщение), то текущая задача должна перейти в состояние ожидания, а не потреблять процессорные ресурсы в бесполезном цикле опроса. Ниже приводится модифицированный вариант этой программы, в котором помимо объекта mutex для взаимного исключения при доступе к разделяемым данным используется объект condvar для синхронизации самих задач.

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

#define BUF_LEN 256

/* Объект синхронизации типа mutex */
pthread_mutex_t mutex;
/* Объект синхронизации типа condvar */
pthread_cond_t  condvar;
/* Признак окончания работы */
int done;
/* Длина сообщения */
int msglen;
/* Буфер для сообщения */
char msgbuf[BUF_LEN];

/* Считать сообщение и поместить его в буфер.
   Функция работает как независимая задача, вызываемая
   операционной системой, поэтому прототип фиксирован.
   Аргумент argp не используется. */
void * reader (void *argp)
{
  for (;;)
    {
```

```
      /* "захватить" mutex */
      pthread_mutex_lock (&mutex);
      if (!msglen)
        {
          /* Считать сообщение. Выход по Ctrl+D */
          if (!fgets (msgbuf, BUF_LEN, stdin))
            break;
          msglen = strlen (msgbuf) + 1;
          /* Послать сигнал condvar */
          pthread_cond_signal (&condvar);
        }
      while (msglen)
        {
          /* Освободить mutex и ждать сигнала от
             condvar, затем "захватить" mutex опять */
          pthread_cond_wait (&condvar, &mutex);
        }
      /* "освободить" mutex */
      pthread_mutex_unlock (&mutex);
    }
  /* Напечатать идентификатор текущей задачи */
  printf ("Thread %x exits\n", (int)pthread_self ());
  done = 1;
  pthread_cond_signal (&condvar);
  /* "освободить" mutex */
  pthread_mutex_unlock (&mutex);
  /* завершить задачу */
  return 0;
}

/* Ждать сообщения, по его поступлении вывести его на
   экран. Функция работает как независимая задача,
   вызываемая операционной системой, поэтому прототип
   фиксирован. Аргумент argp не используется. */
void * writer (void * argp)
```

```
{
  for (;;)
    {
      /* "захватить" mutex */
      pthread_mutex_lock (&mutex);
      while (!msglen)
        {
          if (!done)
            {
              /* Если еще есть задачи-клиенты, то
                 освободить mutex и ждать сигнала от
                 condvar, затем "захватить" mutex опять*
              pthread_cond_wait (&condvar, &mutex);
            }
          if (done)
            {
              /*Напечатать идентификатор текущей задачи*
              printf ("Thread %x exits\n",
                      (int)pthread_self ());
              /* "освободить" mutex */
              pthread_mutex_unlock (&mutex);
              /* завершить задачу */
              return 0;
            }
        }
      /* вывести на экран */
      printf ("-> %s\n", msgbuf);
      msglen = 0;
      /* Послать сигнал condvar */
      pthread_cond_signal (&condvar);
      /* "освободить" mutex */
      pthread_mutex_unlock (&mutex);
    }
}
```

```
int main ()
{
  /* дескрипторы задач */
  pthread_t writer_thread, reader_thread;

  /* создать mutex */
  if (pthread_mutex_init (&mutex, 0))
    {
      fprintf (stderr, "Cannot init mutex\n");
      return 1;
    }
  /* создать condvar */
  if (pthread_cond_init (&condvar, 0))
    {
      fprintf (stderr, "Cannot init condvar\n");
      return 2;
    }

  /* создать задачу reader */
  if (pthread_create (&reader_thread, 0, reader, 0))
    {
      fprintf (stderr, "Cannot create reader thread\n");
      return 3;
    }
  /* создать задачу writer */
  if (pthread_create (&writer_thread, 0, writer, 0))
    {
      fprintf (stderr, "Cannot create writer thread\n");
      return 4;
    }

  /* ждать окончания задачи reader_thread */
  pthread_join (reader_thread, 0);

  /* ждать окончания задачи writer_thread */
```

```
  pthread_join (writer_thread, 0);

  /* удалить mutex */
  if (pthread_mutex_destroy (&mutex))
    {
      fprintf (stderr, "Cannot destroy mutex\n");
      return 5;
    }
  /* удалить condvar */
  if (pthread_cond_destroy (&condvar))
    {
      fprintf (stderr, "Cannot destroy condvar\n");
      return 6;
    }

  return 0;
}
```

Процедура `reader` в бесконечном цикле, прерываемом по ошибке ввода:

1. обеспечивает себе исключительный доступ к общим переменным с помощью блокировки `mutex`;
2. если предыдущее сообщение обработано сервером (т. е. длина сообщения `msglen` равна 0), то считывает строку со стандартного ввода с помощью `fgets`;
3. в случае успешного чтения устанавливает длину сообщения `msglen`, иначе: прерывает цикл, устанавливает переменную `done` в 1, посылает сигнал в `condvar` (означающий наличие очередного сообщения для обработки сервером), разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex` и завершает работу;
4. посылает сигнал в `condvar`, означающий наличие очередного сообщения для обработки сервером;

5. ожидает обработки сообщения сервером, для чего, пока `msglen` не станет равной 0, ожидает прихода сигнала в `condvar`;
6. разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex`.

Процедура `writer` в бесконечном цикле, прерываемом в случае ненулевого значения `done`:

1. обеспечивает себе исключительный доступ к общим переменным с помощью блокировки `mutex`;
2. ожидает прихода сообщения от клиента, для чего, пока `msglen` не станет равной 0, ожидает прихода сигнала в `condvar`;
3. если `done` не равно 0, то разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex` и завершает работу;
4. дождавшись сообщения для обработки сервером (т. е. длина сообщения `msglen` не равна 0), выводит строку с помощью `printf` и устанавливает `msglen` в 0;
5. посылает сигнал в `condvar`, означающий окончание обработки сервером очередного сообщения;
6. разрешает другим задачам работать с общими переменными с помощью разблокировки `mutex`.

10.4. Пример multithread-программы, вычисляющей произведение матрицы на вектор

Рассмотрим задачу вычисления произведения матрицы на вектор (см. также раздел 8.5). Для повышения скорости работы на многопроцессорной вычислительной установке с общей памятью создадим несколько задач (по числу процессоров) и разделим работу между ними. Основным элементом синхронизации задач будет являться функция `synchronize`, которая дает возможность задаче ожидать, пока эту же функцию вызовут

все другие задачи (ср. с одноименной функцией в разделе 8.5). Файлы проекта:

- `synchronize.c`, `synchronize.h` — исходный текст и соответствующий заголовочный файл для функции синхронизации;
- `get_time.c`, `get_time.h` — исходный текст и соответствующий заголовочный файл для функций работы со временем, которые совпадают с одноименными файлами, приведенными в разделе 8.5); функция `get_time` выдает процессорное время, затраченное на задачу (thread), в которой ее вызвали;
- `main.c` — запуск задач и вывод результатов;
- `matrices.c`, `matrices.h` — исходный текст и соответствующий заголовочный файл для функций, работающих с матрицами;
- `Makefile` — для сборки проекта.

Заголовочный файл `synchronize.h`:

```
void synchronize (int total_threads);
```

Файл `synchronize.c`:

```
#include <pthread.h>
#include "synchronize.h"

/* Дождаться в текущем потоке остальных потоков
   (из общего числа total_threads). */
void synchronize (int total_threads)
{
  /* Объект синхронизации типа mutex */
  static pthread_mutex_t mutex
    = PTHREAD_MUTEX_INITIALIZER;
  /* Объект синхронизации типа condvar */
  static pthread_cond_t condvar_in
    = PTHREAD_COND_INITIALIZER;
  /* Объект синхронизации типа condvar */
  static pthread_cond_t condvar_out
    = PTHREAD_COND_INITIALIZER;
```

```
/* Число пришедших в функцию задач */
static int threads_in = 0;
/* Число ожидающих выхода из функции задач */
static int threads_out = 0;

/* "захватить" mutex для работы с переменными
   threads_in и threads_out */
pthread_mutex_lock (&mutex);

/* увеличить на 1 количество прибывших в
   эту функцию задач */
threads_in++;

/* проверяем количество прибывших задач */
if (threads_in >= total_threads)
  {
    /* текущий поток пришел последним */
    /* устанавливаем начальное значение
       для threads_out */
    threads_out = 0;

    /* разрешаем остальным продолжать работу */
    pthread_cond_broadcast (&condvar_in);
  }
else
  {
    /* есть еще не пришедшие потоки */

    /* ожидаем, пока в эту функцию не придут
       все потоки */
    while (threads_in < total_threads)
      {
        /* ожидаем разрешения продолжить работу:
           освободить mutex и ждать сигнала от
           condvar, затем "захватить" mutex опять */
```

```
        pthread_cond_wait (&condvar_in, &mutex);
      }
  }

/* увеличить на 1 количество ожидающих выхода задач */
threads_out++;

/* проверяем количество прибывших задач */
if (threads_out >= total_threads)
  {
    /* текущий поток пришел в очередь последним */
    /* устанавливаем начальное значение
       для threads_in */
    threads_in = 0;

    /* разрешаем остальным продолжать работу */
    pthread_cond_broadcast (&condvar_out);
  }
else
  {
    /* в очереди ожидания еще есть потоки */

    /* ожидаем, пока в очередь ожидания не придет
       последний поток */
    while (threads_out < total_threads)
      {
        /* ожидаем разрешения продолжить работу:
           освободить mutex и ждать сигнала от
           condvar, затем "захватить" mutex опять */
        pthread_cond_wait (&condvar_out, &mutex);
      }
  }

/* "освободить" mutex */
pthread_mutex_unlock (&mutex);
```

```
}
```

Функция `synchronize` получилась достаточно сложной, поскольку она допускает использование внутри циклов:

```
for (...)
  {
    ...
    synchronize (total_threads);
    ...
  }
```

В этой ситуации недостаточно подсчитывать, **сколько задач вошло** в эту функцию, поскольку возможно, что, выйдя из нее, одна из задач опять войдет в функцию раньше, чем остальные успеют из нее выйти. Поэтому необходимо подсчитывать еще **сколько задач вышло** из функции. В функции `synchronize` объект `mutex` обеспечивает взаимное исключение задач при доступе к общим для всех задач переменным: `threads_in` — счетчику числа вошедших в функцию задач, `threads_out` — счетчику числа ожидающих выхода из функции задач. Объект `condvar_in` используется для ожидания, пока счетчик `threads_in` не примет значение `total_threads` (т. е. в функцию не войдут все задачи). Сигнал о наступлении этого события посылает последняя пришедшая задача, она же устанавливает в начальное значение 0 счетчик `threads_out`. Объект `condvar_out` используется для ожидания, пока счетчик `threads_out` не примет значение `total_threads` (т. е. в очередь на выход не встанут все задачи). Сигнал о наступлении этого события посылает последняя вставшая в очередь задача, она же устанавливает в начальное значение 0 счетчик `threads_in`.

Файл `main.c`:

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

```
#include "get_time.h"

/* Аргументы для потока */
typedef struct _ARGS
{
  double *matrix;        /* матрица */
  double *vector;        /* вектор */
  double *result;        /* результирующий вектор */
  int n;                 /* размер матрицы и векторов
  int thread_num;        /* номер задачи */
  int total_threads;     /* всего задач */
} ARGS;

/* Суммарное время работы всех задач */
static long int threads_total_time = 0;
/* Объект типа mutex для синхронизации доступа к
   threads_total_time */
static pthread_mutex_t threads_total_time_mutex
  = PTHREAD_MUTEX_INITIALIZER;

/* Количество тестов (для отладки) */
#define N_TESTS 10

/* Умножение матрицы на вектор для одной задачи */
void * matrix_mult_vector_threaded (void *pa)
{
  ARGS *pargs = (ARGS*)pa;
  long int t;
  int i;

  printf ("Thread %d started\n", pargs->thread_num);
  t = get_time ();       /* время начала работы */
  for (i = 0; i < N_TESTS; i++)
  {
    matrix_mult_vector (pargs->matrix, pargs->vector
```

```
                        pargs->result, pargs->n,
                        pargs->thread_num,
                        pargs->total_threads);
    printf ("Thread %d mult %d times\n",
            pargs->thread_num, i);
  }
  t = get_time () - t;  /* время конца работы */

  /* Суммируем времена работы */
  /* "захватить" mutex для работы с threads_total_time */
  pthread_mutex_lock (&threads_total_time_mutex);
  threads_total_time += t;
  /* "освободить" mutex */
  pthread_mutex_unlock (&threads_total_time_mutex);
  printf ("Thread %d finished, time = %ld\n",
          pargs->thread_num, t);

  return 0;
}


int main ()
{
  /* массив идентификаторов созданных задач */
  pthread_t * threads;
  /* массив аргументов для созданных задач */
  ARGS * args;
  /* число создаваемых задач */
  int nthreads;
  /* астрономическое время работы всего процесса */
  long int t_full;

  int n;                /* размер матрицы и векторов */
  double *matrix;       /* матрица */
  double *vector;       /* вектор */
```

```
double *result;        /* результирующий вектор */
int i, l;

printf ("Input threads number: ");
scanf ("%d", &nthreads);

if (!(threads = (pthread_t*)
                malloc (nthreads * sizeof (pthread_t))))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 1;
  }
if (!(args = (ARGS*) malloc (nthreads * sizeof (ARGS))))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 2;
  }

printf ("matrix size = ");
scanf ("%d", &n);

/* Выделение памяти под массивы */
if (!(matrix = (double*)
               malloc (n * n * sizeof (double))))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 3;
  }
if (!(vector = (double*) malloc (n * sizeof (double))))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 4;
  }
if (!(result = (double*) malloc (n * sizeof (double))))
  {
```

```
    fprintf (stderr, "Not enough memory!\n");
    return 5;
  }

/* Инициализация массивов */
init_matrix (matrix, n);
init_vector (vector, n);
printf ("Matrix:\n");
print_matrix (matrix, n);
printf ("Vector:\n");
print_vector (vector, n);

l = (n * n + 2 * n) * sizeof (double);
printf ("Allocated %d bytes (%dKb or %dMb) of memory\n",
        l, l >> 10, l >> 20);

/* Инициализация аргументов задач */
for (i = 0; i < nthreads; i++)
  {
    args[i].matrix = matrix;
    args[i].vector = vector;
    args[i].result = result;
    args[i].n = n;
    args[i].thread_num = i;
    args[i].total_threads = nthreads;
  }

/* Засекаем астрономическое время начала работы задач*/
t_full = get_full_time ();
/* Запускаем задачи */
for (i = 0; i < nthreads; i++)
  {
    if (pthread_create (threads + i, 0,
                        matrix_mult_vector_threaded,
                        args + i))
```

```
        {
          fprintf (stderr, "cannot create thread #%d!\n",
                   i);
          return 10;
        }
    }
  /* Ожидаем окончания задач */
  for (i = 0; i < nthreads; i++)
    {
      if (pthread_join (threads[i], 0))
        fprintf (stderr, "cannot wait thread #%d!\n", i);
    }
  t_full = get_full_time () - t_full;
  if (t_full == 0)
    t_full = 1; /* очень быстрый компьютер... */

  /* Здесь можно работать с результатом */
  print_vector (result, n);

  /* Освобождаем память */
  free (threads);
  free (args);
  free (matrix);
  free (vector);
  free (result);

  printf ("Total full time = %ld, \
total threads time = %ld (%ld%%), per thread = %ld\n",
          t_full, threads_total_time,
          (threads_total_time * 100) / t_full,
          threads_total_time / nthreads);

  return 0;
}
```

Функция matrix_mult_vector_threaded запускается в функции main в виде задач. Каждой из задач необходимо передать свой набор аргументов. Но функция, запускаемая как задача, может иметь только один аргумент типа void*. Для решения этой проблемы определяется структура данных ARGS, содержащая необходимые аргументы. Указатель на эту структуру передается функции matrix_mult_vector_threaded как void* и преобразуется к типу ARGS*. Затем вызывается matrix_mult_vector, вычисляющая компоненты ответа с индексами в диапазоне

```
n * thread_num / total_threads, ...,
n * (thread_num + 1)  / total_threads - 1
```

Для ожидания завершения всех задач используется функция synchronize.

Для вычисления суммарного процессорного времени, затраченного на процесс, в функции matrix_mult_vector_threaded время работы каждой задачи прибавляется к значению разделяемой переменной threads_total_time. Для взаимного исключения задач при доступе к этой переменной используется mutex threads_total_time_mutex.

Для целей отладки и более точного замера времени работы функция вычисления произведения матрицы на вектор вызывается в цикле N_TESTS раз.

Заголовочный файл matrices.h:

```
void init_matrix (double * matrix, int n);
void init_vector (double * vector, int n);
void print_matrix (double * matrix, int n);
void print_vector (double * vector, int n);
void matrix_mult_vector (double *a, double *b, double *c,
                         int n, int thread_num,
                         int total_threads);
```

Файл matrices.c:

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

```
#include "matrices.h"
#include "synchronize.h"

/* Инициализация матрицы */
void init_matrix (double * matrix, int n)
{
  int i, j;
  double *a = matrix;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      *(a++) = (i > j) ? i : j;
}

/* Инициализация вектора */
void init_vector (double * vector, int n)
{
  int i;
  double *b = vector;

  for (i = 0; i < n; i++)
    *(b++) = 1.;
}

#define N_MAX   5

/* Вывод матрицы */
void print_matrix (double * matrix, int n)
{
  int i, j;
  int m = (n > N_MAX ? N_MAX : n);

  for (i = 0; i < m; i++)
    {
      for (j = 0; j < m; j++)
```

```
        printf (" %12.6lf", matrix[i * n + j]);
      printf ("\n");
    }
}

/* Вывод вектора */
void print_vector (double * vector, int n)
{
  int i;
  int m = (n > N_MAX ? N_MAX : n);

  for (i = 0; i < m; i++)
      printf (" %12.6lf", vector[i]);
  printf ("\n");
}

/* Умножить матрицу a на вектор b, c = ab для задачи с
   номером thread_num из общего количества
   total_threads. */
void matrix_mult_vector (double *a, double *b, double *c,
                         int n, int thread_num,
                         int total_threads)
{
  int i, j;
  double *p, s;
  int first_row, last_row;

  /* Первая участвующая строка матрицы */
  first_row = n * thread_num;
  first_row /= total_threads;
  /* Последняя участвующая строка матрицы */
  last_row = n * (thread_num + 1);
  last_row = last_row / total_threads - 1;

  for (i = first_row, p = a + i * n; i <= last_row; i++)
```

```
    {
      for (s = 0., j = 0; j < n; j++)
        s += *(p++) * b[j];
      c[i] = s;
    }
  synchronize (total_threads);
}
```

Файл Makefile:

```
NAME    = thread_mult

DEBUG   = -g
CC      = gcc -c
LD      = gcc
CFLAGS  = $(DEBUG) -W -Wall
LIBS    = -lpthread -lm
LDFLAGS = $(DEBUG)

OBJS = main.o matrices.o synchronize.o get_time.o

all : $(NAME)

$(NAME) : $(OBJS)
        $(LD) $(LDFLAGS) $^ $(LIBS) -o $@

.c.o:
        $(CC) $(CFLAGS) $< -o $@

clean:
        rm -f $(OBJS) $(NAME)

main.o          : main.c matrices.h get_time.h
matrices.o      : matrices.c matrices.h synchronize.h
synchronize.o   : synchronize.c synchronize.h
get_time.o      : get_time.c get_time.h
```

10.5. Влияние дисциплины доступа к памяти на эффективность параллельной программы

Попробуем обобщить программу предыдущего раздела, получив из нее multithread-программу умножения двух матриц $C = AB$. Если обозначить через x_i i-й столбец матрицы X, а через $X = [x_1, \ldots, x_n]$ — матрицу, составленную из столбцов x_i, то по определению умножения матриц

$$C = [c_1, \ldots, c_n] = AB = [Ab_1, \ldots, Ab_n],$$

что позволяет легко получить из программы умножения матрицы на вектор программу умножения матрицы на матрицу. (Заметим, что этот подход полностью игнорирует все, что сказано в разделе 2.7 об эффективном умножении матриц.) Мы приведем только текст функции `matrix_mult_matrix`, обобщающей описанную выше `matrix_mult_vector` и работающую совершенно аналогично:

```
#include <stdio.h>
#include "matrices.h"
#include "synchronize.h"

/* Умножить матрицу a на матрицу b, c = ab для задачи
   с номером thread_num из общего количества
   total_threads */
void matrix_mult_matrix (double *a, double *b, double *c,
                         int n, int thread_num,
                         int total_threads)
{
  int i, j, m;
  double *pa, *pb, *pc, s;
  int first_row, last_row;

  /* Первая участвующая строка матрицы */
  first_row = n * thread_num;
  first_row /= total_threads;
```

```
  /* Последняя участвующая строка матрицы */
  last_row = n * (thread_num + 1);
  last_row = last_row / total_threads - 1;

  for (m = 0, pc = c; m < n; m++, pc++)
    {
      for (i = first_row, pa = a + i * n, pb = b + m;
           i <= last_row; i++)
        {
          for (s = 0., j = 0; j < n; j++)
            s += *(pa++) * pb[j * n];
          pc[i * n] = s;
        }
    }
  synchronize (total_threads);
}
```

Отношение времени работы «обычной» программы умножения матриц (пример 1 раздела 2.7) к времени работы этой программы (для процесса в целом и для одной задачи) для двух SMP-систем при $n = 2000$ приведено в табл. 10.1.

Эта таблица показывает:

- **практически 100% эффективность распараллеливания** (астрономическое время работы программы совпадает с временем работы одной задачи);

Таблица 10.1. Относительное время работы параллельной программы умножения матриц

Число задач	2 × Pentium III		4 × Pentium Pro	
	процесс	1 thread	процесс	1 thread
1	1.00	1.00	1.00	1.00
2	1.05	1.05	0.73	0.73
4	—	—	1.70	1.70

- **практически полное отсутствие ускорения в работе** (во второй системе астрономическое время на двух процессорах даже увеличилось в 1.37 раз по сравнению с одним процессором);
- **практически одинаковое время работы** «обычной» программы на двух рассматриваемых системах (хотя частоты процессоров отличаются почти в пять раз).

Следовательно, предложенная программа **совершенно непригодна** для параллельных вычислительных установок. Основной причиной этого является неудачная дисциплина доступа к общей памяти, поскольку каждая из задач **одновременно** обращается к **одним и тем же** элементам матрицы b. Несмотря на то, что эти элементы используются только для чтения, это приводит к конфликтам по данным в подсистемах оперативной памяти и кэш-памяти. Заметим, что подобный эффект можно отнести к недоработкам при создании управляющей аппаратной логики рассматриваемых систем.

Попробуем исправить эту ситуацию, организовав работу программы таким образом, чтобы исключить одновременный доступ нескольких задач к одним и тем же элементам общей памяти. Улучшенный вариант умножения матрицы на матрицу и объяснение его работы приведены ниже:

```
#include <stdio.h>
#include "matrices.h"
#include "synchronize.h"

/* Умножить матрицу a на матрицу b, c = ab для задачи
   с номером thread_num из общего количества
   total_threads */
void matrix_mult_matrix (double *a, double *b, double *c
                         int n, int thread_num,
                         int total_threads)
{
  int i, j, k, l, m;
```

```
int first_row, last_row, len;
int first_row_k, last_row_k;
double *pa, *pb, *pc, s;

/* Первая участвующая строка матрицы */
first_row = n * thread_num;
first_row /= total_threads;
/* Последняя участвующая строка матрицы */
last_row = n * (thread_num + 1);
last_row = last_row / total_threads - 1;

/* Обнуляем результат */
len = (last_row - first_row + 1) * n; /* длина c */
for (i = 0, pc = c + first_row * n; i < len; i++)
  *(pc++) = 0.;

/* Цикл по блокам */
for (l = 0; l < total_threads; l++)
  {
    /* Текущий номер блока для полосы b */
    k = (thread_num + l) % total_threads;
    /* Первая строка полосы матрицы b */
    first_row_k = n * k;
    first_row_k /= total_threads;
    /* Последняя строка полосы матрицы b */
    last_row_k = n * (k + 1);
    last_row_k = last_row_k / total_threads - 1;

    /* Умножаем прямоугольный блок матрицы a
       в строках  first_row ... last_row
       и столбцах first_row_k ... last_row_k
       на столбцы полосы b, стоящие в строках
       first_row_k ... first_row_k */
    for (m = 0, pb = b, pc = c; m < n; m++, pb++, pc++)
      {
```

```
        for (i = first_row; i <= last_row; i++)
          {
            for (s = 0., j = first_row_k,
                 pa = a + i * n + j;
                 j <= last_row_k; j++)
              s += *(pa++) * pb[j * n];
            pc[i * n] += s;
          }
      }
    synchronize (total_threads);
  }
}
```

Функция `matrix_mult_matrix` в цикле по $l = 0, 1, \ldots, p-1$, $p =$ `total_threads`, вычисляет матрицу, являющуюся произведением блока матрицы A, стоящего в строках

$$nm/p, \ldots, n(m+1)/p - 1, \quad m = \texttt{thread\_num} \tag{10.1}$$

и столбцах

$$(nm/p + l) \pmod p, \ldots, (n(m+1)/p - 1 + l) \pmod p, \tag{10.2}$$

и блока матрицы B, стоящего в строках с номерами (10.2) и столбцах

$$1, \ldots, n. \tag{10.3}$$

Эта матрица прибавляется к блоку матрицы C, стоящему в строках с номерами (10.1) и столбцах с номерами (10.3). В этом цикле по l мы использовали `synchronize` (см. раздел 10.4) для того, чтобы гарантированно исключить одновременный доступ нескольких задач к одним и тем же элементам b.

Иллюстрация этого процесса при `total_threads` $= 3$ приведена на рис. 10.1: в каждой из задач 1, 2, 3 (здесь мы для удобства нумеруем их, начиная с 1) вычисляются произведения блоков матриц A и B, отмеченных на рисунке соответствующей цифрой, и прибавляются к результату в матрице C.

Отношение времени работы «обычной» программы умножения матриц (пример 1 раздела 2.7) к времени работы этой программы (для процесса в целом и для одной задачи) для тех же

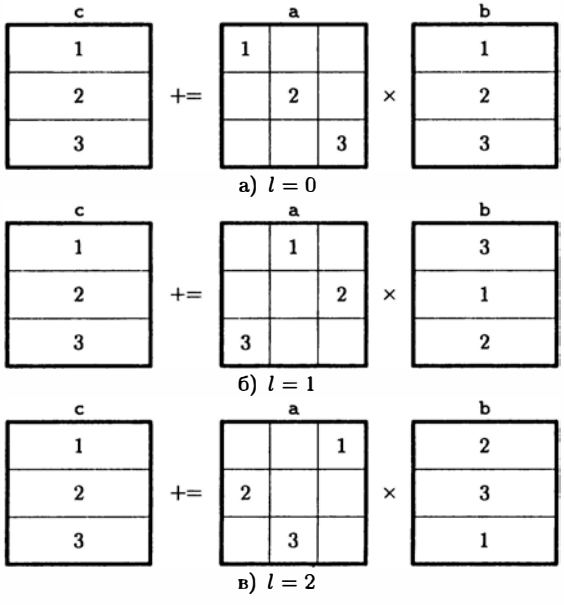


Рис. 10.1. Организация вычислений при умножении матрицы на матрицу

SMP-систем и того же n приведено в табл. 10.2, где в скобках приведено также отношение времени работы этой программы на 1 процессоре к времени ее работы (для процесса в целом и для одной задачи) на многих процессорах.

Таблица 10.2. Относительное время работы улучшенной параллельной программы умножения матриц

Число задач	2 × Pentium III		4 × Pentium Pro	
	процесс	1 thread	процесс	1 thread
1	0.95 (1.00)	0.95 (1.00)	0.74 (1.00)	0.74 (1.00)
2	3.56 (3.75)	3.56 (3.75)	1.40 (1.90)	1.40 (1.90)
4	—	—	2.53 (3.43)	2.53 (3.43)

Эта таблица показывает:

- **практически 100% эффективность распараллеливания** (астрономическое время работы программы совпадает с временем работы одной задачи);
- **хорошее ускорение в работе** при увеличении числа используемых процессоров (см. коэффициенты в скобках);
- **замедление работы «улучшенной» программы** на одном процессоре (это распространенное для параллельных программ явление в данной ситуации объясняется усложнением индексных выражений при обращении к массивам, что особенно чувствительно для второй системы с более низкой частотой процессоров);
- ускорение работы на величину, превышающую количество использованных процессоров (для первой системы — в 3.73 раз на 2-х процессорах), которое некоторые авторы называют **суперускорением** (это достаточно частое для параллельных программ явление здесь объясняется влиянием кэш-памяти, поскольку работа с подматрицами уменьшает количество вовлеченных элементов и увеличивает эффективность кэширования, см. подробное обсуждение этого в разделе 2.7).

Подчеркнем, что полученное ускорение работы в основном связано с изменением дисциплины обращения к общей памяти. Для дальнейшего повышения эффективности программы можно обратиться к таблице 2.1 и выбрать в последней версии функции `matrix_mult_matrix` для перемножения блоков матриц `a` и `b` алгоритм 5 вместо использованного алгоритма 1. В результате получаем следующую (уже «хорошую») подпрограмму:

```
#include <stdio.h>
#include "matrices.h"
#include "synchronize.h"

/* Размер блока, должен делиться на 2 */
#define N       40
```

```
/* Умножить матрицу a на матрицу b, c = ab для задачи
   с номером thread_num из общего количества
   total_threads */
/* Работает только для n/total_threads - четных (для всех
   n придется усложнять разворачивание цикла) */
void matrix_mult_matrix (double *a, double *b, double *c,
                         int n, int thread_num,
                         int total_threads)
{
  int i, j, k, l, m;
  int first_row, last_row, len;
  int first_row_k, last_row_k;
  double *pa, *pb, *pc;
  double s00, s01, s10, s11;
  int c_col, c_ncol, c_row, c_nrow, a_col, a_ncol;

  /* Первая участвующая строка матрицы */
  first_row = n * thread_num;
  first_row /= total_threads;
  /* Последняя участвующая строка матрицы */
  last_row = n * (thread_num + 1);
  last_row = last_row / total_threads - 1;

  /* Обнуляем результат */
  len = (last_row - first_row + 1) * n; /* длина c */
  for (i = 0, pc = c + first_row * n; i < len; i++)
    *(pc++) = 0.;

  /* Цикл по блокам */
  for (l = 0; l < total_threads; l++)
    {
      /* Текущий номер блока для полосы b */
      k = (thread_num + l) % total_threads;
      /* Первая строка полосы матрицы b */
      first_row_k = n * k;
      first_row_k /= total_threads;
```

```
/* Последняя строка полосы матрицы b */
last_row_k = n * (k + 1);
last_row_k = last_row_k / total_threads - 1;

/* Умножаем прямоугольный блок матрицы a
   в строках  first_row ... last_row
   и столбцах first_row_k ... last_row_k
   на столбцы полосы b, стоящие в строках
   first_row_k ... last_row_k
   Получаем блок матрицы c, находящийся в строках
   first_row ... last_row */
for (c_col = 0; c_col < n; c_col += N)
  {
    c_ncol = (c_col + N <= n ? c_col + N : n) - 1;
    for (c_row = first_row; c_row <= last_row;
         c_row += N)
      {
        c_nrow = (c_row + N - 1 <= last_row
                  ? c_row + N - 1 : last_row);
        /* Вычислить результирующий блок матрицы c
           с верхним левым углом (c_row, c_col)
           и правым нижним (c_nrow, c_ncol) */
        /* Вычисляем как произведения блоков матриц
           a и b */
        for (a_col = first_row_k;
             a_col <= last_row_k; a_col += N)
          {
            a_ncol = (a_col + N - 1 <= last_row_k
                      ? a_col + N - 1 : last_row_k);
            /* Вычисляем произведение
               блока матрицы
               a [(c_row,a_col) x (c_nrow,a_ncol)]
               на блок матрицы
               b [(a_col,c_col) x (a_ncol,c_ncol)]
               и прибавляем к блоку
               матрицы
```

```
                    c [(c_row,c_col) x (c_nrow,c_ncol)]
                 */
                for (m = c_col, pb = b + m, pc = c + m;
                     m <= c_ncol; m += 2, pb += 2,
                     pc += 2)
                  for (i = c_row; i <= c_nrow; i += 2)
                    {
                      s00 = s01 = s10 = s11 = 0.;
                      for (j = a_col,
                           pa = a + i * n + j;
                           j <= a_ncol; j++, pa++)
                        {
                          /* элемент (i, m) */
                          s00 += pa[0] * pb[j * n];
                          /* элемент (i, m + 1) */
                          s01 += pa[0] * pb[j * n + 1];
                          /* элемент (i + 1, m) */
                          s10 += pa[n] * pb[j * n];
                          /* элемент (i + 1, m + 1) */
                          s11 += pa[n] * pb[j * n + 1];
                        }
                      pc[i * n]           += s00;
                      pc[i * n + 1]       += s01;
                      pc[(i + 1) * n]     += s10;
                      pc[(i + 1) * n + 1] += s11;
                    }
              }
          }
        }
      synchronize (total_threads);
    }
}
```

Таблица 10.3. Относительное время работы хорошей параллельной программы умножения матриц

Число задач	2 × Pentium III		4 × Pentium Pro	
	процесс	1 thread	процесс	1 thread
1	13.5 (1.00)	13.5 (1.00)	3.10 (1.00)	3.10 (1.00)
2	26.1 (1.93)	26.1 (1.93)	6.20 (1.99)	6.20 (1.99)
4	—	—	12.4 (3.99)	12.4 (3.99)

Характеристики производительности этой программы приведены в табл. 10.3 (обозначения и условия эксперимента те же, что в табл. 10.2).

10.6. Пример multithread-программы, решающей задачу Дирихле для уравнения Пуассона

Рассмотрим решение задачи Дирихле для уравнения Пуассона в двумерной области $D = [0, l_1] \times [0, l_2]$:

$$-\Delta u(x,y) \equiv -\frac{\partial^2 u(x,y)}{\partial x^2} - \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y), \tag{10.4}$$
$$(x,y) \in D, u(x,y) = 0, \quad (x,y) \in \partial D.$$

Пусть $h_1, h_2 > 0$ — достаточно малы. Тогда

$$-\Delta u(x,y) \approx -(u(x-h_1,y) - 2u(x,y) + u(x+h_1,y))/h_1^2$$
$$-(u(x,y-h_2) - 2u(x,y) + u(x,y+h_2))/h_2^2.$$

Поэтому рассмотрим следующую конечномерную аппроксимацию задачи (10.4). Зададимся положительными целочисленными параметрами n_1, n_2. Положим

$$h_1 = l_1/n_1, \quad h_2 = l_2/n_2,$$
$$x_i = ih_1, \quad i = 0,1,\ldots,n_1, \; y_j = jh_2, \quad j = 0,1,\ldots,n_2,$$
$$u_{ij} = u(x_i, y_j), \quad i = 0,1,\ldots,n_1, \quad j = 0,1,\ldots,n_2,$$
$$f_{ij} = f(x_i, y_j), \quad i = 0,1,\ldots,n_1, \quad j = 0,1,\ldots,n_2.$$

Вместо задачи (10.4) рассмотрим следующую систему линейных уравнений:

$$-(u_{i-1,j} - 2u_{i,j} + u_{i+1,j})/h_1^2 - (u_{i,j-1} - 2u_{i,j} + u_{i,j+1})/h_2^2 = f_{i,j},$$
$$i = 1, 2, \ldots, n_1 - 1,\ j = 1, 2, \ldots, n_2 - 1,$$
$$u_{i,0} = 0,\ u_{i,n_2} = 0, \quad i = 0, 1, \ldots, n_1,$$
$$u_{0,j} = 0,\ u_{n_1,j} = 0, \quad j = 0, 1, \ldots, n_2. \tag{10.5}$$

Эту систему часто называют **разностной аппроксимацией** задачи (10.4).

Систему (10.5) можно представить в обычном виде:

$$Au = f, \tag{10.6}$$

где

$$u = (u_{ij})_{i=0,1,\ldots,n_1,\ j=0,1,\ldots,n_2}, \quad f = (f_{ij})_{i=0,1,\ldots,n_1,\ j=0,1,\ldots,n_2}$$

— $(n_1+1)\times(n_2+1)$ векторы, A — матрица $(n_1+1)(n_2+1)\times(n_1+1)(n_2+1)$, действие которой на вектор задается формулой

$$v = Au,$$
$$v_{ij} = -(u_{i-1,j} - 2u_{i,j} + u_{i+1,j})/h_1^2 - (u_{i,j-1} - 2u_{i,j} + u_{i,j+1})/h_2^2$$
$$i = 1, 2, \ldots, n_1 - 1,\ j = 1, 2, \ldots, n_2 - 1,$$
$$v_{i,0} = 0,\ v_{i,n_2} = 0, \quad i = 0, 1, \ldots, n_1,$$
$$v_{0,j} = 0,\ v_{n_1,j} = 0, \quad j = 0, 1, \ldots, n_2. \tag{10.7}$$

В силу большой размерности матрицы A систему (10.6) обычно решают тем или иным итерационным алгоритмом, для проведения которого достаточно уметь вычислять произведение матрицы на вектор, а саму матрицу A хранить не требуется. Мы рассмотрим простейший из таких алгоритмов — **метод Якоби**. Он состоит в вычислении последовательности приближений $\{u^k\}$,

$k = 0, 1, \ldots$, начиная с некоторого начального приближения u^0 (мы будем использовать $u^0 = 0$), по следующим формулам:

$$B\frac{u^{k+1} - u^k}{\tau_k} + Au^k = f, \quad k = 0, 1, \ldots, \tag{10.8}$$

где τ_k — итерационный параметр, B — диагональная матрица, равная главной диагонали матрицы A. Матрица B в итерационном процессе вида (10.8) называется предобуславливателем и в силу (10.7) ее действие на вектор задается формулой

$$\begin{gathered} v = Bu, \;\; v_{ij} = \left(\frac{2}{h_1^2} + \frac{2}{h_2^2}\right) u_{i,j}, \\ i = 1, 2, \ldots, n_1 - 1, \;\; j = 1, 2, \ldots, n_2 - 1, \\ v_{i,0} = 0, \;\; v_{i,n_2} = 0, \quad i = 0, 1, \ldots, n_1, \\ v_{0,j} = 0, \;\; v_{n_1,j} = 0, \quad j = 0, 1, \ldots, n_2. \end{gathered} \tag{10.9}$$

Вектор

$$r^k = f - Au^k$$

называется невязкой и его норма характеризует качество полученного приближения u^k. Обычно вычисления в процессе (10.8) продолжают, пока норма невязки не упадет в заданное число раз (например, 10^6) по сравнению с нормой r^0. Итерационный параметр τ_k в (10.8) мы будем выбирать **методом скорейшего спуска**:

$$\tau_k = (B^{-1}r^k, r^k)/(AB^{-1}r^k, B^{-1}r^k), \quad k = 0, 1, \ldots,$$

где $(\cdot, \cdot)$ означает евклидово скалярное произведение. Если обозначить $v^k = B^{-1}r^k$, то в силу симметричности матрицы A

$$\tau_k = \frac{(v^k, r^k)}{(Av^k, v^k)} = \frac{(v^k, f - Au^k)}{(Av^k, v^k)} = \frac{(f, v^k) - (Av^k, u^k)}{(Av^k, v^k)},$$

т. е.

$$\tau_k = ((f, v^k) - (Av^k, u^k))/(Av^k, v^k), \;\; v^k = B^{-1}r^k, \;\; k = 0, 1, \ldots. \tag{10.10}$$

Расчетные формулы нашего алгоритма, использующие 3 вектора: u, r, f, имеют вид:

1. пусть u содержит приближение на шаге k (0 на шаге 0);
2. вычисляем $r = Au$;
3. вычисляем невязку $r = f - r$ и ее норму;
4. если норма невязки меньше заданной, то заканчиваем вычисления;
5. вычисляем $r = B^{-1}r$;
6. вычисляем $\tau = ((f, r) - (Ar, u))/(Ar, r)$;
7. вычисляем приближение на шаге $k + 1$: $u = u + \tau r$;
8. переходим к пункту 1.

В выражении для τ вектор Ar не вычисляется целиком. Определяется очередной его элемент, и он сразу используется для нахождения значений скалярных произведений (Ar, u) и (Ar, r).

Для отладки программы мы используем в качестве области единичный квадрат $D = [0, 1] \times [0, 1]$, в качестве правой части функцию

$$f(x, y) = 2x(1 - x) + 2y(1 - y),$$

которой соответствует решение

$$u(x, y) = x(1 - x)y(1 - y).$$

Это позволит вычислять норму ошибки, т. е. норму разности между точным решением и полученным в итерационном процессе u^k.

Отметим, что метод Якоби для системы (10.5) является весьма неэффективным и рассматривается здесь для простоты.

Для повышения скорости работы на многопроцессорной вычислительной установке с общей памятью создадим несколько задач (по числу процессоров) и разделим работу между ними. Файлы проекта:

- `get_time.c`, `get_time.h` — исходный текст и соответствующий заголовочный файл для функций работы со временем (см. раздел 10.4);
- `reduce_sum.c`, `reduce_sum.h` — исходный текст и соответствующий заголовочный файл для функции вычисления суммы массивов, распределенных между процессами;

- `main.c` — ввод данных, запуск задач и вывод результатов;
- `init.c`, `init.h` — исходный текст и соответствующий заголовочный файл для функций, инициализирующих правую часть уравнения (10.4) и вычисляющих ошибку решения;
- `operators.c`, `operators.h` — исходный текст и соответствующий заголовочный файл для функций, вычисляющих различные операторы;
- `laplace.c`, `laplace.h` — исходный текст и соответствующий заголовочный файл для функции, решающей задачу;
- `Makefile` — для сборки проекта.

Заголовочный файл `reduce_sum.h`:

```
void reduce_sum (int total_threads, double* a, int k);
#define synchronize(X)  reduce_sum (X, 0, 0)
```

Здесь использовано макроопределение для получения функции `synchronize`, работающей так же, как одноименная функция из раздела 10.4.

Файл `reduce_sum.c`:

```
#include <pthread.h>
#include "reduce_sum.h"

/* Вычислить сумму элементов массива a длины k во всех
   потоках и вернуть ее в массиве a в каждом потоке
   (из общего числа total_threads). */
void reduce_sum (int total_threads, double* a, int k)
{
  /* Объект синхронизации типа mutex */
  static pthread_mutex_t mutex
    = PTHREAD_MUTEX_INITIALIZER;
  /* Объект синхронизации типа condvar */
  static pthread_cond_t condvar_in
    = PTHREAD_COND_INITIALIZER;
  /* Объект синхронизации типа condvar */
  static pthread_cond_t condvar_out
```

```
  = PTHREAD_COND_INITIALIZER;
/* Число пришедших в функцию задач */
static int threads_in = 0;
/* Число ожидающих выхода из функции задач */
static int threads_out = 0;
/* Указатель на массив с ответом */
static double *pres = 0;
int i;

/* "захватить" mutex для работы с переменными
   threads_in, threads_out, pres */
pthread_mutex_lock (&mutex);

/* если текущий поток пришел первым, то установить
   pres, иначе вычислить сумму с pres */
if (!pres)
  {
    /* первый вошедший поток */
    pres = a;
  }
else
  {
    /* вычислить сумму в остальных потоках */
    for (i = 0; i < k; i++)
      pres[i] += a[i];
  }

/* увеличить на 1 количество прибывших в
   эту функцию задач */
threads_in++;

/* проверяем количество прибывших задач */
if (threads_in >= total_threads)
  {
    /* текущий поток пришел последним */
```

```
    /* устанавливаем начальное значение
       для threads_out */
    threads_out = 0;

    /* разрешаем остальным продолжать работу */
    pthread_cond_broadcast (&condvar_in);
  }
else
  {
    /* есть еще не пришедшие потоки */

    /* ожидаем, пока в эту функцию не придут
       все потоки */
    while (threads_in < total_threads)
      {
        /* ожидаем разрешения продолжить работу:
           освободить mutex и ждать сигнала от
           condvar, затем "захватить" mutex опять */
        pthread_cond_wait (&condvar_in, &mutex);
      }
  }

/* в pres находится результат, берем его в каждой
   задаче */
if (pres != a)
  {
    for (i = 0; i < k; i++)
      a[i] = pres[i];
  }

/* увеличить на 1 количество ожидающих выхода задач */
threads_out++;

/* проверяем количество прибывших задач */
if (threads_out >= total_threads)
```

```
    {
      /* текущий поток пришел в очередь последним */
      /* устанавливаем начальное значение для pres */
      pres = 0;
      /* устанавливаем начальное значение
         для threads_in */
      threads_in = 0;

      /* разрешаем остальным продолжать работу */
      pthread_cond_broadcast (&condvar_out);
    }
  else
    {
      /* в очереди ожидания еще есть потоки */

      /* ожидаем, пока в очередь ожидания не придет
         последний поток */
      while (threads_out < total_threads)
        {
          /* ожидаем разрешения продолжить работу:
             освободить mutex и ждать сигнала от
             condvar, затем "захватить" mutex опять */
          pthread_cond_wait (&condvar_out, &mutex);
        }
    }

  /* "освободить" mutex */
  pthread_mutex_unlock (&mutex);
}
```

Функция `reduce_sum` работает аналогично функции `synchronize` из раздела 10.4, но, в отличие от последней, помимо синхронизации задач она вычисляет сумму элементов указанных в качестве аргументов массивов. Результат записывается на место исходных массивов в каждой из задач. Если длина

массива равна нулю, то функции reduce_sum и synchronize совпадают, что использовано в файле reduce_sum.h.

Файл main.c по структуре похож на одноименный файл из раздела 10.4:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "init.h"
#include "get_time.h"
#include "laplace.h"

/* Аргументы для потока */
typedef struct _ARGS
{
  unsigned int n1;       /* число точек по x */
  unsigned int n2;       /* число точек по y */
  double h1;             /* шаг сетки по x */
  double h2;             /* шаг сетки по y */
  unsigned int max_it;   /* максимальное число итераций */
  double prec;           /* величина падения невязки */
  double *f;             /* правая часть */
  double *u;             /* решение */
  double *r;             /* невязка */
  int thread_num;        /* номер задачи */
  int total_threads;     /* всего задач */
} ARGS;

int get_data (char * name, ARGS *parg);

/* Суммарное время работы всех задач */
static long int threads_total_time = 0;
/* Объект типа mutex для синхронизации доступа к
   threads_total_time */
static pthread_mutex_t threads_total_time_mutex
```

```
  = PTHREAD_MUTEX_INITIALIZER;

/* Умножение матрицы на вектор для одной задачи */
void * laplace_solve_threaded (void *pa)
{
  ARGS *p = (ARGS*)pa;
  long int t;

  printf ("Thread %d started\n", p->thread_num);
  t = get_time ();       /* время начала работы */

  laplace_solve (p->n1, p->n2, p->h1, p->h2, p->max_it,
                 p->prec, p->f, p->u, p->r,
                 p->thread_num, p->total_threads);
  t = get_time () - t;  /* время конца работы */

  /* Суммируем времена работы */
  /* "захватить" mutex для работы с threads_total_time *
  pthread_mutex_lock (&threads_total_time_mutex);
  threads_total_time += t;
  /* "освободить" mutex */
  pthread_mutex_unlock (&threads_total_time_mutex);
  printf ("Thread %d finished, time = %ld\n",
          p->thread_num, t);

  return 0;
}


int main ()
{
  /* массив идентификаторов созданных задач */
  pthread_t * threads;
  /* массив аргументов для созданных задач */
  ARGS * args;
```

```
/* астрономическое время работы всего процесса */
long int t_full;
/* аргументы */
ARGS arg;
int i, l;

/* считываем данные из файла */
if (get_data ("a.dat", &arg) < 0)
  {
    fprintf (stderr, "Read error!\n");
    return 1;
  }

printf ("Input threads number: ");
scanf ("%d", &arg.total_threads);
arg.thread_num = 0;

if (!(threads = (pthread_t*)
                malloc (arg.total_threads
                        * sizeof (pthread_t))))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 2;
  }
if (!(args = (ARGS*) malloc (arg.total_threads
                             * sizeof (ARGS))))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 3;
  }

/* Выделение памяти под массивы */
l = (arg.n1 + 1) * (arg.n2 + 1) * sizeof (double);
if (   !(arg.f = (double*) malloc (l))
    || !(arg.u = (double*) malloc (l))
```

```
    || !(arg.r = (double*) malloc (l)))
  {
    fprintf (stderr, "Not enough memory!\n");
    return 3;
  }
l *= 3;
printf ("Allocated %d bytes (%dKb or %dMb) of memory\n",
        l, l >> 10, l >> 20);

/* Инициализация аргументов задач */
for (i = 0; i < arg.total_threads; i++)
  {
    args[i] = arg;
    args[i].thread_num = i;
  }

/* Засекаем астрономическое время начала работы задач*/
t_full = get_full_time ();
/* Запускаем задачи */
for (i = 0; i < arg.total_threads; i++)
  {
    if (pthread_create (threads + i, 0,
                        laplace_solve_threaded,
                        args + i))
      {
        fprintf (stderr, "cannot create thread #%d!\n",
                 i);
        return 10;
      }
  }
/* Ожидаем окончания задач */
for (i = 0; i < arg.total_threads; i++)
  {
    if (pthread_join (threads[i], 0))
      fprintf (stderr, "cannot wait thread #%d!\n", i)
```

```
    }
  t_full = get_full_time () - t_full;
  if (t_full == 0)
    t_full = 1; /* очень быстрый компьютер... */

  /* Здесь можно работать с результатом */
  /* Воспользуемся тем, что мы знаем ответ */
  print_error (arg.n1, arg.n2, arg.h1, arg.h2, arg.u);

  /* Освобождаем память */
  free (arg.r);
  free (arg.u);
  free (arg.f);
  free (args);
  free (threads);

  printf ("Total full time = %ld, \
total threads time = %ld (%ld%%), per thread = %ld\n",
          t_full, threads_total_time,
          (threads_total_time * 100) / t_full,
          threads_total_time / arg.total_threads);
  return 0;
}

/* Считать данные из файла, где находятся:
     число точек по x
     число точек по y
     длина стороны, параллельной оси x
     длина стороны, параллельной оси y
     максимальное число итераций
     величина падения невязки
   Например:
     32 32 1. 1. 1000 1e-6
   Возвращает <0 в случае ошибки.
 */
```

```
int get_data (char * name, ARGS *p)
{
  FILE *in = fopen (name, "r");
  double l1, l2;

  if (!in)
    {
      fprintf (stderr, "Error opening data file %s\n",
               name);
      return -1;
    }

  if (fscanf (in, "%u%u%lf%lf%u%lf", &p->n1, &p->n2,
              &l1, &l2, &p->max_it, &p->prec) != 6)
    {
      fprintf (stderr, "Error reading data file %s\n",
               name);
      return -2;
    }

  /* Вычисляем шаги сетки по направлениям x и y */
  p->h1 = l1 / p->n1;
  p->h2 = l1 / p->n2;

  fclose (in);
  return 0;
}
```

В качестве характеристики полученного приближения выдается норма ошибки, вычисляемая главным процессом.

Заголовочный файл `init.h`:

```
void
print_error (unsigned int n1, unsigned int n2,
             double h1, double h2, double *u);
void init_f (unsigned int n1, unsigned int n2,
```

```
                double h1, double h2, double *f,
                int thread_num, int total_threads);
```

В файле `init.c` находятся функции:

- `solution` — вычисляет точное решение;
- `print_error` — вычисляет норму ошибки полученного приближения, пользуясь тем, что мы знаем точное решение; для уменьшения вычислительной погрешности используется тот же прием, что в функции `get_residual` (см. ниже);
- `right_side` — вычисляет правую часть;
- `init_f` — задает правую часть для каждой из задач.

Файл `init.c`:

```
#include <stdio.h>
#include <math.h>
#include "init.h"
#include "reduce_sum.h"

/* Инициализация правой части задачи.
   В тестах будем рассматривать задачу в единичном
   квадрате [0,1]x[0,1] с ответом
   u(x,y) = x * (1 - x) * y * (1 - y),
   которому соответствует правая часть
   f(x,y) = 2 * x * (1 - x) + 2 * y * (1 - y)
 */

/* Точный ответ */
static double
solution (double x, double y)
{
  return x * (1 - x) * y * (1 - y);
}

/* Вычислить и напечатать L2 норму ошибки */
void
```

```
print_error (
  unsigned int n1,       / *число точек по x */
  unsigned int n2,       / *число точек по y */
  double h1,             / *шаг сетки по x */
  double h2,             / *шаг сетки по y */
  double *u)             / *решение */
{
  double s1, s2, t, error;
  unsigned int i1, i2, addr;

  for (i1 = 0, addr = 0, s1 = 0.; i1 <= n1; i1++)
    {
      for (i2 = 0, s2 = 0.; i2 <= n2; i2++)
        {
          t = u[addr++] - solution (i1 * h1, i2 * h2);
          s2 += t * t;
        }
      s1 += s2;
    }
  error = sqrt (s1 * h1 * h2);
  printf ("Error = %le\n", error);
}

/* Точная правая часть */
static double
right_side (double x, double y)
{
  return 2 * x * (1 - x) + 2 * y * (1 - y);
}

/ *Заполнить правую часть для задачи с
   номером thread_num из общего количества
   total_threads. */
void init_f (
  unsigned int n1,       / *число точек по x */
```

```
  unsigned int n2,      /* число точек по y */
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  double *f,            /* правая часть */
  int thread_num,       /* номер задачи */
  int total_threads)    /* всего задач */
{
  unsigned int first_row, last_row;
  unsigned int i1, i2, addr;

  /* Первая участвующая строка */
  first_row = (n1 + 1) * thread_num;
  first_row /= total_threads;
  /* Последняя участвующая строка */
  last_row = (n1 + 1) * (thread_num + 1);
  last_row = last_row / total_threads - 1;

  for (i1 = first_row, addr = i1 * (n2 + 1);
       i1 <= last_row; i1++)
    for (i2 = 0; i2 <= n2; i2++)
      f[addr++] = right_side (i1 * h1, i2 * h2);

  /* подождать всех */
  synchronize (total_threads);
}
```

Заголовочный файл operators.h:

```
double
get_residual (unsigned int n1, unsigned int n2,
              double h1, double h2, double *r,
              int thread_num, int total_threads);
void
get_operator (unsigned int n1, unsigned int n2,
              double h1, double h2, double *u, double *v,
              int thread_num, int total_threads);
```

```
void
get_preconditioner (unsigned int n1, unsigned int n2,
                    double h1, double h2, double *v,
                    int thread_num, int total_threads);
double
get_optimal_tau (
  unsigned int n1, unsigned int n2,
  double h1, double h2, double *f, double *u, double *r,
  int thread_num, int total_threads);
```

В файле `operators.c` находятся функции:

- `get_residual` — возвращает L_2 норму невязки; основные особенности функции:
  - сумма квадратов элементов вычисляется в каждой задаче для своего участка массива, затем с помощью `reduce_sum` получается сумма квадратов всех элементов массива, которая используется для вычисления результата, возвращаемого функцией в каждой задаче;
  - граничные точки не учитываются;
  - для уменьшения вычислительной погрешности, связанной с суммированием чисел, сильно различающихся по порядку, вычисляются суммы квадратов элементов по строкам, которые затем прибавляются к окончательному результату;
- `get_operator` — вычисляет произведение матрицы A системы (10.6) (т. е. (10.5)) на вектор u: $v = Au$; оператор вычисляется согласно формулам (10.7) в каждой задаче для своего участка массива, синхронность работы всех задач обеспечивается с помощью `synchronize`;
- `get_preconditioner` — вычисляет произведение матрицы B^{-1} в (10.8) на вектор u: $v = B^{-1}u$; оператор вычисляется согласно формулам (10.9) для матрицы B в каждой задаче для своего участка массива, синхронность работы всех задач обеспечивается с помощью `synchronize`;

- `get_optimal_tau` — возвращает оптимальное значение итерационного параметра τ_k в (10.8) согласно формулам (10.10); основные особенности функции:
  - скалярные произведения (f, r), (Ar, u), (Ar, r) вычисляются в каждой задаче для своих участков массивов, затем с помощью `reduce_sum` получаются соответствующие скалярные произведения для всех элементов массива, которые используются для вычисления результата, возвращаемого функцией в каждой задаче;
  - значение Ar вычисляется в точке и сразу используется при вычислении скалярных произведений (Ar, u) и (Ar, r);
  - для уменьшения вычислительной погрешности используется тот же прием, что в `get_residual`.

Файл `operators.c`:

```
#include <stdio.h>
#include <math.h>
#include "operators.h"
#include "reduce_sum.h"

/* Вычислить L2 норму невязки для задачи с
   номером thread_num из общего количества total_threads.
   Граничные узлы не входят. */
double
get_residual (
  unsigned int n1,      /* число точек по x */
  unsigned int n2,      /* число точек по y */
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  double *r,            /* невязка */
  int thread_num,       /* номер задачи */
  int total_threads)    /* всего задач */
{
  unsigned int first_row, last_row;
  unsigned int i1, i2, addr;
```

```
double s1, s2, t;

/* Первая участвующая строка */
first_row = (n1 + 1) * thread_num;
first_row /= total_threads;
/* Последняя участвующая строка */
last_row = (n1 + 1) * (thread_num + 1);
last_row = last_row / total_threads - 1;

/* В силу нулевых краевых условий на границе
   невязка = 0 */
if (first_row == 0)
  first_row = 1;
if (last_row == n1)
  last_row = n1 - 1;

for (i1 = first_row, addr = i1 * (n2 + 1), s1 = 0.;
     i1 <= last_row; i1++)
  {
    /* в первом элементе невязка = 0 в силу
       краевых условий */
    addr++;
    for (i2 = 1, s2 = 0.; i2 < n2; i2++)
      {
        t = r[addr++];
        s2 += t * t;
      }
    s1 += s2;
    /* в последнем элементе невязка = 0 в силу
       краевых условий */
    addr++;
  }

/* Вычислить сумму по всем задачам */
reduce_sum (total_threads, &s1, 1);
```

```
  return sqrt (s1 * h1 * h2);
}

/* Вычислить v = Au, где A - оператор Лапласа для задачи
   с номером thread_num из общего количества
   total_threads. */
void
get_operator (
  unsigned int n1,       /* число точек по x */
  unsigned int n2,       /* число точек по y */
  double h1,             /* шаг сетки по x */
  double h2,             /* шаг сетки по y */
  double *u,             /* решение */
  double *v,             /* результат */
  int thread_num,        /* номер задачи */
  int total_threads)     /* всего задач */
{
  unsigned int first_row, last_row;
  unsigned int i1, i2, addr;
  double hh1 = 1. / (h1 * h1), hh2 = 1. / (h2 * h2);

  /* Первая участвующая строка */
  first_row = (n1 + 1) * thread_num;
  first_row /= total_threads;
  /* Последняя участвующая строка */
  last_row = (n1 + 1) * (thread_num + 1);
  last_row = last_row / total_threads - 1;

  if (first_row == 0)
    {
      /* нулевые краевые условия */
      for (i2 = 0, addr = 0; i2 <= n2; i2++)
        v[addr++] = 0.;
      first_row = 1;
```

```
    }
  if (last_row == n1)
    {
      /* нулевые краевые условия */
      for (i2 = 0, addr = n1 * (n2 + 1); i2 <= n2; i2++)
        v[addr++] = 0.;
      last_row = n1 - 1;
    }

  for (i1 = first_row, addr = i1 * (n2 + 1);
       i1 <= last_row; i1++)
    {
      /* первый элемент = 0 в силу краевых условий */
      v[addr++] = 0.;
      for (i2 = 1; i2 < n2; i2++, addr++)
        {
          /* v(i1,i2)
           =- (u(i1-1,i2)-2u(i1,i2)+u(i1+1,i2))/(h1*h1)
            - (u(i1,i2-1)-2u(i1,i2)+u(i1,i2-1))/(h2*h2)*/
          v[addr] = - (u[addr - (n2 + 1)] - 2 * u[addr]
                       + u[addr + (n2 + 1)]) * hh1
                    - (u[addr - 1] - 2 * u[addr]
                       + u[addr + 1]) * hh2;
        }
      /* последний элемент = 0 в силу краевых условий */
      v[addr++] = 0.;
    }
  /* подождать всех */
  synchronize (total_threads);
}

/* Вычислить v = B^{-1}v, где B - предобуславливатель для
   оператора Лапласа для задачи с номером thread_num
   из общего количества total_threads.
   В качестве B используется диагональ матрицы A. */
```

```
void
get_preconditioner (
  unsigned int n1,      /* число точек по x */
  unsigned int n2,      /* число точек по y */
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  double *v,            /* аргумент/результат */
  int thread_num,       /* номер задачи */
  int total_threads)    /* всего задач */
{
  unsigned int first_row, last_row;
  unsigned int i1, i2, addr;
  double hh1 = 1. / (h1 * h1), hh2 = 1. / (h2 * h2);
  double w = 1. / (2 * hh1 + 2 * hh2);

  /* Первая участвующая строка */
  first_row = (n1 + 1) * thread_num;
  first_row /= total_threads;
  /* Последняя участвующая строка */
  last_row = (n1 + 1) * (thread_num + 1);
  last_row = last_row / total_threads - 1;

  if (first_row == 0)
    {
      /* нулевые краевые условия */
      for (i2 = 0, addr = 0; i2 <= n2; i2++)
        v[addr++] = 0.;
      first_row = 1;
    }
  if (last_row == n1)
    {
      /* нулевые краевые условия */
      for (i2 = 0, addr = n1 * (n2 + 1); i2 <= n2; i2++)
        v[addr++] = 0.;
      last_row = n1 - 1;
```

```
    }

  for (i1 = first_row, addr = i1 * (n2 + 1);
       i1 <= last_row; i1++)
    {
      /* первый элемент = 0 в силу краевых условий */
      v[addr++] = 0.;
      for (i2 = 1; i2 < n2; i2++)
        {
          /* v(i1,i2) = v(i1,i2)/(2/(h1*h1)+2/(h2*h2)) */
          v[addr++] *= w;
        }
      /* последний элемент = 0 в силу краевых условий */
      v[addr++] = 0.;
    }
  /* подождать всех */
  synchronize (total_threads);
}

/* Вычислить ((f, r) - (Ar, u)) / (Ar, r)
   где A - оператор Лапласа для задачи с
   номером thread_num из общего количества total_threads.
   Функции u, r удовлетворяют нулевым краевым условиям */
double
get_optimal_tau (
  unsigned int n1,      /* число точек по x */
  unsigned int n2,      /* число точек по y */
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  double *f,            /* правая часть */
  double *u,            /* решение */
  double *r,            /* B^{-1}(невязка) */
  int thread_num,       /* номер задачи */
  int total_threads)    /* всего задач */
{
```

```
unsigned int first_row, last_row;
unsigned int i1, i2, addr;
double hh1 = 1. / (h1 * h1), hh2 = 1. / (h2 * h2);
double s1_fr, s2_fr;  /* суммы для (f, r)    */
double s1_Aru, s2_Aru;/* суммы для (Ar, u)   */
double s1_Arr, s2_Arr;/* суммы для (Ar, r)   */
double Ar;            /* значение Ar в точке */
double a[3];

/* Первая участвующая строка */
first_row = (n1 + 1) * thread_num;
first_row /= total_threads;
/* Последняя участвующая строка */
last_row = (n1 + 1) * (thread_num + 1);
last_row = last_row / total_threads - 1;

/* В силу нулевых краевых условий на границе,
   слагаемые, соответствующие граничным узлам = 0 */
if (first_row == 0)
  first_row = 1;
if (last_row == n1)
  last_row = n1 - 1;

for (i1 = first_row, addr = i1 * (n2 + 1),
     s1_fr = 0., s1_Aru = 0., s1_Arr = 0.;
     i1 <= last_row; i1++)
  {
    /* в первом элементе слагаемые = 0 в силу
       краевых условий */
    addr++;
    for (i2 = 1, s2_fr = 0., s2_Aru = 0., s2_Arr = 0.;
         i2 < n2; i2++, addr++)
      {
        /* Вычисляем (f, r) */
        s2_fr += f[addr] * r[addr];
```

```
            /* Вычисляем Ar */
            Ar = - (r[addr - (n2 + 1)] - 2 * r[addr]
                    + r[addr + (n2 + 1)]) * hh1
                 - (r[addr - 1] - 2 * r[addr]
                    + r[addr + 1]) * hh2;
            /* Вычисляем (Ar, u) */
            s2_Aru += Ar * u[addr];
            /* Вычисляем (Ar, r) */
            s2_Arr += Ar * r[addr];
          }
        s1_fr  += s2_fr;
        s1_Aru += s2_Aru;
        s1_Arr += s2_Arr;
        /* в последнем элементе слагаемые = 0 в силу
           краевых условий */
        addr++;
      }
    a[0] = s1_fr;
    a[1] = s1_Aru;
    a[2] = s1_Arr;

    /* Вычислить суммы по всем задачам */
    reduce_sum (total_threads, a, 3);

    /* Вычислить ответ */
    return (a[0] - a[1]) / a[2];
}
```

Заголовочный файл laplace.h:

```
int laplace_solve (unsigned int n1, unsigned int n2,
  double h1, double h2, unsigned int max_it, double prec,
  double *f, double *u, double *r, int thread_num,
  int total_threads);
```

В файле laplace.c находится функция laplace_solve, осуществляющая итерационный процесс (10.8), (10.10) с матрицами (10.7) и (10.9) по описанным выше расчетным формулам. Файл laplace.c:

```
#include <stdio.h>
#include <math.h>
#include "init.h"
#include "operators.h"
#include "reduce_sum.h"
#include "laplace.h"

/* Решить задачу для задачи с
   номером thread_num из общего количества total_threads.
   Возвращает количество потребовавшихся итераций. */
int laplace_solve (
  unsigned int n1,       /* число точек по x */
  unsigned int n2,       /* число точек по y */
  double h1,             /* шаг сетки по x */
  double h2,             /* шаг сетки по y */
  unsigned int max_it,   /* максимальное число итераций */
  double prec,           /* величина падения невязки */
  double *f,             /* правая часть */
  double *u,             /* решение */
  double *r,             /* невязка */
  int thread_num,        /* номер задачи */
  int total_threads)     /* всего задач */
{
  unsigned int first_row, last_row;
  unsigned int first_addr, last_addr;
  unsigned int addr;
  /* Норма невязки на первом шаге */
  double norm_residual_1;
  /* Норма невязки на очередном шаге */
  double norm_residual;
```

```
/* Норма невязки на предыдущем шаге */
double norm_residual_prev;
unsigned int it;      /* Номер итерации */
double tau;           /* Итерационный параметр */

/* Первая участвующая строка */
first_row = (n1 + 1) * thread_num;
first_row /= total_threads;
/* Последняя участвующая строка */
last_row = (n1 + 1) * (thread_num + 1);
last_row = last_row / total_threads - 1;
/* Соответствующие адреса */
first_addr = first_row * (n2 + 1);
last_addr  = last_row * (n2 + 1) + n2;

/* Инициализируем правую часть */
init_f (n1, n2, h1, h2, f, thread_num, total_threads);

/* Начальное приближение = 0 */
for (addr = first_addr; addr <= last_addr; addr++)
  u[addr] = 0.;

/* Невязка при нулевом начальном приближении = f */
norm_residual_1 = norm_residual = norm_residual_prev
  = get_residual (n1, n2, h1, h2, f, thread_num,
                  total_threads);

if (thread_num == 0)
  printf ("Residual = %le\n", norm_residual_1);

/* Итерационный процесс */
for (it = 1; it < max_it; it++)
  {
    /* Шаг итерационного процесса */
```

```
      /* Вычисляем r = Au */
      get_operator (n1, n2, h1, h2, u, r, thread_num,
                    total_threads);

      /* Вычисляем невязку r = f - r = f - Au */
      for (addr = first_addr; addr <= last_addr; addr++)
        r[addr] = f[addr] - r[addr];
      /* подождать всех */
      synchronize (total_threads);

      /* Вычисляем норму невязки */
      norm_residual = get_residual (n1, n2, h1, h2, r,
                                    thread_num,
                                    total_threads);
      if (thread_num == 0)
        printf ("It # =%2.2d, residual=%11.4e, \
convergence=%6.3f, average convergence=%6.3f\n",
            it, norm_residual,
            norm_residual / norm_residual_prev,
            pow (norm_residual / norm_residual_1, 1./it));

      /* Проверяем условие окончания процесса */
      if (norm_residual < norm_residual_1 * prec)
        break;

      /* Обращение предобуславливателя r = B^{-1} r */
      get_preconditioner (n1, n2, h1, h2, r, thread_num,
                          total_threads);

      /* Вычисление итерационного параметра
         tau = ((b, r) - (Ar, u)) / (Ar, r) */
      tau = get_optimal_tau (n1, n2, h1, h2, f, u, r,
                             thread_num, total_threads);

      /* Построение очередного приближения u+=tau * r */
```

```
      for (addr = first_addr; addr <= last_addr; addr++)
        u[addr] += tau * r[addr];
      /* подождать всех */
      synchronize (total_threads);

      norm_residual_prev = norm_residual;
    }

  return it;
}
```

11

Интерфейс MPI (Message Passing Interface)

В этой главе мы рассмотрим интерфейс MPI (Message Passing Interface), ставший de-facto стандартом для программирования систем с распределенной памятью. MPI представляет собой набор утилит и библиотечных функций (для языков C/C++, FORTRAN), позволяющих создавать и запускать приложения, работающие на параллельных вычислительных установках самой различной природы. Появившись в 1990-х годах как унифицированный подход к программированию для систем с распределенной памятью, MPI приобрел большую популярность и теперь широко используется также в системах с общей памятью и разнообразных смешанных вычислительных установках.

11.1. Общее устройство MPI-программы

Имена всех функций, типов данных, констант и т. д., относящихся к MPI-библиотеке, начинаются с префикса `MPI_` и описаны в заголовочном файле `mpi.h`. Все функции (за исключением `MPI_Wtime` и `MPI_Wtick`) имеют тип возвращаемого значения `int` и возвращают код ошибки или `MPI_SUCCESS` в случае успеха. Однако в случае ошибки перед возвратом из вызвавшей ее функции вызывается стандартный обработчик ошибки, который аварийно терминирует программу. Поэтому проверять возвращаемое значение не имеет смысла. Стандартный обработчик ошибки можно заменить (с помощью функции `MPI_Errhandler_set`),

но стандарт MPI не гарантирует, что программа может продолжать работать после ошибки. Так что это тоже обычно не имеет смысла.

До первого вызова любой MPI-функции необходимо вызвать функцию `MPI_Init`. Ее прототип:

```
int MPI_Init (int *argc, char ***argv);
```

где `argc`, `argv` — указатели на число аргументов программы и на вектор аргументов соответственно (это адреса аргументов функции `main` программы). Многие реализации MPI требуют, чтобы процесс до вызова `MPI_Init` не делал ничего, что могло бы изменить его состояние, например открытие или чтение/запись файлов, включая стандартные ввод и вывод.

После окончания работы с MPI-функциями необходимо вызвать функцию `MPI_Finalize`. Ее прототип:

```
int MPI_Finalize (void);
```

Количество работающих параллельных процессов после вызова этой функции не определено, поэтому после ее вызова лучше всего сразу же закончить работу программы.

Общий вид MPI-программы:

```
#include "mpi.h"
/* Другие описания */

int main (int argc, char * argv[])
{
  /* Описания локальных переменных */

  MPI_Init (&argc, &argv);

  /* Тело программы */

  MPI_Finalize ();
  return 0;
}
```

11.2. Сообщения

Параллельно работающие MPI-процессы обмениваются между собой информацией и обеспечивают взаимную синхронизацию посредством **сообщений**.

Различают обмен сообщениями:

- **попарный (point-to-point)** — сообщение посылается одним процессом другому, в обмене участвуют только два процесса;
- **коллективный** — сообщение посылается процессом всем процессам из его группы (коммуникатора, см. ниже) и получается всеми процессами из его группы (коммуникатора). Коллективный обмен может быть представлен как последовательность попарных обменов, однако специализированные MPI-функции для коллективных обменов учитывают специфику построения конкретной вычислительной установки и могут выполняться значительно быстрее соответствующей последовательности попарных обменов.

Различают обмен сообщениями:

- **синхронный** — процесс-отправитель сообщения переходит в состояние ожидания, пока процесс-получатель не будет готов взять сообщение, а процесс-получатель сообщения переходит в состояние ожидания, пока процесс-отправитель не будет готов послать сообщение;
- **асинхронный** — процесс-отправитель сообщения не ждет готовности процесса-получателя, сообщение копируется подсистемой MPI во внутренний буфер, и отправитель продолжает работу.

В первых реализациях MPI все обмены были синхронными. Для повышения эффективности работы (за счет параллельности работы процесса и пересылки сообщения) последние реализации MPI стараются сделать как можно больше обменов асинхронными. Например, если размер сообщения небольшой, то его обычно буферизуют и устраивают асинхронный обмен. Длительное время все коллективные обмены были синхронными.

Отметим, что поведение некачественной программы может быть разным при использовании синхронного или асинхронного режима. Например, если процесс A посылает сообщение процессу B, который не вызывает функцию получения сообщения, то это приводит к «зависанию» процесса A в синхронном режиме. В асинхронном режиме такая программа будет работать.

Основные составляющие сообщения:

1. **Блок данных сообщения** — представляется типом `void*`.
2. **Информация о данных сообщения**:
   (a) **тип данных** — представляется типом `MPI_Datatype`; соответствие MPI-типов данных и типов языка C приведено в табл. 11.1.
   (b) **количество данных** — количество единиц данного типа в блоке сообщения. Представляется типом `int`. (Причина появления этого поля достаточно очевидна: выгоднее за один раз передать большое сообщение с 10-ю элементами, чем посылать 10 сообщений с одним элементом.)

Таблица 11.1. Соответствие типов данных MPI и типов языка C

Тип MPI	Тип C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char
MPI_PACKED	

3. **Информация о получателе и отправителе сообщения**:

   (a) **коммуникатор** — идентификатор группы запущенных программой параллельных процессов, которые могут обмениваться между собой сообщениями. Представляется типом `MPI_Comm`. Как получатель, так и отправитель должны принадлежать указанной группе. Процесс может принадлежать одновременно многим группам, все запущенные процессы принадлежат группе с идентификатором `MPI_COMM_WORLD`.

   (b) **ранг получателя** — номер процесса–получателя в указанной группе (коммуникаторе). Представляется типом `int`. Это поле отсутствует при коллективных обменах сообщениями.

   (c) **ранг отправителя** — номер процесса–отправителя в указанной группе (коммуникаторе). Представляется типом `int`. Получатель может указать, что он будет принимать только сообщения от отправителя с определенным рангом, или использовать константу `MPI_ANY_SOURCE`, позволяющую принимать сообщения от любого отправителя в группе с указанным идентификатором. Это поле отсутствует при коллективных обменах сообщениями.

4. **Тег сообщения** — произвольное число типа `int` (стандарт гарантирует возможность использования чисел от 0 до 2^{15}), приписываемое отправителем сообщению. Получатель может указать, что он будет принимать только сообщения, имеющие определенный тег, или использовать константу `MPI_ANY_TAG`, позволяющую принимать сообщения с любым тегом. Это поле отсутствует при коллективных обменах сообщениями, поскольку все процессы обмениваются одинаковыми по структуре данными и в первых версиях MPI все коллективные обмены были синхронными.

11.3. Коммуникаторы

Напомним, **коммуникатор** — это объект типа `MPI_Comm`, представляющий собой идентификатор группы запущенных программой параллельных процессов, которые могут обмениваться сообщениями. Коммуникатор является аргументом всех функций, осуществляющих обмен сообщениями. Программа может разделять исходную группу всех запущенных процессов, идентифицируемую коммуникатором `MPI_COMM_WORLD`, на подгруппы для:

- организации коллективных обменов внутри этих подгрупп,
- изоляции одних обменов от других (так часто поступают параллельные библиотечные функции, чтобы их сообщения не пересекались с сообщениями вызвавшего их процесса),
- учета топологии распределенной вычислительной установки (например, часто распределенный кластер можно представить в виде пространственной решетки, составленной из скоростных линий связи, в узлах которой находятся рабочие станции; скорости обмена данными между узлами тут различны и это можно учесть введением соответствующих коммуникаторов).

Получить количество процессов в группе (коммуникаторе) можно с помощью функции `MPI_Comm_size`. Ее прототип:

```
int MPI_Comm_size (MPI_Comm comm, int *size);
```

где `comm` — коммуникатор (входной параметр), `size` — указатель на результат. Например, общее количество запущенных процессов можно получить следующим образом:

```
int p;
MPI_Comm_size (MPI_COMM_WORLD, &p);
```

Получить ранг (номер) процесса в группе (коммуникаторе) можно с помощью функции `MPI_Comm_rank`. Ее прототип:

```
int MPI_Comm_rank (MPI_Comm comm, int *rank);
```

где comm — коммуникатор (входной параметр), rank — указатель на результат. Например, номер текущего процесса среди всех запущенных можно получить следующим образом:

```
int my_rank;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
```

11.4. Попарный обмен сообщениями

Послать сообщение можно с помощью функции MPI_Send. Ее прототип:

```
int MPI_Send (void *buf, int count,
  MPI_Datatype datatype, int dest, int tag,
  MPI_Comm comm);
```

где входные параметры:

- buf — адрес буфера с данными,
- count — число элементов (типа datatype) в буфере,
- datatype — тип данных каждого из элементов буфера,
- dest — ранг (номер) получателя в коммуникаторе (группе) comm,
- tag — тег сообщения,
- comm — коммуникатор.

Эта функция может перевести текущий процесс в состояние ожидания, пока получатель с номером dest в группе comm не примет сообщение с тегом tag (если используется синхронный режим обмена).

Получить сообщение можно с помощью функции MPI_Recv. Ее прототип:

```
int MPI_Recv (void *buf, int count,
  MPI_Datatype datatype, int source, int tag,
  MPI_Comm comm, MPI_Status *status);
```

где входные параметры:

- buf — адрес буфера с данными, где следует разместить полученное сообщение,

- `count` — максимальное число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `source` — ранг (номер) отправителя в коммуникаторе (группе) `comm` (может быть `MPI_ANY_SOURCE`),
- `tag` — тег сообщения (может быть `MPI_ANY_TAG`),
- `comm` — коммуникатор,

и выходные параметры (результаты):

- `buf` — полученное сообщение,
- `status` — информация о полученном сообщении (см. ниже).

Эта функция переводит текущий процесс в состояние ожидания, пока отправитель с номером `source` (или любым номером, если в качестве ранга используется константа `MPI_ANY_SOURCE`) в группе `comm` не отправит сообщение с тегом `tag` (или любым тегом, если в качестве тега используется константа `MPI_ANY_TAG`).

При попарных обменах сообщениями получатель сообщения может получить через переменную `status` типа `MPI_Status *` информацию о полученном сообщении. Структура данных `MPI_Status` включает в себя следующие поля:

- `MPI_SOURCE` — реальный ранг отправителя (может потребоваться, если в качестве ранга отправителя использована константа `MPI_ANY_SOURCE`),
- `MPI_TAG` — реальный тег сообщения (может потребоваться, если в качестве тега сообщения использована константа `MPI_ANY_TAG`),
- `MPI_ERROR` — код ошибки,
- дополнительные служебные поля, зависящие от реализации MPI.

Размер полученного сообщения непосредственно в структуре данных `MPI_Status` не хранится, но может быть получен из нее с помощью функции `MPI_Get_count`. Ее прототип:

```
int MPI_Get_count (MPI_Status *status,
  MPI_Datatype datatype, int *count);
```

где входные параметры:

- `status` — информация о полученном сообщении,
- `datatype` — тип данных, в единицах которого требуется получить размер сообщения,

и выходной параметр (результат):

- `count` — количество полученных элементов в единицах типа `datatype` или константа `MPI_UNDEFINED`, если длина данных сообщения не делится нацело на размер типа `datatype`.

Заметим, что при коллективных обменах получатель информацию о сообщении не получает, поскольку процессы обмениваются одинаковыми по структуре данными.

Еще раз отметим, что поведение некачественной программы может быть разным при использовании синхронного или асинхронного режима отправки сообщений. Например, если процесс A последовательно посылает процессу B два сообщения с тегами 0 и 1, а процесс B последовательно вызывает функцию получения сообщения с тегами 1 и 0, то это приводит к «зависанию» обоих процессов в синхронном режиме. В асинхронном режиме такая программа будет работать.

11.5. Операции ввода–вывода в MPI-программах

Рассмотрим вначале стандартный ввод–вывод (на экран). Здесь ситуация зависит от конкретной реализации MPI:

- Существуют реализации MPI, в которых всем работающим процессам разрешено осуществлять ввод–вывод на экран. В этом случае на экране будет наблюдаться перемешанный вывод от всех процессов, в котором часто трудно разобраться.
- Для предотвращения перемешивания вывода разных процессов некоторые реализации MPI (и таких большинство) разрешают осуществлять ввод–вывод на экран только процессу с номером 0 в группе `MPI_COMM_WORLD`. В этом случае остальные процессы посылают этому процессу запросы на производство своего ввода-вывода на экран в качестве сообщений.

Этим обеспечивается последовательный вывод на экран (без перемешивания).

- Существуют реализации MPI, в которых всем работающим процессам запрещено осуществлять ввод–вывод на экран. При выводе на экран создается файл, в котором содержится все выведенное, а при вводе с клавиатуры фиксируется ошибка ввода. Обычно такие реализации работают на распределенных вычислительных установках, где запуск программы на исполнение осуществляется с выделенного компьютера, не входящего в состав узлов, на которых собственно исполняется MPI-программа. В таких реализациях весь ввод–вывод осуществляется через файлы.

Поэтому для написания переносимой между различными реализациями MPI-программы следует придерживаться предположения о том, что программа не может осуществлять стандартный ввод–вывод и должна работать только с файлами (см. ниже). Единственным исключением является вывод на экран, который появится либо на экране, либо в созданном системой файле.

На файловый ввод-вывод какие-либо ограничения формально отсутствуют. Однако при одновременной записи данных в файл несколькими процессами может происходить их перемешивание и даже потеря. Одновременное чтение данных из файла несколькими процессами на распределенной вычислительной установке может приводить к блокировке всех процессов, кроме одного, и их последовательному выполнению. Это связано с работой сетевой файловой системы (NFS, network file system). Поэтому для написания переносимой между различными реализациями MPI-программы следует придерживаться предположения о том, что осуществлять файловый ввод–вывод можно только процессу с номером 0 в группе `MPI_COMM_WORLD`. Остальные процессы получают или передают ему свои данные посредством сообщений. Естественно, это не относится к ситуации, когда каждый из процессов осуществляет ввод–вывод в свой собственный файл.

11.6. Пример простейшей MPI-программы

Рассмотрим в качестве примера программу, выводящую на экран сообщение «Hello» от каждого из процессов.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

/* Длина буфера сообщений */
#define BUF_LEN 256

int
main (int argc, char *argv[])
{
  int my_rank;          /* ранг текущего процесса */
  int p;                /* общее число процессов */
  int source;           /* ранг отправителя */
  int dest;             /* ранг получателя */
  int tag = 0;          /* тег сообщений */
  char message[BUF_LEN];/* буфер для сообщения */
  MPI_Status status;    /* информация о полученном
                           сообщении */

  /* Начать работу с MPI */
  MPI_Init (&argc, &argv);

  /* Получить номер текущего процесса в группе всех
     процессов  */
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  /* Получить общее количество запущенных процессов */
  MPI_Comm_size (MPI_COMM_WORLD, &p);

  /* Посылаем сообщения процессу 0, который их выводит
     на экран */
```

```
  if (my_rank != 0)
    {
      /* Создаем сообщение */
      sprintf (message, "Hello from process %d!",
               my_rank);
      /* Отправляем его процессу 0 */
      dest = 0;
      MPI_Send (message, strlen (message) + 1, MPI_CHAR,
                dest, tag, MPI_COMM_WORLD);
    }
  else
    {
      /* В процессе 0: получаем сообщение от процесса
         1,...,p-1 и выводим его на экран */
      for (source = 1; source < p; source++)
        {
          MPI_Recv (message, BUF_LEN, MPI_CHAR, source,
                    tag, MPI_COMM_WORLD, &status);
          printf ("%s\n", message);
        }
    }

  /* Заканчиваем работу с MPI */
  MPI_Finalize ();
  return 0;
}
```

11.7. Дополнительные функции для попарного обмена сообщениями

Для удобства программиста стандарт MPI предоставляет ряд дополнительных функций для попарного обмена сообщениями. Любая программа может быть написана без их использования, но для многих вычислительных установок они могут значительно увеличить производительность.

Если программе важно, чтобы был использован именно синхронный способ посылки сообщений, то вместо функции MPI_Send можно использовать функцию MPI_Ssend, имеющую те же аргументы и возвращаемое значение (дополнительная буква s в имени — от synchronous).

При попарных обменах между процессами очень часто требуется именно обменяться данными, т. е. послать и получить разные сообщения. При этом программный код для этих процессов получается несимметричным: для одного из них надо вначале использовать MPI_Send, а затем MPI_Recv, для второго — эти же функции, но в обратной последовательности. Если же оба процесса вызывают эти функции в одинаковом порядке, то в случае синхронного обмена сообщениями они оба «зависают». Для этой ситуации стандарт MPI предоставляет функцию MPI_Sendrecv, являющуюся как-бы гибридом MPI_Send и MPI_Recv. Ее прототип:

```
int MPI_Sendrecv (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, int dest, int sendtag,
  void *recvbuf, int recvcount,
  MPI_Datatype recvtype, int source, int recvtag,
  MPI_Comm comm, MPI_Status *status);
```

где входные параметры:

- sendbuf — адрес буфера с посылаемыми данными,
- sendcount — число элементов (типа sendtype) в буфере sendbuf,
- sendtype — тип данных каждого из элементов буфера sendbuf,
- dest — ранг (номер) получателя в коммуникаторе (группе) comm,
- sendtag — тег посылаемого сообщения,
- recvbuf — адрес буфера с данными, где следует разместить полученное сообщение,
- recvcount — максимальное число элементов (типа recvtype) в буфере recvbuf,

- `recvtype` — тип данных каждого из элементов буфера `recvbuf`,
- `source` — ранг (номер) отправителя в коммуникаторе (группе) `comm` (может быть `MPI_ANY_SOURCE`),
- `recvtag` — тег получаемого сообщения (может быть константой `MPI_ANY_TAG`),
- `comm` — коммуникатор,

и выходные параметры (результаты):

- `recvbuf` — полученное сообщение,
- `status` — информация о полученном сообщении.

Эта функция может перевести текущий процесс в состояние ожидания, пока получатель с номером `dest` в группе `comm` не примет сообщение с тегом `sendtag` (если используется синхронный режим обмена) или пока отправитель с номером `source` (или любым номером, если в качестве ранга используется константа `MPI_ANY_SOURCE`) в группе `comm` не отправит сообщение с тегом `recvtag` (или любым тегом, если в качестве тега используется константа `MPI_ANY_TAG`). Посылать сообщения для приема этой функцией можно с помощью любой функции для попарных обменов, например, `MPI_Send` или ее самой, принимать сообщения от этой функции можно посредством любой функции для попарных обменов, например, `MPI_Recv` или ее самой.

Стандарт MPI запрещает использовать одинаковые указатели для любых двух аргументов любой MPI-функции, если хотя бы один из них является выходным параметром. Для функции `MPI_Sendrecv` это означает, что нельзя использовать одинаковые значения для входных и выходных буферов. Если программе это требуется, то надо применять функцию `MPI_Sendrecv_replace`. Ее прототип:

```
int MPI_Sendrecv_replace (void *buf, int count,
  MPI_Datatype datatype, int dest, int sendtag,
  int source, int recvtag,
  MPI_Comm comm, MPI_Status *status);
```

где входные параметры:

- `buf` — адрес буфера с посылаемыми данными (также является выходным параметром),
- `count` — число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `dest` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `sendtag` — тег посылаемого сообщения,
- `source` — ранг (номер) отправителя в коммуникаторе (группе) `comm` (может быть `MPI_ANY_SOURCE`),
- `recvtag` — тег получаемого сообщения (может быть константой `MPI_ANY_TAG`),
- `comm` — коммуникатор,

и выходные параметры (результаты):

- `buf` — полученное сообщение (также является входным параметром),
- `status` — информация о полученном сообщении.

Работа этой функции аналогична `MPI_Sendrecv`, за исключением того, что полученные данные замещают посланные.

В некоторых вычислительных установках для обмена сообщениями существует специальный коммуникационный процессор, способный работать параллельно с основным. Следовательно, передача сообщений и вычислительная работа могут идти параллельно, если для проведения дальнейших вычислений данные сообщения не нужны (получателю) и не будут модифицироваться (отправителем). Послать сообщение без блокирования процесса можно с помощью функции `MPI_Isend` (дополнительная буква `i` в имени — от immediate). Ее прототип:

```
int MPI_Isend (void *buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm, MPI_Request *request);
```

где входные параметры:

- `buf` — адрес буфера с данными,
- `count` — число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `dest` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `tag` — тег сообщения,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `request` — идентификатор запроса на обслуживание сообщения; имеет тип `MPI_Request`, не доступный пользователю (указатель на объект этого типа возвращается процессу и может быть использован в качестве аргумента для других функций).

Принимать сообщения от этой функции можно с помощью любой функции для попарных обменов, например, `MPI_Recv`.

Получить сообщение без блокирования процесса можно с помощью функции `MPI_Irecv` (дополнительная буква `i` в имени — от immediate). Ее прототип:

```
int MPI_Irecv (void *buf, int count,
  MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, MPI_Request *request);
```

где входные параметры:

- `buf` — адрес буфера с данными, где следует разместить полученное сообщение,
- `count` — максимальное число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `source` — ранг (номер) отправителя в коммуникаторе (группе) `comm` (может быть `MPI_ANY_SOURCE`),
- `tag` — тег сообщения (может быть `MPI_ANY_TAG`),
- `comm` — коммуникатор,

и выходные параметры (результаты):

- `buf` – полученное сообщение,
- `request` — идентификатор запроса на обслуживания сообщения.

Посылать сообщения для приема этой функцией можно с помощью любой функции для попарных обменов, например, `MPI_Send`.

Узнать, доставлено ли сообщение (как для `MPI_Isend`, так и для `MPI_Irecv`), можно с помощью функции `MPI_Test`. Ее прототип:

```
int MPI_Test (MPI_Request *request, int *flag,
  MPI_Status *status);
```

где входной параметр:

- `request` – идентификатор запроса на обслуживание сообщения (также является выходным параметром: если обслуживание завершено, то он становится равным специальному значению `MPI_REQUEST_NULL`).

и выходные параметры (результаты):

- `flag` — признак окончания обслуживания, значение `*flag` становится истинным, если запрос обработан,
- `status` — если запрос обработан, то содержит информацию о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции `MPI_Recv`).

Функция `MPI_Test` не блокирует вызвавший ее процесс, а только устанавливает свои аргументы указанным выше образом. Если требуется проверить доставку одного из нескольких сообщений, то можно использовать функцию `MPI_Testany`. Ее прототип:

```
int MPI_Testany (int count,
  MPI_Request array_of_requests[], int *index,
  int *flag, MPI_Status *status);
```

где входные параметры

- `count` — количество запросов в массиве `array_of_requests`,
- `array_of_requests` — массив идентификаторов запросов на обслуживание сообщений (также является выходным параметром: если обслуживание сообщения с номером `*index` в массиве `array_of_requests` завершено, то соответствующий элемент `array_of_requests[*index]` становится равным `MPI_REQUEST_NULL`)

и выходные параметры (результаты):

- `flag` — признак окончания обслуживания, значение `*flag` становится истинным, если один из запросов обработан,
- `index` — если один из запросов обработан, то содержит номер обработанного сообщения в массиве `array_of_requests`,
- `status` — если один из запросов обработан, то содержит информацию о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции `MPI_Recv`).

Дождаться доставки сообщения (как для `MPI_Isend`, так и для `MPI_Irecv`) можно с помощью функции `MPI_Wait`. Ее прототип:

```
int MPI_Wait (MPI_Request *request,
  MPI_Status *status);
```

где входной параметр

- `request` — идентификатор запроса на обслуживание сообщения (также является выходным параметром: после завершения этой функции он становится равным `MPI_REQUEST_NULL`).

и выходной параметр (результат):

- `status` — информация о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции `MPI_Recv`).

Функция `MPI_Wait` блокирует вызвавший ее процесс до окончания обслуживания сообщения с идентификатором `request`. Если

требуется дождаться доставки одного из нескольких сообщений то можно использовать функцию MPI_Waitany. Ее прототип:

```
int MPI_Waitany (int count,
  MPI_Request array_of_requests[], int *index,
  MPI_Status *status);
```

где входные параметры:

- count — количество запросов в массиве array_of_requests,
- array_of_requests — массив идентификаторов запросов на обслуживание сообщений (также является выходным параметром: после завершения этой функции соответствующий элемент array_of_requests[*index] становится равным MPI_REQUEST_NULL)

и выходные параметры (результаты):

- index — содержит номер обработанного сообщения в массиве array_of_requests,
- status — информация о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции MPI_Recv).

Функция MPI_Waitany блокирует вызвавший ее процесс до окончания обработки одного из запросов на обслуживание в массиве array_of_requests.

11.8. Коллективный обмен сообщениями

Все описываемые ниже возможности MPI являются избыточными в том смысле, что любая программа может быть написана без их использования. Действительно, всякий коллективный обмен сообщениями может быть заменен на соответствующий цикл попарных обменов. Однако это приведет к значительному снижению скорости работы программы, так как, во-первых, простейшее решение (один из процессов рассылает данные всем остальным) не годится (в каждый момент времени работают только два процесса), а, во-вторых, любое решение (например,

рассылка по принципу бинарного дерева: первый процесс посылает данные второму, затем первый и второй посылают данные третьему и четвертому соответственно, и т. д.) не будет учитывать специфику конкретной вычислительной установки.

Сообщения, посылаемые с помощью функций коллективного обмена, не могут быть получены с помощью функций попарного обмена и наоборот. Напомним также, что в первых версиях MPI все функции коллективного обмена были синхронными и это наложило отпечаток на их синтаксис.

Для синхронизации процессов можно использовать функцию `MPI_Barrier`. Ее прототип:

```
int MPI_Barrier (MPI_Comm comm);
```

где входной параметр

- `comm` — коммуникатор.

Функция `MPI_Barrier` блокирует вызвавший ее процесс до тех пор, пока все процессы в группе с идентификатором `comm` (коммуникаторе) не вызовут эту функцию (ср. с функцией `synchronize` в разделе 10.4).

Для рассылки данных из одного процесса всем остальным в его группе можно использовать функцию `MPI_Bcast`. Ее прототип:

```
int MPI_Bcast (void *buf, int count,
  MPI_Datatype datatype, int root, MPI_Comm comm);
```

где входные параметры:

- `buf`
  - в процессе с номером `root` — адрес буфера с посылаемыми данными (входной параметр),
  - в остальных процессах группы `comm` — адрес буфера с данными, где следует разместить полученное сообщение (выходной параметр),
- `count`

  - в процессе с номером `root` — число элементов (типа `datatype`) в буфере,
  - в остальных процессах группы `comm` — максимальное число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `root` — ранг (номер) отправителя в коммуникаторе (группе) `comm`,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `buf` — в процессах группы `comm` с номерами, отличными от `root`, — полученное сообщение.

Эта функция должна быть вызвана **во всех** процессах группы `comm` с **одинаковыми** значениями для аргументов `root` и `comm`. Также в большинстве реализаций MPI требуется, чтобы значения аргументов `count` и `datatype` были одинаковыми во всех процессах группы `comm`. Эта функция рассылает сообщение `buf` из процесса с номером `root` всем процессам группы `comm`. Поскольку все процессы вызывают эту функцию одновременно и с одинаковыми аргументами, то, в отличие от функций `MPI_Send` и `MPI_Recv`, поля `tag` и `status` отсутствуют.

Часто требуется выполнить в каком-то смысле обратную к производимой функцией `MPI_Bcast` операцию — переслать данные одному из процессов группы от всех остальных процессов этой группы; при этом часто нужно получить не сами эти данные, а некоторую функцию от них, например, сумму всех данных. Для этого можно использовать функцию `MPI_Reduce`. Ее прототип:

```
int MPI_Reduce (void *sendbuf, void *recvbuf,
  int count, MPI_Datatype datatype, MPI_Op op,
  int root, MPI_Comm comm);
```

где входные параметры:

- `sendbuf` — адрес буфера с данными,
- `count` — число элементов (типа `datatype`) в буфере,

Таблица 11.2. Операции над данными в MPI

Операция MPI	Значение
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое «и»
MPI_BAND	побитовое «и»
MPI_LOR	логическое «или»
MPI_BOR	побитовое «или»
MPI_LXOR	логическое «исключающее или»
MPI_BXOR	побитовое «исключающее или»
MPI_MAXLOC	максимум и его позиция
MPI_MINLOC	минимум и его позиция

- `datatype` — тип данных каждого из элементов буфера,
- `op` — идентификатор операции (типа `MPI_Op`), которую нужно осуществить над пересланными данными для получения результата в буфере `recvbuf`; см. в табл. 11.2 возможные значения этого аргумента,
- `root` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `recvbuf` — указатель на буфер, где требуется получить результат; используется только в процессе с номером `root` в группе `comm`.

Эта функция должна быть вызвана **во всех** процессах группы `comm` с **одинаковыми** значениями для аргументов `root`, `comm`, `count`, `datatype`, `op`.

Если требуется, чтобы результат, полученный посредством функции `MPI_Reduce`, стал известен не только одному процессу (с номером `root` в группе `comm`), а всем процессам группы, то

можно использовать MPI_Bcast, но более эффективным решением является функция MPI_Allreduce. Ее прототип:

```
int MPI_Allreduce (void *sendbuf, void *recvbuf,
  int count, MPI_Datatype datatype, MPI_Op op,
  MPI_Comm comm);
```

где входные параметры:

- sendbuf — адрес буфера с данными,
- count — число элементов (типа datatype) в буфере,
- datatype — тип данных каждого из элементов буфера,
- op — идентификатор операции (типа MPI_Op), которую нужно осуществить над пересланными данными для получения результата в буфере recvbuf; см. в табл. 11.2 возможные значения этого аргумента,
- comm — коммуникатор,

и выходной параметр (результат):

- recvbuf — указатель на буфер, где требуется получить результат.

Функция аналогична MPI_Reduce, но результат образуется в буфере recvbuf во всех процессах группы comm, поэтому нет аргумента root.

Если требуется выполнить свою собственную операцию над данными в функциях MPI_Reduce и MPI_Allreduce, то можно использовать функцию MPI_Op_create. Ее прототип:

```
int MPI_Op_create (MPI_User_function *function,
  int commute, MPI_Op *op);
```

где входные параметры:

- function — указатель на пользовательскую функцию, задающую операцию и имеющую тип MPI_User_function, определенный как

```
typedef void (MPI_User_function) (void * a,
    void * b, int * len, MPI_Datatype * datatype);
```

Эта функция выполняет операцию op над элементами типа datatype массивов a и b длины *len и складывает результат в массив b:

```
for (i = 0; i < *len; i++)
  b[i] = a[i] op b[i];
```

- commute — целое число, истинное, если операция коммутативна, и равное 0 иначе,

и выходной параметр (результат):

- op — идентификатор операции (типа MPI_Op).

Полученный идентификатор операции op можно использовать в функциях MPI_Reduce и MPI_Allreduce. После окончания работы с созданной операцией необходимо освободить используемые для нее ресурсы с помощью функции MPI_Op_free. Ее прототип:

```
int MPI_Op_free (MPI_Op *op);
```

где входной параметр:

- op — идентификатор операции, возвращенный функцией MPI_Op_create,

и выходной параметр (результат):

- op — устанавливается в MPI_OP_NULL

Важным случаем коллективных обменов является запрос на завершение всех процессов, например, в случае ошибки в одном из них. Для этого можно использовать функцию MPI_Abort. Ее прототип:

```
int MPI_Abort (MPI_Comm comm, int errorcode);
```

где входные параметры:

- comm — коммуникатор, описывающий группу задач, которую надо завершить,
- errorcode — код завершения (аналог аргумента функции exit).

Функция завершает все процессы в группе comm. В большинстве реализаций завершаются **все** запущенные процессы.

11.9. Пример MPI-программы, вычисляющей определенный интеграл

Рассмотрим в качестве примера программу, вычисляющую определенный интеграл. Идея ускорения работы описана в разделе 1.2, с. 12. Простейшая реализация этой задачи приведена в разделе 1.5, с. 21. Каждый из процессов инициализирует подсистему MPI с помощью `MPI_Init`, получает свой номер и общее количество процессов посредством `MPI_Comm_rank` и `MPI_Comm_size`. Основная работа производится в функции `process_function`, по окончании которой процесс заканчивает работу с MPI с помощью `MPI_Finalize` и завершается. Функция `process_function` вычисляет свою часть интеграла с помощью функции `integrate` и прибавляет его к ответу `total` в процессе с номером 0 посредством `MPI_Reduce`. Процесс с номером 0 перед окончанием своей работы выводит переменную `total` на экран.

В этом разделе мы разовьем описанный выше пример так, чтобы входные данные считывались из файла. Этим будет заниматься функция `get_data`. В процессе с номером 0 она открывает указанный файл, считывает из него данные и закрывает. Затем эти данные рассылаются из процесса с номером 0 всем остальным процессам с помощью `MPI_Bcast`. Текст программы:

```
#include <stdio.h>
#include "mpi.h"
#include "integral.h"

/* Получить данные */
void get_data (char * name, double *a_ptr, double *b_ptr,
               int *n_ptr, int my_rank);

int
main (int argc, char **argv)
```

```
{
  int my_rank;      /* ранг текущего процесса */
  int p;            /* общее число процессов */
  double a;         /* левый конец интервала */
  double b;         /* правый конец интервала */
  int n;            /* число точек разбиения */
  /* длина отрезка интегрирования для текущего процесса*/
  double len;
  /* левый конец интервала для текущего процесса */
  double local_a;
  /* правый конец интервала для текущего процесса */
  double local_b;
  /* число точек разбиения для текущего процесса */
  int local_n;
  /* значение интеграла в текущем процессе */
  double integral;
  double total;     /* результат: интеграл */

  /* Начать работу с MPI */
  MPI_Init (&argc, &argv);

  /* Получить номер текущего процесса в группе всех
     процессов */
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  /* Получить общее количество запущенных процессов */
  MPI_Comm_size (MPI_COMM_WORLD, &p);

  /* Получить данные */
  get_data ("a.dat", &a, &b, &n, my_rank);

  len = (b - a) / p;
  local_n = n / p;

  /* Вычисляем отрезок интегрирования для текущего
```

```
     процесса */
  local_a = a + my_rank * len;
  local_b = local_a + len;
  /* Вычислить интеграл в каждом из процессов */
  integral = integrate (local_a, local_b, local_n);

  /* Сложить все ответы и передать процессу 0 */
  MPI_Reduce (&integral, &total, 1, MPI_DOUBLE, MPI_SUM,
              0, MPI_COMM_WORLD);

  /* Напечатать ответ */
  if (my_rank == 0)
    printf ("Integral from %lf to %lf = %.18lf\n",
            a, b, total);

  /* Заканчиваем работу с MPI */
  MPI_Finalize ();
  return 0;
}

/* Прочитать значения a, b, и n из файла name */
void
get_data (char * name, double *a_ptr, double *b_ptr,
          int *n_ptr, int my_rank)
{
  /* Читаем данные в процессе 0 */
  if (my_rank == 0)
    {
      FILE *in = fopen (name, "r");
      if (!in)
        {
          fprintf (stderr, "Error opening data file %s\n'
                   name);
          *a_ptr = 0.;
          *b_ptr = 0.;
```

```
            *n_ptr = 1;
          }
        else
          {
            if (fscanf (in, "%lf%lf%d", a_ptr, b_ptr, n_ptr)
                != 3)
            {
              fprintf (stderr,
                       "Error reading data file %s\n", name);
              *a_ptr = 0.;
              *b_ptr = 0.;
              *n_ptr = 1;
            }
            fclose (in);
          }
      }
    /* Рассылаем данные из процесса 0 остальным */
    MPI_Bcast (a_ptr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast (b_ptr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast (n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

Текст функции `integrate` приведен в разделе 1.1.

11.10. Работа с временем

В разделе 10.4 мы вычисляли процессорное и астрономическое времена работы программы. Однако для MPI-приложения процессорное время не является показателем скорости работы, поскольку оно не учитывает время передачи сообщений между процессами (которое может быть больше процессорного времени!). С другой стороны, и абсолютное значение астрономического времени может быть различным в разных процессах из-за несинхронности часов на узлах параллельного компьютера. Поэтому MPI предоставляет свои функции работы с временем.

Узнать **астрономическое** время можно с помощью функции `MPI_Wtime`. Ее прототип:

```
double MPI_Wtime ();
```

Она возвращает для текущего процесса астрономическое время в секундах от некоторого фиксированного момента в прошлом. Это время может быть, а может не быть синхронизированным для всех процессов работающей программы. Точность функции `MPI_Wtime` можно узнать с помощью функции `MPI_Wtick`. Ее прототип:

```
double MPI_Wtick ();
```

Она возвращает частоту внутреннего таймера. Например, если счетчик времени увеличивается 100 раз в секунду, то эта функция вернет 10^{-2}.

На всех вычислительных установках, где используется MPI, разрешается запускать только один процесс на каждый процессор (иначе очень тяжело балансировать загруженность всех узлов параллельного компьютера). В такой ситуации астрономическое время **почти** совпадает с временем работы задачи.

Типичная процедура измерения времени работы программы в MPI выглядит следующим образом:

```
double t;
...
/* Синхронизация всех процессов */
MPI_Barrier (MPI_COMM_WORLD);
/* Время начала */
t = MPI_Wtime ();
/*Вызов процедуры, время работы которой надо измерить*/
...
/* Синхронизация всех процессов */
MPI_Barrier (MPI_COMM_WORLD);
/* Время работы */
t = MPI_Wtime () - t;
...
```

11.11. Пример MPI-программы, вычисляющей произведение матрицы на вектор

Рассмотрим задачу вычисления произведения матрицы на вектор (см. раздел 10.4). Файлы проекта:

- `main.c` — инициализация данных и вывод результатов;
- `matrices.c`, `matrices.h` — исходный текст и соответствующий заголовочный файл для функций, работающих с матрицами.
- `Makefile` — для сборки проекта.

Файл `main.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "matrices.h"

/* Получить данные */
void get_data (char * name, int *n_ptr, int my_rank);

/* Количество тестов (для отладки) */
#define N_TESTS 10

int
main (int argc, char **argv)
{
  int my_rank;  /* ранг текущего процесса */
  int p;        /* общее число процессов */
  int n;        /* размер матрицы и векторов */
  double t;     /* время работы всей задачи */

  int first_row, last_row, rows;
  int max_rows;
  double *matrix;       /* матрица */
  double *vector;       /* вектор */
```

```
double *result;       /* результирующий вектор */
int i;

/* Начать работу с MPI */
MPI_Init (&argc, &argv);

/* Получить номер текущего процесса в группе всех
   процессов */
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

/* Получить общее количество запущенных процессов */
MPI_Comm_size (MPI_COMM_WORLD, &p);

/* Получить данные */
get_data ("a.dat", &n, my_rank);

/* Первая участвующая строка матрицы */
first_row = n * my_rank;
first_row /= p;
/* Последняя участвующая строка матрицы */
last_row = n * (my_rank + 1);
last_row = last_row / p - 1;
/* Количество участвующих строк матрицы */
rows = last_row - first_row + 1;

/* Вычисляем максимальное количество строк на процесс*/
max_rows = n / p + n % p;

/* Выделение памяти под массивы */
if (!(matrix = (double*)
              malloc (max_rows * n * sizeof (double)))
  {
    fprintf (stderr, "Not enough memory!\n");
    MPI_Abort (MPI_COMM_WORLD, 1);
  }
```

```
  if (!(vector = (double*)
                 malloc (max_rows * sizeof (double))))
    {
      fprintf (stderr, "Not enough memory!\n");
      MPI_Abort (MPI_COMM_WORLD, 2);
    }
  if (!(result = (double*)
                 malloc (max_rows * sizeof (double))))
    {
      fprintf (stderr, "Not enough memory!\n");
      MPI_Abort (MPI_COMM_WORLD, 3);
    }

  /* Инициализация массивов */
  init_matrix (matrix, n, first_row, last_row);
  init_vector (vector, first_row, last_row);
  if (my_rank == 0)
    {
      int l;

      printf ("Matrix:\n");
      print_matrix (matrix, n, first_row, last_row);
      printf ("Vector:\n");
      print_vector (vector, first_row, last_row);

      l = (max_rows * n + 2 * max_rows) * sizeof (double);
      printf ("Allocated %d bytes (%dKb or %dMb) \
of memory per process\n", l, l >> 10, l >> 20);
    }

  /* Синхронизация всех процессов */
  MPI_Barrier (MPI_COMM_WORLD);
  /* Время начала */
  t = MPI_Wtime ();
```

```
  for (i = 0; i < N_TESTS; i++)
  {
  /* Умножить матрицу на вектор для процесса с номером
     my_rank из общего количества p */
  matrix_mult_vector (matrix, vector, result, n,
                      first_row, last_row, my_rank, p);
  printf ("Process %d mult %d times\n", my_rank, i);
  }

  /* Синхронизация всех процессов */
  MPI_Barrier (MPI_COMM_WORLD);
  /* Время работы */
  t = MPI_Wtime () - t;
  if (my_rank == 0)
    {
      print_vector (result, first_row, last_row);
      printf ("Total time = %le\n", t);
    }

  /* Освобождаем память */
  free (matrix);
  free (vector);
  free (result);

  /* Заканчиваем работу с MPI */
  MPI_Finalize ();
  return 0;
}

void
get_data (char * name, int *n_ptr, int my_rank)
{
  /* Читаем данные в процессе 0 */
  if (my_rank == 0)
    {
```

```
      FILE *in = fopen (name, "r");
      if (!in)
        {
          fprintf (stderr, "Error opening data file %s\n",
                   name);
          *n_ptr = 0;
        }
      else
        {
          if (fscanf (in, "%d", n_ptr) != 1)
          {
            fprintf (stderr,
                     "Error reading data file %s\n", name);
            *n_ptr = 0;
          }
          fclose (in);
        }
    }
  /* Рассылаем данные из процесса 0 остальным */
  MPI_Bcast (n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

В функции `main` каждый из процессов инициализирует подсистему MPI с помощью `MPI_Init`, получает свой номер `my_rank` и общее количество процессов `p` посредством `MPI_Comm_rank` и `MPI_Comm_size`. Затем значение размерности матрицы `n` считывается из файла функцией `get_data` так, как описано в разделе 11.9. В каждом из процессов хранятся строки матрицы `a` и компоненты векторов `b` и `c` с номерами

```
n * my_rank / p, ..., n * (my_rank + 1)  / p - 1.
```

Поскольку эти блоки пересылаются между процессами, то память в каждом из них выделяется под блок с максимальным числом компонент

```
max_rows = n / p + n % p.
```

Блоки инициализируются в каждом из процессов, затем процесс с номером 0 печатает свою часть. С помощью механизма, описанного в разделе 11.10, замеряется время работы функции `matrix_mult_vector`, вычисляющей компоненты произведения матрицы на вектор, имеющие индексы в диапазоне

```
n * my_rank / p, ..., n * (my_rank + 1)  / p - 1.
```

Для целей отладки и более точного замера времени работы функция вычисления произведения матрицы на вектор вызывается в цикле `N_TESTS` раз.

Заголовочный файл `matrices.h`:

```
void init_matrix (double * matrix, int n, int first_row,
                  int last_row);
void init_vector (double * vector, int first_row,
                  int last_row);
void print_matrix (double * matrix, int n, int first_row,
                   int last_row);
void print_vector (double * vector, int first_row,
                   int last_row);
void matrix_mult_vector (double *a, double *b, double *c,
                         int n, int first_row,
                         int last_row, int my_rank,int p);
```

Файл `matrices.c`:

```
#include <stdio.h>
#include "matrices.h"
#include "mpi.h"

/* Инициализация матрицы */
void init_matrix (double * matrix, int n, int first_row,
                  int last_row)
{
  int i, j;
  double *a = matrix;
```

```
  for (i = first_row; i <= last_row; i++)
    for (j = 0; j < n; j++)
      *(a++) = (i > j) ? i : j;
}

/* Инициализация вектора */
void init_vector (double * vector, int first_row,
                  int last_row)
{
  int i;
  double *b = vector;

  for (i = first_row; i <= last_row; i++)
    *(b++) = 1.;
}

#define N_MAX   6

/* Вывод матрицы */
void print_matrix (double * matrix, int n, int first_row
                   int last_row)
{
  int i, j;
  int rows = last_row - first_row + 1;
  int mi = (rows > N_MAX ? N_MAX : rows);
  int mj = (n > N_MAX ? N_MAX : n);

  for (i = 0; i < mi; i++)
    {
      for (j = 0; j < mj; j++)
        printf (" %12.6lf", matrix[i * n + j]);
      printf ("\n");
    }
}
```

```
/* Вывод вектора */
void print_vector (double * vector, int first_row,
                   int last_row)
{
  int i;
  int rows = last_row - first_row + 1;
  int m = (rows > N_MAX ? N_MAX : rows);

  for (i = 0; i < m; i++)
      printf (" %12.6lf", vector[i]);
  printf ("\n");
}

/* Умножить матрицу a на вектор b, c = ab для процесса с
   номером my_rank из общего количества p */
void matrix_mult_vector (double *a, double *b, double *c,
                         int n, int first_row,
                         int last_row, int my_rank, int p
{
  int i, j, k, l;
  int rows = last_row - first_row + 1;
  int max_rows;
  int first_row_k, last_row_k, rows_k;
  double s;
  int dest, source, tag = 0;
  MPI_Status status;

  /* Вычисляем максимальное количество строк на процесс*/
  max_rows = n / p + n % p;

  /* Обнуляем результат */
  for (i = 0; i < rows; i++)
    c[i] = 0.;

  /* Вычисляем источник и получатель для текущего
```

```
   процесса */
source = (my_rank + 1) % p;
if (my_rank == 0)
  dest = p - 1;
else
  dest = my_rank - 1;

/* Цикл по блокам */
for (l = 0; l < p; l++)
  {
    /* Номер процесса, от которого был получен вектор*/
    k = (my_rank + l) % p;
    /* Первая участвующая строка матрицы в процессе k*/
    first_row_k = n * k;
    first_row_k /= p;
    /* Последняя участвующая строка в процессе k */
    last_row_k = n * (k + 1);
    last_row_k = last_row_k / p - 1;
    /* Количество участвующих строк в процессе k */
    rows_k = last_row_k - first_row_k + 1;

    /* Умножаем прямоугольный блок матрицы a
       в строках  first_row ... last_row
       и столбцах first_row_k ... last_row_k
       на вектор b, соответствующий компонентам
       first_row_k ... first_row_k */
    for (i = 0; i < rows; i++)
      {
        for (s = 0., j = 0; j < rows_k; j++)
          s += a[i * n + j + first_row_k] * b[j];
        c[i] += s;
      }

    /* Пересылаем вектор b процессу dest и получаем его
       от source */
```

```
      MPI_Sendrecv_replace (b, max_rows, MPI_DOUBLE,
        dest, tag, source, tag, MPI_COMM_WORLD, &status);
    }
}
```

Функция `matrix_mult_vector` получает указатели на строки матрицы `a` и компоненты векторов `b` и `c` с номерами

```
n * my_rank / p, ..., n * (my_rank + 1) / p - 1
```

и вычисляет указанные компоненты ответа `c` — произведения матрицы `a` на вектор `b`. При этом память под матрицы и векторы выделена для хранения

```
max_rows = n / p + n % p
```

компонент (строк матрицы или элементов вектора).

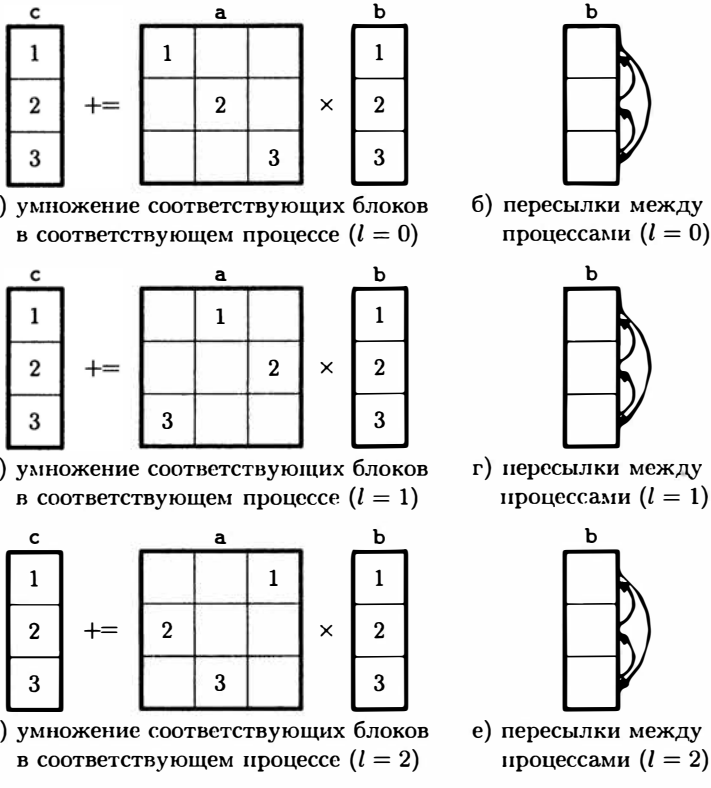
Обозначим $A = (a_{ij})$, $i, j = 0, 1, \ldots, n-1$ — матрица A, $b = (b_i)$, $i = 0, 1, \ldots, n-1$, $c = (c_i)$, $i = 0, 1, \ldots, n-1$ — векторы b и c, $m =$ `my_rank`. Функция `matrix_mult_vector` в цикле по $l = 0, 1, \ldots, p-1$ вычисляет вектор, являющийся произведением блока матрицы A, стоящего в строках

$$nm/p, \ldots, n(m+1)/p - 1 \tag{11.1}$$

и столбцах

$$(nm/p + l) \pmod p, \ldots, (n(m+1)/p - 1 + l) \pmod p, \tag{11.2}$$

и вектора, образованного компонентами вектора b с номерами (11.2). Этот вектор прибавляется к вектору, образованному компонентами вектора c с номерами (11.1). Затем вектор, образованный компонентами вектора b с номерами (11.2), пересылается процессу с номером $(m-1) \pmod p$, а на его место помещается вектор, полученный от процесса с номером $(m+1) \pmod p$, т. е. осуществляется циклическая пересылка доступной каждому процессу части вектора b длиной не более `max_rows`.



а) умножение соответствующих блоков в соответствующем процессе ($l = 0$)

б) пересылки между процессами ($l = 0$)

в) умножение соответствующих блоков в соответствующем процессе ($l = 1$)

г) пересылки между процессами ($l = 1$)

д) умножение соответствующих блоков в соответствующем процессе ($l = 2$)

е) пересылки между процессами ($l = 2$)

Рис. 11.1. Организация вычислений и пересылки данных при умножении матрицы на вектор

Иллюстрация этого процесса при $p = 3$ приведена на рис. 11.1:

а) в каждом из процессов 1, 2, 3 (здесь мы для удобства нумеруем их, начиная с 1) вычисляем произведения блоков матрицы a и вектора b, отмеченных на рисунке соответствующей цифрой, и прибавляем их к результату в векторе c;

б) циклически пересылаем блоки вектора b;

в) перемножаем соответствующие блоки матрицы a и вектора b и прибавляем их к результату в векторе c;

г) циклически пересылаем блоки вектора b;

д) перемножаем соответствующие блоки матрицы a и вектора b и прибавляем их к результату в векторе c, получая при этом окончательный результат;

е) циклически пересылаем блоки вектора b, восстанавливая его первоначальное состояние.

Файл Makefile:

```
NAME    = mpi_mult

DEBUG   =
CC      = mpicc -c
LD      = mpicc
CFLAGS  = $(DEBUG) -W -Wall
LIBS    = -lm
LDFLAGS = $(DEBUG)

OBJS = main.o matrices.o

all : $(NAME)

$(NAME) : $(OBJS)
        $(LD) $(LDFLAGS) $^ $(LIBS) -o $@

.c.o:
        $(CC) $(CFLAGS) $< -o $@

clean:
        rm -f $(OBJS) $(NAME)

main.o          : main.c matrices.h
matrices.o      : matrices.c matrices.h
```

11.12. Дополнительные функции коллективного обмена сообщениями для работы с массивами

В программе, вычисляющей произведение матрицы на вектор (см. раздел 11.11), ни один из процессов не хранит массивы полностью, а в каждый момент времени работает только с их частями. Такая MPI-программа при запуске на установке с общей памятью не использует для хранения массивов памяти больше, чем ее multithread-аналог (см. раздел 10.4). Если некоторому процессу потребовался весь массив целиком (например, для вывода в процессе с номером 0), то можно переслать ему части массива от других процессов с помощью функций попарного обмена сообщениями. Однако удобнее и быстрее это сделать посредством функции `MPI_Gather`. Ее прототип:

```
int MPI_Gather (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf,
  int recvcount, MPI_Datatype recvtype, int root,
  MPI_Comm comm);
```

где входные параметры:

- `sendbuf` — адрес буфера с посылаемыми данными,
- `sendcount` — число элементов (типа `sendtype`) в буфере `sendbuf`,
- `sendtype` — тип данных каждого из элементов буфера `sendbuf`,
- `recvbuf` — адрес буфера с данными, где следует разместить полученное сообщение (используется только в процессе с номером `root` группы `comm`),
- `recvcount` — максимальное число элементов (типа `recvtype`) в буфере `recvbuf` (используется только в процессе с номером `root` группы `comm`),
- `recvtype` — тип данных каждого из элементов буфера `recvbuf` (используется только в процессе с номером `root` группы `comm`),

- `root` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `comm` — коммуникатор,

и выходной параметр (результат) в процессе с номером `root` группы `comm`:

- `recvbuf` — указатель на полученные данные.

Эта функция:

- в процессе с номером `root` группы `comm` — принимает `recvcount` данных типа `recvtype` от всех остальных процессов группы `comm` и размещает их последовательно в буфере `recvbuf`: вначале данные от процесса с номером 0 группы `comm`, затем данные от процесса с номером 1 группы `comm` и т. д.; от самого себя процесс данные, конечно, не принимает, а просто копирует их из `sendbuf` в соответствующее место `recvbuf`;
- в процессе группы `comm` с номером, отличным от `root`, — посылает `sendcount` данных типа `sendtype` из буфера `sendbuf` процессу с номером `root`.

Эта функция должна быть вызвана **во всех** процессах группы `comm` с **одинаковыми** значениями для аргументов `root` и `comm`. Также в большинстве случаев требуется, чтобы значения аргументов `sendcount` и `sendtype` были одинаковыми во всех процессах группы `comm`, причем равными значениям `recvcount` и `recvtype` соответственно.

Если требуется, чтобы результат, полученный посредством функции `MPI_Gather`, стал известен не только одному процессу (с номером `root` в группе `comm`), а всем процессам группы, то можно использовать `MPI_Bcast`, но более эффективным решением является функция `MPI_Allgather` (ср. ситуацию с функциями `MPI_Reduce` и `MPI_Allreduce`). Ее прототип:

```
int MPI_Allgather (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf,
  int recvcount, MPI_Datatype recvtype,
```

```
    MPI_Comm comm);
```

где входные параметры:

- `sendbuf` — адрес буфера с посылаемыми данными,
- `sendcount` — число элементов (типа `sendtype`) в буфере `sendbuf`,
- `sendtype` — тип данных каждого из элементов буфера `sendbuf`,
- `recvbuf` — адрес буфера с данными, где следует разместить полученное сообщение,
- `recvcount` — максимальное число элементов (типа `recvtype`) в буфере `recvbuf`,
- `recvtype` — тип данных каждого из элементов буфера `recvbuf`,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `recvbuf` — указатель на полученные данные.

Функция аналогична `MPI_Gather`, но результат образуется в буфере `recvbuf` во всех процессах группы `comm`, поэтому нет аргумента `root`.

MPI предоставляет функцию `MPI_Scatter`, в некотором смысле обратную к `MPI_Gather`. Ее прототип:

```
int MPI_Scatter (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf,
  int recvcount, MPI_Datatype recvtype, int root,
  MPI_Comm comm);
```

где входные параметры:

- `sendbuf` — адрес буфера с посылаемыми данными (используется только в процессе с номером `root` группы `comm`),
- `sendcount` — число элементов (типа `sendtype`) в буфере `sendbuf` (используется только в процессе с номером `root` группы `comm`),

- `sendtype` — тип данных каждого из элементов буфера `sendbuf` (используется только в процессе с номером `root` группы `comm`),
- `recvbuf` — адрес буфера с данными, где следует разместить полученное сообщение,
- `recvcount` — максимальное число элементов (типа `recvtype`) в буфере `recvbuf`,
- `recvtype` — тип данных каждого из элементов буфера `recvbuf`,
- `root` — ранг (номер) отправителя в коммуникаторе (группе) `comm`,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `recvbuf` — указатель на полученные данные.

Эта функция:

- в процессе с номером `root` группы `comm` — посылает первые `sendcount` элементов типа `sendtype` в буфере `sendbuf` процессу с номером 0 группы `comm`, следующие `sendcount` элементов типа `sendtype` в буфере `sendbuf` — процессу с номером 1 группы `comm` и т. д.; сам себе процесс данные, конечно, не посылает, а просто копирует их из соответствующего места `sendbuf` в `recvbuf`;
- в процессе группы `comm` с номером, отличным от `root`, — принимает `recvcount` данных типа `recvtype` от процесса с номером `root` и складывает в буфер `recvbuf`.

Эта функция должна быть вызвана **во всех** процессах группы `comm` с **одинаковыми** значениями для аргументов `root` и `comm`. Также в большинстве случаев требуется, чтобы значения аргументов `recvcount` и `recvtype` были одинаковыми во всех процессах группы `comm`, причем равными значениям `sendcount` и `sendtype` соответственно.

11.13. Пересылка структур данных

Рассмотрим задачу пересылки между процессами определенной пользователем структуры данных на примере двух наиболее часто встречающихся задач:

1. пересылка разнородных данных, локализованных в одном блоке памяти (например, пересылка определенного с помощью конструкции `struct` языка C типа данных);
2. пересылка распределенных в памяти однородных данных (например, пересылка столбца матрицы в языке C, где многомерные массивы хранятся по строкам).

Описанные ниже способы пересылки можно комбинировать, образуя, например, методы пересылки распределенных разнородных данных и т. д.

11.13.1. Пересылка локализованных разнородных данных

Пусть между процессами требуется пересылать определенную пользователем структуру данных, например, следующего вида:

```
typedef struct
  {
    int n;
    char s[20];
    double v;
  }
  USERTYPE;
```

Для этой задачи можно предложить следующие решения:

1. один раз создать новый MPI-тип, соответствующий C типу `USERTYPE`, и использовать его в функциях обмена сообщениями;
2. при каждой пересылке процесс–отправитель запаковывает объект типа `USERTYPE` в один объект типа `MPI_PACKED` и пересылает его процессу–получателю, который его распаковывает;

3. при любом обмене объект типа USERTYPE рассматривается как массив длины sizeof(USERTYPE) элементов типа MPI_BYTE (это решение работает только на однородных вычислительных установках).

Создание нового типа данных. При создании нового типа данных подсистеме MPI необходимо указать спецификацию этого типа следующего вида:

$$n,\ \{(c_0, d_0, t_0), (c_1, d_1, t_1), \ldots, (c_{n-1}, d_{n-1}, t_{n-1})\}, \qquad (11.3)$$

где

- n (типа int) — количество элементов базовых типов в новом типе;
- c_i (типа int) — количество элементов типа t_i в i-м базовом элементе нового типа;
- d_i (типа MPI_Aint, «address int») — смещение i-го базового элемента от начала объекта;
- t_i (типа MPI_Datatype) — тип i-го базового элемента.

Например, для С-типа USERTYPE спецификация MPI-типа на 32-битной вычислительной установке будет иметь вид:

$$3,\ \{(1, 0, \texttt{MPI\_INT}), (20, 4, \texttt{MPI\_CHAR}), (1, 24, \texttt{MPI\_DOUBLE})\}.$$

Использование типа MPI_Aint (равного int или long int) позволяет не зависеть от выбранного в языке С размера типа int на 64-битных компьютерах. Получить адрес любого объекта в терминах типа MPI_Aint можно с помощью функции MPI_Address. Ее прототип:

```
int MPI_Address (void *location, MPI_Aint *address);
```

где входной параметр:

- location — адрес объекта в терминах языка С,

и выходной параметр (результат):

- address — адрес объекта как MPI_Aint.

Функция MPI_Address не связана с межпроцессным обменом и выполняется локально, в одном процессе.

Построением нового типа по спецификации вида (11.3) занимается функция MPI_Type_struct. Ее прототип:

```
int MPI_Type_struct (int count, int blocklens[],
  MPI_Aint displacements[], MPI_Datatype oldtypes[],
  MPI_Datatype *newtype);
```

где входные параметры:

- count — количество элементов в создаваемом типе, т. е. n в (11.3);
- blocklens — массив длины count, содержащий количество элементов базового типа в каждом поле создаваемого типа, т. е.

$$\texttt{blocklens[]} = \{c_0, c_1, \ldots, c_{n-1}\}$$

  в терминах описания (11.3);
- displacements — массив длины count, содержащий смещение элементов базовых типов (полей) от начала объекта создаваемого типа, т. е.

$$\texttt{displacements[]} = \{d_0, d_1, \ldots, d_{n-1}\}$$

  в терминах описания (11.3),
- oldtypes — массив длины count, содержащий типы базовых элементов (полей) создаваемого типа, т. е.

$$\texttt{oldtypes[]} = \{t_0, t_1, \ldots, t_{n-1}\}$$

  в терминах описания (11.3);

и выходной параметр (результат):

- newtype — идентификатор созданного типа.

Функция MPI_Type_struct не связана с межпроцессным обменом и выполняется локально, в одном процессе. Это связано с тем, что мы можем строить новый тип не только для пересылки

данных этого типа между процессами, а для использования его в качестве одного из базовых типов t_i в (11.3).

Созданный тип нельзя сразу использовать для передачи сообщений между процессами. Перед этим необходимо вызвать функцию `MPI_Type_commit`, которая создает для указанного в качестве аргумента типа все структуры данных, необходимые для его эффективной пересылки. Ее прототип:

```
int MPI_Type_commit (MPI_Datatype *newtype);
```

где единственный входной и выходной аргумент `newtype` — указатель на созданный тип. Эта функция должна быть вызвана **всеми** процессами, которые будут обмениваться между собой объектами типа `newtype`.

Если построенный тип не требуется для дальнейшей работы программы, то необходимо освободить все выделенные в момент его построения ресурсы с помощью функции `MPI_Type_free`. Ее прототип:

```
int MPI_Type_free (MPI_Datatype *newtype);
```

где единственный входной и выходной аргумент `newtype` — указатель на созданный тип, который после выполнения этой функции становится равным `MPI_DATATYPE_NULL`. Если в качестве аргумента этой функции передать стандартный MPI-тип (например, любой из табл. 11.1), то это приведет к ошибке.

Теперь мы можем привести законченный пример построения MPI-типа, соответствующего C-типу `USERTYPE`:

```
/* Объект исходного типа USERTYPE */
USERTYPE t = { 1, {'H', 'e', 'l', 'l', 'o', 0}, 2. };
/* Количество элементов в каждом поле USERTYPE */
int t_c[3] = { 1, 20, 1 };
/* Смещение каждого поля относительно начала
   структуры USERTYPE: предстоит вычислить */
MPI_Aint t_d[3];
/* MPI-тип каждого поля в USERTYPE */
MPI_Datatype t_t[3] = {MPI_INT, MPI_CHAR, MPI_DOUBLE};
```

```
/* Результат: MPI-тип, соответствующий USERTYPE */
MPI_Datatype MPI_USERTYPE;
/* Переменные, используемые для вычисления смещений */
MPI_Aint start_address, address;

/* Вычисляем адрес начала структуры */
MPI_Address (&t, &start_address);
/* Вычисляем смещение первого поля */
MPI_Address (&t.n, &address);
t_d[0] = address - start_address;
/* Вычисляем смещение второго поля */
MPI_Address (&t.s, &address);
t_d[1] = address - start_address;
/* Вычисляем смещение третьего поля */
MPI_Address (&t.v, &address);
t_d[2] = address - start_address;

/* Создаем MPI-тип данных с вычисленными свойствами */
MPI_Type_struct (3, t_c, t_d, t_t, &MPI_USERTYPE);
/* Делаем его доступным при обмене сообщениями */
MPI_Type_commit (&MPI_USERTYPE);

/* Тип MPI_USERTYPE можно теперь использовать наряду
   со стандартными типами */
...
/* Например: разослать данные из процесса 0 остальным*/
MPI_Bcast (&t, 1, MPI_USERTYPE, 0, MPI_COMM_WORLD);
...
/* Удалить тип, если он больше не понадобится */
MPI_Type_free (&MPI_USERTYPE);
```

Отметим, что процедура построения нового типа является достаточно длительной и оправдана лишь в том случае, если этот тип будет использоваться многократно.

Запаковка/распаковка данных может применяться к любым данным, необязательно локализованным в одном блоке памяти. При запаковке данные последовательно записываются подряд в указанном пользователем буфере, а затем пересылаются как один объект типа `MPI_PACKED`. При распаковке данные последовательно извлекаются из этого буфера.

Запаковать данные можно с помощью функции `MPI_Pack`. Ее прототип:

```
int MPI_Pack (void *pack_data, int pack_count,
  MPI_Datatype pack_type, void *buffer,
  int buffer_size, int *position, MPI_Comm comm);
```

где входные параметры:

- `pack_data` — адрес пакуемых данных,
- `pack_count` — количество пакуемых элементов (каждый типа `pack_type`),
- `pack_type` — тип каждого из пакуемых `pack_count` данных,
- `buffer` — адрес буфера, в который складывается запакованный результат,
- `buffer_size` — размер буфера `buffer`,
- `position` — адрес целого числа, задающего смещение в байтах первой свободной позиции в `buffer`,
- `comm` — коммуникатор, идентифицирующий группу процессов, которая будет в последующем использовать `buffer` для обмена данными,

и выходные параметры (результаты):

- `buffer` — буфер, в котором образуется запакованный результат,
- `position` — указывает уже на новое значение первой свободной позиции в буфере.

Узнать размер буфера, необходимого для запаковки данных, можно с помощью функции `MPI_Pack_size`. Ее прототип:

```
int MPI_Pack_size (int pack_count,
```

```
MPI_Datatype pack_type, MPI_Comm comm, int *size);
```

где входные параметры:

- `pack_count` — количество пакуемых элементов (каждый типа `pack_type`),
- `pack_type` — тип каждого из пакуемых `pack_count` данных,
- `comm` — коммуникатор, идентифицирующий группу процессов, которая будет в последующем использовать запакованные данные,

и выходной параметр (результат):

- `size` — размер буфера (в байтах), достаточный для запаковки `pack_count` элементов типа `pack_type`.

Запакованные данные можно переслать/получить любой функцией передачи сообщений, указав в качестве типа `MPI_PACKED`, а в качестве размера — `buffer_size`. Если используется попарный обмен сообщениями, то можно избежать передачи неиспользованной части буфера, указав в процессе-отправителе в качестве длины значение `position` после последнего вызова `MPI_Pack`, а в процессе-получателе — значение `buffer_size` (поскольку значение `position` известно только отправителю).

Распаковать данные можно с помощью функции `MPI_Unpack`. Ее прототип:

```
int MPI_Unpack (void *buffer, int buffer_size,
  int *position, void *unpack_data,
  int unpack_count, MPI_Datatype unpack_type,
  MPI_Comm comm);
```

где входные параметры:

- `buffer` — адрес буфера, из которого берется запакованный результат,
- `buffer_size` — размер буфера `buffer`,
- `position` — адрес целого числа, задающего смещение в байтах позиции в `buffer`, с которой начинается распаковка,

- `unpack_data` — адрес буфера, в который следует разместить распаковываемые данные,
- `unpack_count` — количество распаковываемых элементов (типа `unpack_type`),
- `unpack_type` — тип каждого из `unpack_count` распаковываемых данных,
- `comm` — коммуникатор, идентифицирующий группу процессов, которая использовала `buffer` для обмена данными,

и выходные параметры (результаты):

- `unpack_data` — распакованные данные,
- `position` — указывает уже на новое значение позиции в буфере, с которой начнется распаковка очередного элемента.

Приведем законченный пример пересылки структуры данных типа `USERTYPE` от процесса 0 всем остальным процессам:

```
/* Размер буфера */
#define BUF_SIZE 128
char buffer[BUF_SIZE]; /* буфер */
int position; /* текущая позиция в буфере */
/* Объект исходного типа USERTYPE */
USERTYPE t;

if (my_rank == 0)
  {
    /* Процесс 0 заносит данные в объект t */
    t.n = 1;
    strcpy (t.s, "Hello");
    t.v = 2.;

    /* Запаковываем объект t в процессе 0 */
    position = 0; /* складываем в начало буфера */
    /* Пакуем первое поле */
    MPI_Pack (&t.n, 1, MPI_INT, buffer, BUF_SIZE,
     &position, MPI_COMM_WORLD);
```

```
    /* Пакуем второе поле, position указывает на место,
     где следует разместить результат */
    MPI_Pack (&t.s, 20, MPI_CHAR, buffer, BUF_SIZE,
     &position, MPI_COMM_WORLD);
    /* Пакуем третье поле, position указывает на место,
     где следует разместить результат */
    MPI_Pack (&t.v, 1, MPI_DOUBLE, buffer, BUF_SIZE,
     &position, MPI_COMM_WORLD);
  }

/* Рассылаем буфер из процесса 0 всем остальным */
MPI_Bcast (buffer, BUF_SIZE, MPI_PACKED, 0,
  MPI_COMM_WORLD);

if (my_rank != 0)
  {
    /* Распаковываем объект t в остальных процессах */
    position = 0; /*начинаем с первой позиции буфера*/
    /* Распаковываем первое поле */
    MPI_Unpack (buffer, BUF_SIZE, &position, &t.n, 1,
     MPI_INT, MPI_COMM_WORLD);
    /* Распаковываем второе поле, position указывает
     на место, откуда следует начинать */
    MPI_Unpack (buffer, BUF_SIZE, &position, &t.s, 20,
     MPI_CHAR, MPI_COMM_WORLD);
    /* Распаковываем третье поле, position указывает
     на место, откуда следует начинать */
    MPI_Unpack (buffer, BUF_SIZE, &position, &t.v, 1,
     MPI_DOUBLE, MPI_COMM_WORLD);
  }
```

Отметим, что использование MPI_Pack, MPI_Unpack для передачи информации может приводить к многократному копированию данных: вначале они помещаются в буфер функцией MPI_Pack, затем, при отправке, — во внутренний буфер MPI

для формирования сообщения. При получении сообщения данные копируются из внутреннего буфера MPI в буфер функции `MPI_Unpack`, затем уже они попадают на свое место. Поэтому этот способ можно рекомендовать лишь для единичных обменов. Некоторые реализации MPI для уменьшения лишних пересылок данных обрабатывают тип `MPI_PACKED` специальным образом: данные этого типа не буферизируются подсистемой MPI, а сам `buffer` в `MPI_Pack`, `MPI_Unpack` используется для формирования сообщения. Это снижает накладные расходы при таком способе обмена.

Пересылка неформатированных данных. MPI-подсистема пересылки сообщений работает с блоками данных, о которых ей нужно знать лишь размер блока. Содержимое блока (т. е. тип данных элементов) при пересылке не важен. Тип данных становится важным лишь только после доставки сообщения, когда данные в двоичном формате процесса-отправителя должны быть представлены в двоичном формате процесса-получателя. Если эти процессы работают на процессорах разной архитектуры, то двоичный формат данных может отличаться (см. главу 4). Как отмечалось в главе 4, все современные процессоры имеют единый формат для целочисленных данных и данных с плавающей точкой. Отличаться может лишь способ нумерации байтов в этих данных, и тогда после доставки сообщения необходимо преобразование двоичных данных. Ради поддержки таких гетерогенных вычислительных установок (т. е. составленных из процессоров разной архитектуры) система MPI требует указания типа каждого пересылаемого объекта.

Однако оказалось, что разработка MPI-программ является весьма сложной, и подавляющее большинство программного обеспечения строится при ряде упрощающих его разработку предположений. Основным из них является предположение о равенстве производительности всех процессоров, на которых выполняется MPI-программа. Собственно, это предполагаем и мы

во всех примерах, разделяя вычислительную работу **поровну** между всеми процессами. Следовательно, если, например, один процессор имеет меньшую производительность, чем остальные, то вся программа будет работать именно с его скоростью, поскольку остальные процессы будут ожидать работающий на нем процесс в точках обмена сообщениями. Поэтому в настоящий момент подавляющее большинство параллельных ЭВМ используют не просто процессоры одной архитектуры, а **одинаковые** процессоры (т. е. одной архитектуры с равной тактовой частотой).

Рассмотрим задачу пересылки структуры данных на однородной в слабом смысле вычислительной установке (т. е. использованы процессоры одной архитектуры, а их частота нам не важна). Эту структуру можно переслать как единый блок элементов типа `MPI_BYTE` размером, равным размеру структуры в байтах. Приведем законченный пример пересылки структуры данных типа `USERTYPE` от процесса 0 всем остальным процессам:

```
/* Объект исходного типа USERTYPE */
USERTYPE t;

if (my_rank == 0)
  {
    /* Процесс 0 заносит данные в объект t */
    t.n = 1;
    strcpy (t.s, "Hello");
    t.v = 2.;
  }

/* Рассылаем структуру из процесса 0 всем остальным */
MPI_Bcast (&t, sizeof (USERTYPE), MPI_BYTE, 0,
  MPI_COMM_WORLD);
```

Этот текст является самым коротким и быстродействующим из всех приведенных выше, но работает не на всех вычислительных установках.

11.13.2. Пересылка распределенных однородных данных

Рассмотрим подробнее работу с двумерными массивами. Если мы храним $n \times n$ массив a как вектор размера n^2:

```
double * a; /* массив n x n */,
```

то переслать k-ю строку этого массива от процесса `src` процессу `dest` можно посредством

```
if (my_rank == src)
  MPI_Send (a + k * n, n, MPI_DOUBLE, dest, tag,
    MPI_COMM_WORLD);
else if (my_rank == dest)
  MPI_Recv (a + k * n, n, MPI_DOUBLE, src, tag,
    MPI_COMM_WORLD, &status);
```

Если требуется переслать k-й столбец, то можно скопировать его элементы в вектор длины n и затем переслать его; процесс-получатель принимает данные также во вспомогательный вектор и затем заносит их на место k-го столбца. Однако может получиться, что данные при отправке будут копироваться многократно: вначале из столбца матрицы в наш вспомогательный вектор, затем – во внутренний буфер MPI для формирования сообщения. То же, но в обратном порядке, происходит при получении сообщения. Если такие пересылки происходят часто, то имеет смысл создать новый тип MPI для столбца матрицы. Это можно сделать с помощью функции `MPI_Type_struct`, как описано в разделе 11.13.1. Но стандарт MPI предоставляет для этого случая более удобную функцию `MPI_Type_vector`. Ее прототип:

```
int MPI_Type_vector (int count, int blocklen,
  int stride, MPI_Datatype oldtype,
  MPI_Datatype *newtype);
```

где входные параметры:

- `count` — количество элементов в создаваемом векторном типе,

- `blocklen` — длина одного элемента (в единицах типа `oldtype`) в создаваемом векторном типе,
- `stride` — расстояние между элементами (в единицах типа `oldtype`) в создаваемом векторном типе,
- `oldtype` — базовый тип элементов в создаваемом векторном типе,

и выходной параметр (результат):

- `newtype` — идентификатор созданного векторного типа.

Функция создает новый тип данных: вектор длины `count`, элементами которого являются массивы длины `blocklen` объектов типа `oldtype`; расстояние между элементами — `stride` объектов типа `oldtype`. Например, тип, соответствующий столбцу $n \times n$ матрицы, можно создать следующим образом:

```
MPI_Datatype column_t;
MPI_Type_vector (n, 1, n, MPI_DOUBLE, &column_t);
```

Созданный тип нельзя сразу использовать для передачи сообщений. Перед этим необходимо вызвать функцию `MPI_Type_commit`, см. раздел 11.13.1:

```
MPI_Type_commit (&column_t);
```

После этого переслать `k`-й столбец массива `a` от процесса `src` процессу `dest` можно посредством

```
if (my_rank == src)
  MPI_Send (a + k, 1, column_t, dest, tag,
    MPI_COMM_WORLD);
else if (my_rank == dest)
  MPI_Recv (a + k, 1, column_t, src, tag,
    MPI_COMM_WORLD, &status);
```

Если построенный векторный тип не требуется для дальнейшей работы программы, то необходимо освободить все выделенные в момент его построения ресурсы с помощью функции `MPI_Type_free`.

Напомним, что создание нового типа — достаточно дорогостоящая операция и к ней имеет смысл прибегать, только если тип будет использоваться многократно.

11.14. Ограничение коллективного обмена на подмножество процессов

Пусть нам требуется многократно пересылать сообщения между всеми процессами некоторой подгруппы всех процессов. Можно использовать функции попарного обмена сообщениями, но более эффективным является использование функций коллективного обмена (см. раздел 11.8), ограниченных на эту подгруппу. Для этого нам необходимо создать новый коммуникатор.

С каждым коммуникатором (т. е. идентификатором группы процессов) связана собственно **группа** — список процессов, входящих в группу. Структура этого объекта зависит от реализации MPI, и потому недоступна для непосредственного использования в прикладных программах. Группа имеет тип `MPI_Group` и может быть получена по коммуникатору с помощью функции `MPI_Comm_group`. Ее прототип:

```
int MPI_Comm_group (MPI_Comm comm,
  MPI_Group *group);
```

где входной параметр `comm` — коммуникатор, и выходной параметр (результат) `group` — группа. Создать новую группу, включающую выбранное подмножество процессов из существующей группы (или все, но перенумерованные по-другому), можно с помощью функции `MPI_Group_incl`. Ее прототип:

```
int MPI_Group_incl (MPI_Group group, int n,
  int * ranks, MPI_Group *newgroup);
```

где входные параметры:

- `group` — существующая группа,
- `n` — количество элементов в массиве `ranks` (и тем самым размер новой группы),

- `ranks` — номера процессов из старой группы, которые войдут в новую,

и выходной параметр (результат):

- `newgroup` — созданная группа, процесс номер `k` в ней имеет номер `ranks[k]` в группе `group`, где $k = 0, 1, \ldots, n-1$.

Создать новый коммуникатор, соответствующий созданной группе, можно с помощью функции `MPI_Comm_create`. Ее прототип:

```
int MPI_Comm_create (MPI_Comm comm,
  MPI_Group newgroup, MPI_Comm *newcomm);
```

где входные параметры:

- `comm` — существующий коммуникатор,
- `newgroup` — группа, являющаяся подмножеством группы с идентификатором `comm`,

и выходной параметр (результат):

- `newcomm` — созданный коммуникатор, состоящий из процессов, входящих в `comm` и группу `newgroup`.

Например, создать коммуникатор для коллективного обмена между процессами с четным номером можно с помощью следующего фрагмента (здесь `p` — размер `MPI_COMM_WORLD`):

```
MPI_Group group_world;
MPI_Group group_even;
MPI_Comm  comm_even;
int * ranks;
int size; /* размер новой группы */

size = (p + p % 2) / 2;
ranks = (int*) malloc (size * sizeof (int));
if (!ranks) ... /* обработка ошибки */
for (i = 0; i < size; i++)
  ranks[i] = 2 * i;
```

```
MPI_Comm_group (MPI_COMM_WORLD, &group_world);
MPI_Group_incl (group_world, size, ranks, &group_even);
MPI_Comm_create (MPI_COMM_WORLD,group_even,&comm_even);
```

После окончания работы с созданным коммуникатором его необходимо удалить с помощью функции `MPI_Comm_free`. Ее прототип:

```
int MPI_Comm_free (MPI_Comm comm);
```

где входной и одновременно выходной параметр `comm` — удаляемый коммуникатор.

11.15. Пример MPI-программы, решающей задачу Дирихле для уравнения Пуассона

Рассмотрим решение задачи Дирихле для уравнения Пуассона (10.4) в двумерной области (см. раздел 10.6). Используем алгоритм, описанный в разделе 10.6.

Мы используем такое же, как в разделе 11.11, разделение $(n_1 + 1) \times (n_2 + 1)$ матриц u, f, r между процессами. Именно, процесс с номером `my_rank` из общего количества `p` работает со строками матриц с номерами

```
(n1+1)*my_rank / p, ..., (n1+1)*(my_rank + 1) / p - 1,
```

т. е. с блоком `rows x (n2+1)`, где

```
rows = (n1+1)*(my_rank + 1) / p - (n1+1)*my_rank / p.
```

Поскольку в силу выражения (10.7) для оператора A при вычислении i-й строки матрицы Au $(i = 1, 2, \ldots, n_1 - 1)$ требуется знать предыдущую и последующую строки, то в процессах с номером `my_rank` > 0 мы будем хранить предыдущую к первой строку, т. е. строку с номером

```
(n1 + 1) * my_rank / p - 1,
```

а в процессах с номером `my_rank` $<$ `p` $- 1$ мы будем хранить следующую за последней строку, т. е. строку с номером

`(n1 + 1) * (my_rank + 1) / p.`

Следовательно, распределение данных между процессами будет следующим:

- в первом процессе (`my_rank` = 0) находится блок размером `(rows + 1) x (n2 + 1)`, составленный из строк с номерами

  `(n1+1)*my_rank / p, ..., (n1+1)*(my_rank+1) / p,`

- в «средних» процессах (0 < `my_rank` < `p` − 1) находится блок `(rows + 2) x (n2 + 1)`, составленный из строк с номерами

  `(n1+1)*my_rank / p - 1, ..., (n1+1)*(my_rank+1) / p,`

- в последнем процессе (`my_rank` = `p` − 1) находится блок `(rows + 1) x (n2 + 1)`, составленный из строк с номерами

  `(n1+1)*my_rank/p - 1, ..., (n1+1)*(my_rank+1)/p - 1.`

Следовательно, во всех блоках, кроме первого, в первой строке находится предпоследняя строка предыдущего блока; во всех блоках, кроме последнего, в последней строке находится вторая строка следующего блока. При вычислении любого оператора от матрицы (т. е. вектора) это соотношение между блоками должно быть соблюдено в получившемся результате. Этого можно добиться двумя способами:

1. Вычислить ответ, находящийся в строках с номерами

   `(n1+1)*my_rank / p, ..., (n1+1)*(my_rank+1) / p - 1,`

   а затем:

   (a) при `my_rank` > 0: переслать первую из вычисленных строк (т. е. вторую в блоке) предыдущему (по номеру) процессу;

   (b) при `my_rank` < `p` − 1: переслать последнюю из вычисленных строк (т. е. предпоследнюю в блоке) следующему (по номеру) процессу.

   Этот способ мы будем использовать при вычислении оператора A в (10.7).

2. Если i-ю строку ответа можно вычислить, зная только i-ю строку аргумента (как, например, в (10.9)), то: вычислить ответ для всех строк блока. При этом значения в дублирующихся строках (т. е. в первой строке при `my_rank` > 0 и в последней строке при `my_rank` $<$ `p` $- 1$) будут вычислены дважды, но разными процессами. Если вычисления не очень сложные, то это дает выигрыш в производительности (из-за отсутствия пересылок) и упрощает программу. Мы будем использовать этот способ при вычислении оператора B в (10.9).

Для упрощения программы мы будем выделять в каждом процессе блоки памяти одинаковой длины, не учитывая, что в первом и последнем (по номеру) процессах количество хранимых строк меньше.

Файлы проекта:

- `main.c` — ввод данных и вывод результатов;
- `init.c`, `init.h` — исходный текст и соответствующий заголовочный файл для функций, инициализирующих правую часть уравнения (10.4) и вычисляющих ошибку решения;
- `operators.c`, `operators.h` — исходный текст и соответствующий заголовочный файл для функций, вычисляющих различные операторы;
- `laplace.c`, `laplace.h` — исходный текст и соответствующий заголовочный файл для функции, решающей задачу.
- `Makefile` — для сборки проекта.

Файл `main.c` по структуре похож на одноименный файл из раздела 11.11:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "init.h"
#include "laplace.h"
```

```
/* Аргументы для задачи */
typedef struct _ARGS
{
  unsigned int n1;       /* число точек по x */
  unsigned int n2;       /* число точек по y */
  double h1;             /* шаг сетки по x */
  double h2;             /* шаг сетки по y */
  unsigned int max_it;   /* максимальное число итераций */
  double prec;           /* величина падения невязки */
} ARGS;

/* Получить данные */
int get_data (char * name, ARGS *parg, int my_rank);

int
main (int argc, char **argv)
{
  int my_rank;  /* ранг текущего процесса */
  int p;        /* общее число процессов */
  double t;     /* время работы всей задачи */
  ARGS arg;     /* аргументы */

  unsigned int first_row, last_row, rows, max_rows;
  double *f;            /* правая часть */
  double *u;            /* решение */
  double *r;            /* невязка */
  int l;

  /* Начать работу с MPI */
  MPI_Init (&argc, &argv);

  /* Получить номер текущего процесса в группе всех
     процессов */
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
```

```
/* Получить общее количество запущенных процессов */
MPI_Comm_size (MPI_COMM_WORLD, &p);

/* Получить данные */
if (get_data ("a.dat", &arg, my_rank) < 0)
  {
    if (my_rank == 0)
      fprintf (stderr, "Read error!\n");
    /* Заканчиваем работу с MPI */
    MPI_Finalize ();
    return 1;
  }

/* Первая участвующая строка */
first_row = (arg.n1 + 1) * my_rank;
first_row /= p;
/* Последняя участвующая строка */
last_row = (arg.n1 + 1) * (my_rank + 1);
last_row = last_row / p - 1;
/* Количество участвующих строк */
rows = last_row - first_row + 1;

/* Вычисляем максимальное количество строк на процесс*/
max_rows = (arg.n1 + 1) / p + (arg.n1 + 1) % p + 2;

/* Выделение памяти под массивы */
l = max_rows * (arg.n2 + 1) * sizeof (double);
if (   !(f = (double*) malloc (l))
    || !(u = (double*) malloc (l))
    || !(r = (double*) malloc (l)))
  {
    fprintf (stderr, "Not enough memory!\n");
    MPI_Abort (MPI_COMM_WORLD, 1);
  }
```

```
  if (my_rank == 0)
    {
      l *= 3;
      printf ("Allocated %d bytes (%dKb or %dMb) \
of memory per process\n", l, l >> 10, l >> 20);
    }

  /* Синхронизация всех процессов */
  MPI_Barrier (MPI_COMM_WORLD);
  /* Время начала */
  t = MPI_Wtime ();
  /* Решить уравнение */
  laplace_solve (arg.n1, arg.n2, arg.h1, arg.h2,
                 arg.max_it, arg.prec, f, u, r,
                 my_rank, p);
  /* Синхронизация всех процессов */
  MPI_Barrier (MPI_COMM_WORLD);
  /* Время работы */
  t = MPI_Wtime () - t;

  /* Здесь можно работать с результатом */
  /* Воспользуемся тем, что мы знаем ответ */
  print_error (arg.n1, arg.n2, arg.h1, arg.h2, u,
               my_rank, p);

  if (my_rank == 0)
    {
      printf ("Total time = %le\n", t);
    }

  /* Освобождаем память */
  free (r);
  free (u);
  free (f);
```

```
  /* Заканчиваем работу с MPI */
  MPI_Finalize ();
  return 0;
}

/* Считать данные из файла, где находятся:
     число точек по x
     число точек по y
     длина стороны, параллельной оси x
     длина стороны, параллельной оси y
     максимальное число итераций
     величина падения невязки
   Например:
     32 32 1. 1. 1000 1e-6
 */
int get_data (char * name, ARGS *p, int my_rank)
{
  /* Количество полей в типе ARGS */
#define N_ARGS  6
  /* Структуры данных, необходимых для создания MPI-типа
     данных, соответствующего C-типу ARGS */
  /* MPI-тип каждого поля в ARGS */
  MPI_Datatype ARGS_types[N_ARGS] = { MPI_INT, MPI_INT,
    MPI_DOUBLE, MPI_DOUBLE, MPI_INT, MPI_DOUBLE };
  /* Количество элементов этого типа в каждом поле ARGS*/
  int ARGS_counts[N_ARGS] = { 1, 1, 1, 1, 1, 1 };
  /* Смещение каждого поля относительно начала
     структуры */
  MPI_Aint ARGS_disp[N_ARGS];
  /* Переменные, используемые для вычисления смещений */
  MPI_Aint start_address, address;
  /* Новый тип */
  MPI_Datatype MPI_ARGS;

  /* Читаем данные в процессе 0 */
```

```
if (my_rank == 0)
  {
    FILE *in = fopen (name, "r");
    double l1, l2;
    static ARGS zero_args; /* инициализируется 0 */

    if (!in)
      {
        fprintf (stderr, "Error opening data file %s\n",
                 name);
        *p = zero_args; /* обнулить все поля */
      }
    else
      {
        if (fscanf (in, "%u%u%lf%lf%u%lf", &p->n1,
                    &p->n2, &l1, &l2, &p->max_it,
                    &p->prec) != 6)
          {
            fprintf (stderr,
                     "Error reading data file %s\n",
                     name);
            *p = zero_args; /* обнулить все поля */
          }
        fclose (in);

        /* Вычисляем шаги сетки по направлениям x и y*/
        p->h1 = l1 / p->n1;
        p->h2 = l1 / p->n2;
      }
  }

/*Создаем MPI-тип данных, соответствующий C-типу ARGS*/
/* Вычисляем начало структуры */
MPI_Address (p, &start_address);
/* Вычисляем смещение первого поля */
```

```
  MPI_Address (&p->n1, &address);
  ARGS_disp[0] = address - start_address;
  /* Вычисляем смещение второго поля */
  MPI_Address (&p->n2, &address);
  ARGS_disp[1] = address - start_address;
  /* Вычисляем смещение третьего поля */
  MPI_Address (&p->h1, &address);
  ARGS_disp[2] = address - start_address;
  /* Вычисляем смещение четвертого поля */
  MPI_Address (&p->h2, &address);
  ARGS_disp[3] = address - start_address;
  /* Вычисляем смещение пятого поля */
  MPI_Address (&p->max_it, &address);
  ARGS_disp[4] = address - start_address;
  /* Вычисляем смещение шестого поля */
  MPI_Address (&p->prec, &address);
  ARGS_disp[5] = address - start_address;

  /* Создаем MPI-тип данных с вычисленными свойствами */
  MPI_Type_struct (N_ARGS, ARGS_counts, ARGS_disp,
                   ARGS_types, &MPI_ARGS);
  /* Делаем его доступным при обмене сообщениями */
  MPI_Type_commit (&MPI_ARGS);

  /* Рассылаем данные из процесса 0 остальным */
  MPI_Bcast (p, 1, MPI_ARGS, 0, MPI_COMM_WORLD);

  /* Тип нам больше не понадобится */
  MPI_Type_free (&MPI_ARGS);

  /* Проверяем, не было ли ошибки */
  if (p->n1 == 0 && p->n2 == 0)
    return -1;  /* ошибка */
  return 0;
}
```

Основные отличия от примера из раздела 11.11:

- По аналогии с разделом 10.6 мы использовали структуру данных для хранения всех входных параметров алгоритма. Для пересылки этой структуры создается MPI-тип данных (см. раздел 11.13.1). Это решение неэффективно для однократной пересылки и выбрано исключительно для еще одной иллюстрации создания нового MPI-типа.
- В случае ошибки чтения из файла **все** процессы будут корректно завершены с кодом ошибки, поскольку функция `get_data` вернет признак неудачи во всех процессах.
- В качестве характеристики полученного приближения выдается норма ошибки.

Заголовочный файл `init.h`:

```
void
print_error (unsigned int n1, unsigned int n2,
             double h1, double h2, double *u,
             int my_rank, int p);
void init_f (unsigned int n1, unsigned int n2,
             double h1, double h2, double *f,
             int my_rank, int p);
```

В файле `init.c` находятся функции:

- `solution` — вычисляет точное решение;
- `print_error` — вычисляет норму ошибки полученного приближения, пользуясь тем, что мы знаем точное решение; для уменьшения вычислительной погрешности используется тот же прием, что в функции `get_residual` (см. ниже);
- `right_side` — вычисляет правую часть;
- `init_f` — задает правую часть в каждом процессе.

Файл `init.c`:

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
```

```
#include "init.h"

/* Инициализация правой части задачи.
   В тестах будем рассматривать задачу в единичном
   квадрате [0,1]x[0,1] с ответом
   u(x,y) = x * (1 - x) * y * (1 - y)
   которому соответствует правая часть
   f(x,y) = 2 * x * (1 - x) + 2 * y * (1 - y)
 */

/* Точный ответ */
static double
solution (double x, double y)
{
  return x * (1 - x) * y * (1 - y);
}

/* Вычислить и напечатать L2 норму ошибки
   в процессе с номером my_rank из общего количества p.*/
void
print_error (
  unsigned int n1,      /* число точек по x */
  unsigned int n2,      /* число точек по y */
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  double *u,            /* решение */
  int my_rank,          /* номер процесса */
  int p)                /* всего процессов */
{
  unsigned int first_row, last_row, rows;
  double s1, s2, t, error;
  unsigned int i1, i2, addr;

  /* Первая участвующая строка */
  first_row = (n1 + 1) * my_rank;
```

```
  first_row /= p;
  /* Последняя участвующая строка */
  last_row = (n1 + 1) * (my_rank + 1);
  last_row = last_row / p - 1;
  /* Количество участвующих строк */
  rows = last_row - first_row + 1;

  addr = (first_row > 0 ? n2 + 1 : 0); /* начало блока */
  for (i1 = 0, s1 = 0.; i1 < rows; i1++)
    {
      for (i2 = 0, s2 = 0.; i2 <= n2; i2++)
        {
          t = u[addr++]
              - solution ((first_row + i1) * h1, i2*h2);
          s2 += t * t;
        }
      s1 += s2;
    }

  /* Сложить все s1 и передать процессу 0 */
  MPI_Reduce (&s1, &error, 1, MPI_DOUBLE, MPI_SUM, 0,
              MPI_COMM_WORLD);

  /* Напечатать ошибку */
  if (my_rank == 0)
    {
      error = sqrt (error * h1 * h2);
      printf ("Error = %le\n", error);
    }
}

/* Точная правая часть */
static double
right_side (double x, double y)
{
```

```
  return 2 * x * (1 - x) + 2 * y * (1 - y);
}

/ *Заполнить правую часть в процессе с номером my_rank из
   общего количества p. */
void init_f (
  unsigned int n1,       /* число точек по x */
  unsigned int n2,       / *число точек по y */
  double h1,             / *шаг сетки по x */
  double h2,             / *шаг сетки по y */
  double *f,             / *правая часть */
  int my_rank,           / *номер процесса */
  int p)                 / *всего процессов */
{
  unsigned int first_row, last_row, rows;
  unsigned int i1, i2, addr = 0;

  / *Первая участвующая строка */
  first_row = (n1 + 1) * my_rank;
  first_row /= p;
  / *Последняя участвующая строка */
  last_row = (n1 + 1) * (my_rank + 1);
  last_row = last_row / p - 1;
  / *Количество участвующих строк */
  rows = last_row - first_row + 1;

  / *Первая строка блока */
  if (first_row > 0)
    {
      / *в первом блоке нет дублирования данных в первой
         строке, в остальных блоках в первой строке
         находится предпоследняя строка предыдущего
         блока */
      for (i2 = 0; i2 <= n2; i2++)
        f[addr++] = right_side ((first_row - 1) * h1,
```

```
                                    i2 * h2);
    }

  /* Середина блока */
  for (i1 = 0; i1 < rows; i1++)
    for (i2 = 0; i2 <= n2; i2++)
      f[addr++] = right_side ((first_row + i1) * h1,
                              i2 * h2);

  /* Последняя строка блока */
  if (last_row < n1)
    {
      /* в последнем блоке нет дублирования данных в
         последней строке, в остальных блоках в последней
         строке находится вторая строка следующего
         блока */
      for (i2 = 0; i2 <= n2; i2++)
        f[addr++] = right_side ((last_row + 1) * h1,
                                i2 * h2);
    }
}
```

Заголовочный файл operators.h:

```
double
get_residual (unsigned int n1, unsigned int n2,
              double h1, double h2, double *r,
              int my_rank, int p);
void
get_operator (unsigned int n1, unsigned int n2,
              double h1, double h2, double *u, double *v,
              int my_rank, int p);
void
get_preconditioner (unsigned int n1, unsigned int n2,
                    double h1, double h2, double *v,
                    int my_rank, int p);
```

```
double
get_optimal_tau (unsigned int n1, unsigned int n2,
                 double h1, double h2, double *f,
                 double *u, double *r,
                 int my_rank, int p);
```

В файле `operators.c` находятся функции:

- `get_residual` – возвращает L_2 норму невязки; основные особенности функции:
  - сумма квадратов элементов вычисляется в каждом процессе для строк с номерами `(n1+1)*my_rank/p, ..., (n1+1)*(my_rank+1)/p-1`, затем с помощью `MPI_Allreduce` получается сумма квадратов всех элементов массива, которая используется для вычисления результата, возвращаемого функцией в каждом процессе;
  - граничные точки не учитываются;
  - для уменьшения вычислительной погрешности, связанной с суммированием чисел, сильно различающихся по порядку, вычисляются суммы квадратов элементов по строкам, которые затем прибавляются к окончательному результату;
- `get_operator` – вычисляет произведение матрицы A системы (10.6) (т. е. (10.5)) на вектор u: $v = Au$; оператор вычисляется согласно формулам (10.7) описанным выше способом 1; схема пересылки данных приведена на рис. 11.2:
  а) на первом этапе (рис. 11.2 а):
  - в первом процессе (`my_rank = 0`) пересылаем следующему процессу (`my_rank+1`) предпоследнюю строку, имеющую номер `rows - 1` (поскольку всего строк `rows + 1`), с помощью функции `MPI_Send`;
  - в «средних» процессах (`0 < my_rank < p - 1`) пересылаем следующему процессу (`my_rank+1`) предпоследнюю строку, имеющую номер `rows` (поскольку всего строк `rows + 2`), и получаем от предыдущего процесса

(my_rank-1) первую строку, имеющую номер 0, с помощью функции MPI_Sendrecv;

- в последнем процессе (my_rank = p − 1) получаем от предыдущего процесса (my_rank-1) первую строку, имеющую номер 0, с помощью функции MPI_Recv;

б) на втором этапе этапе (рис. 11.2 б):

- в первом процессе (my_rank = 0) получаем от следующего процесса (my_rank+1) последнюю строку, имеющую номер rows, с помощью функции MPI_Recv;
- в «средних» процессах (0 < my_rank < p − 1) пересылаем предыдущему процессу (my_rank-1) вторую строку, имеющую номер 1, и получаем от следующего про-

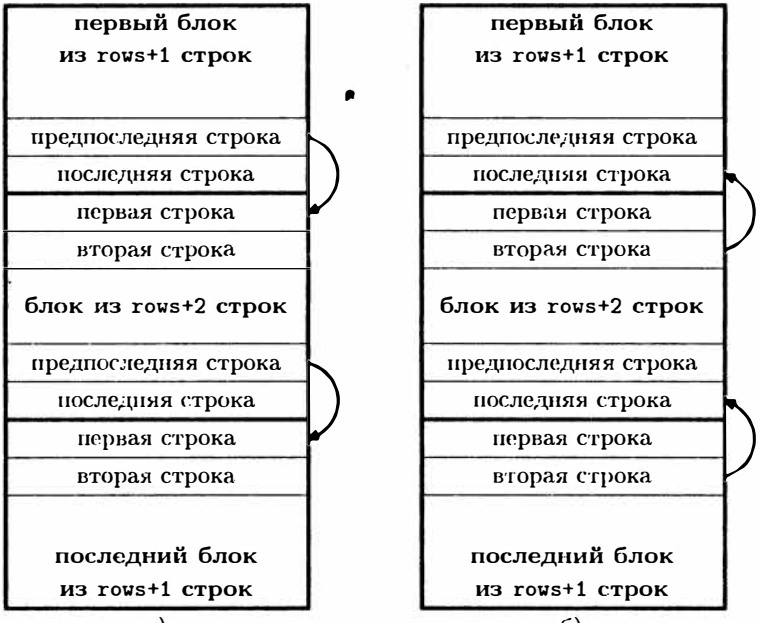


Рис. 11.2. Схема пересылки данных между процессами при вычислении оператора Лапласа

цесса (my_rank+1) последнюю строку, имеющую номер rows+1, с помощью функции MPI_Sendrecv;
- в последнем процессе (my_rank = p − 1) пересылаем предыдущему процессу (my_rank-1) вторую строку, имеющую номер 1, с помощью функции MPI_Send;

этапы не зависят друг от друга и могут выполняться в произвольном порядке;

- get_preconditioner — вычисляет произведение матрицы B^{-1} в (10.8) на вектор u: $v = B^{-1}u$; оператор вычисляется согласно формулам (10.9) для матрицы B описанным выше способом 2;
- get_optimal_tau — возвращает оптимальное значение итерационного параметра τ_k в (10.8) согласно формулам (10.10); основные особенности функции:
  - скалярные произведения (f, r), (Ar, u), (Ar, r) вычисляются в каждом процессе для строк с номерами (n1+1)*my_rank/p, ..., (n1+1)*(my_rank+1)/p-1, затем с помощью MPI_Allreduce получаются соответствующие скалярные произведения для всех элементов массива, которые используются для вычисления результата, возвращаемого функцией в каждой задаче;
  - значение Ar вычисляется в точке и сразу используется при вычислении скалярных произведений (Ar, u) и (Ar, r);
  - для уменьшения вычислительной погрешности используется тот же прием, что в get_residual.

Файл operators.c:

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#include "operators.h"

/* Вычислить L2 норму невязки
   в процессе с номером my_rank из общего количества p.
   Граничные узлы не входят. */
```

```
double
get_residual (
  unsigned int n1,       /* число точек по x */
  unsigned int n2,       /* число точек по y */
  double h1,             /* шаг сетки по x */
  double h2,             /* шаг сетки по y */
  double *r,             /* невязка */
  int my_rank,           /* номер процесса */
  int p)                 /* всего процессов */
{
  unsigned int first_row, last_row, rows;
  unsigned int i1, i2, addr;
  double s1, s2, t, residual;

  /* Первая участвующая строка */
  first_row = (n1 + 1) * my_rank;
  first_row /= p;
  /* Последняя участвующая строка */
  last_row = (n1 + 1) * (my_rank + 1);
  last_row = last_row / p - 1;

  /* В силу нулевых краевых условий на границе
     невязка = 0 */
  /* Адрес начала рассматриваемой части блока */
  addr = n2 + 1;
  if (first_row == 0)
    first_row = 1;
  if (last_row == n1)
    last_row = n1 - 1;

  /* Количество участвующих строк */
  rows = last_row - first_row + 1;

  for (i1 = 0, s1 = 0.; i1 < rows; i1++)
    {
```

```
      /* в первом элементе невязка = 0 в силу
         краевых условий */
      addr++;
      for (i2 = 1, s2 = 0.; i2 < n2; i2++)
        {
          t = r[addr++];
          s2 += t * t;
        }
      s1 += s2;
      /* в последнем элементе невязка = 0 в силу
         краевых условий */
      addr++;
    }

  /* Сложить все s1 и передать ответ всем процессам */
  MPI_Allreduce (&s1, &residual, 1, MPI_DOUBLE, MPI_SUM,
                 MPI_COMM_WORLD);

  return sqrt (residual * h1 * h2);
}

/* Вычислить v = Au, где A - оператор Лапласа,
   в процессе с номером my_rank из общего количества p.*/
void
get_operator (
  unsigned int n1,       /* число точек по x */
  unsigned int n2,       /* число точек по y */
  double h1,             /* шаг сетки по x */
  double h2,             /* шаг сетки по y */
  double *u,             /* решение */
  double *v,             /* результат */
  int my_rank,           /* номер процесса */
  int p)                 /* всего процессов */
{
  unsigned int first_row, last_row, rows;
```

```
unsigned int i1, i2, addr;
double hh1 = 1. / (h1 * h1), hh2 = 1. / (h2 * h2);
int tag = 0;
MPI_Status status;

/* Первая участвующая строка */
first_row = (n1 + 1) * my_rank;
first_row /= p;
/* Последняя участвующая строка */
last_row = (n1 + 1) * (my_rank + 1);
last_row = last_row / p - 1;
/* Количество участвующих строк в середине блока */
rows = last_row - first_row + 1;
if (first_row == 0)
  rows--;   /* первая строка будет вычислена ниже */
if (last_row == n1)
  rows--;   /* последняя строка будет вычислена ниже */

if (first_row == 0)
  {
    /* Первая строка = 0 в силу краевых условий */
    for (i2 = 0, addr = 0; i2 <= n2; i2++)
      v[addr++] = 0.;
  }
else
  {
    /* Адрес начала рассматриваемой части блока */
    addr = n2 + 1;
  }

/* Середина блока */
for (i1 = 0; i1 < rows; i1++)
  {
    /* первый элемент = 0 в силу краевых условий */
    v[addr++] = 0.;
```

```
    for (i2 = 1; i2 < n2; i2++, addr++)
      {
        /* v(i1,i2)
         =- (u(i1-1,i2)-2u(i1,i2)+u(i1+1,i2))/(h1*h1)
          - (u(i1,i2-1)-2u(i1,i2)+u(i1,i2-1))/(h2*h2)*/
        v[addr] = - (u[addr - (n2 + 1)] - 2 * u[addr]
                     + u[addr + (n2 + 1)]) * hh1
                  - (u[addr - 1] - 2 * u[addr]
                     + u[addr + 1]) * hh2;
      }
    /* последний элемент = 0 в силу краевых условий */
    v[addr++] = 0.;
  }

if (last_row == n1)
  {
    /* Последняя строка = 0 в силу краевых условий */
    for (i2 = 0; i2 <= n2; i2++)
      v[addr++] = 0.;
  }

if (first_row == 0 && last_row == n1)
  {
    /* Случай одного процесса: все сделано. */
    return;
  }

/* Количество участвующих строк */
rows = last_row - first_row + 1;

/* В первом блоке первая строка равна 0 в силу краевых
   условий, в остальных блоках в первой строке
   находится предпоследняя строка предыдущего блока.
   В последнем блоке последняя строка равна 0 в силу
   краевых условий, в остальных блоках в последней
```

```
    строке находится вторая строка следующего блока. */
  if (first_row == 0)
    {
      /* Посылаем предпоследнюю строку следующему.
         В первом блоке rows + 1 строк */
      MPI_Send (v + (rows - 1) * (n2 + 1), n2 + 1,
                MPI_DOUBLE, my_rank + 1, tag,
                MPI_COMM_WORLD);
    }
  if (last_row == n1)
    {
      /* Принимаем первую строку от предыдущего.
         В последнем блоке rows + 1 строк */
      MPI_Recv (v, n2 + 1, MPI_DOUBLE, my_rank - 1, tag,
                MPI_COMM_WORLD, &status);
    }
  if (first_row > 0 && last_row < n1)
    {
      /* Посылаем предпоследнюю строку следующему
         и принимаем первую от предыдущего.
         В блоке rows + 2 строк */
      MPI_Sendrecv (v + rows * (n2 + 1), n2 + 1,
                    MPI_DOUBLE, my_rank + 1, tag,
                    v, n2 + 1, MPI_DOUBLE, my_rank - 1,
                    tag, MPI_COMM_WORLD, &status);
    }

  if (first_row == 0)
    {
      /* Принимаем последнюю строку от следующего.
         В первом блоке rows + 1 строк */
      MPI_Recv (v + rows * (n2 + 1), n2 + 1, MPI_DOUBLE,
                my_rank + 1, tag, MPI_COMM_WORLD,
                &status);
    }
```

```
  if (last_row == n1)
    {
      /* Посылаем вторую строку предыдущему.
         В последнем блоке rows + 1 строк */
      MPI_Send (v + (n2 + 1), n2 + 1, MPI_DOUBLE,
                my_rank - 1, tag, MPI_COMM_WORLD);
    }
  if (first_row > 0 && last_row < n1)
    {
      /* Посылаем вторую строку предыдущему
         и принимаем последнюю от следующего.
         В блоке rows + 2 строк */
      MPI_Sendrecv (v + (n2 + 1), n2 + 1, MPI_DOUBLE,
                    my_rank - 1, tag,
                    v, (rows + 1) * (n2 + 1), MPI_DOUBLE
                    my_rank + 1, tag, MPI_COMM_WORLD,
                    &status);
    }
}

/* Вычислить v = B^{-1}v, где B - предобуславливатель
   для оператора Лапласа в процессе с номером my_rank
   из общего количества p.
   В качестве B используется диагональ матрицы A. */
void
get_preconditioner (
  unsigned int n1,      /* число точек по x */
  unsigned int n2,      /* число точек по y */
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  double *v,            /* аргумент/результат */
  int my_rank,          /* номер процесса */
  int p)                /* всего процессов */
{
  unsigned int first_row, last_row, rows;
```

```
unsigned int i1, i2, addr = 0;
double hh1 = 1. / (h1 * h1), hh2 = 1. / (h2 * h2);
double w = 1. / (2 * hh1 + 2 * hh2);

/* Первая участвующая строка */
first_row = (n1 + 1) * my_rank;
first_row /= p;
/* Последняя участвующая строка */
last_row = (n1 + 1) * (my_rank + 1);
last_row = last_row / p - 1;
/* Количество участвующих строк */
rows = last_row - first_row + 1;

/* Первая строка блока */
if (first_row == 0)
  {
    /* нулевые краевые условия */
    for (i2 = 0; i2 <= n2; i2++)
      v[addr++] = 0.;
    rows--;   /* первая строка уже вычислена */
  }
else /* first_row > 0 */
  {
    /* В первой строке находится предпоследняя строка
       предыдущего блока */
    for (i2 = 0; i2 <= n2; i2++)
      {
        /* v(i1,i2) = v(i1,i2)/(2/(h1*h1)+2/(h2*h2)) */
        v[addr++] *= w;
      }
  }

if (last_row == n1)
  rows--;   /* последняя строка будет вычислена ниже */
```

```
  /* Середина блока */
  for (i1 = 0; i1 < rows; i1++)
    {
      /* первый элемент = 0 в силу краевых условий */
      v[addr++] = 0.;
      for (i2 = 1; i2 < n2; i2++)
        {
          /* v(i1,i2) = v(i1,i2)/(2/(h1*h1)+2/(h2*h2)) */
          v[addr++] *= w;
        }
      /* последний элемент = 0 в силу краевых условий */
      v[addr++] = 0.;
    }

  /* Последняя строка блока */
  if (last_row == n1)
    {
      /* нулевые краевые условия */
      for (i2 = 0; i2 <= n2; i2++)
        v[addr++] = 0.;
    }
  else /* last_row < n1 */
    {
      /* В последней строке находится вторая строка
         следующего блока */
      for (i2 = 0; i2 <= n2; i2++)
        {
          /* v(i1,i2) = v(i1,i2)/(2/(h1*h1)+2/(h2*h2)) */
          v[addr++] *= w;
        }
    }
}

/* Вычислить ((f, r) - (Ar, u)) / (Ar, r)
   где A - оператор Лапласа,
```

```
   в процессе с номером my_rank из общего количества p.
   Функции u, r удовлетворяют нулевым краевым условиям */
double
get_optimal_tau (
  unsigned int n1,      /* число точек по x */
  unsigned int n2,      /* число точек по y */
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  double *f,            /* правая часть */
  double *u,            /* решение */
  double *r,            /* невязка */
  int my_rank,          /* номер процесса */
  int p)                /* всего процессов */
{
  unsigned int first_row, last_row, rows;
  unsigned int i1, i2, addr;
  double hh1 = 1. / (h1 * h1), hh2 = 1. / (h2 * h2);
  double s1_fr, s2_fr;  /* суммы для (f, r)    */
  double s1_Aru, s2_Aru;/* суммы для (Ar, u)   */
  double s1_Arr, s2_Arr;/* суммы для (Ar, r)   */
  double Ar;            /* значение Ar в точке */
  double a[3], b[3];

  /* Первая участвующая строка */
  first_row = (n1 + 1) * my_rank;
  first_row /= p;
  /* Последняя участвующая строка */
  last_row = (n1 + 1) * (my_rank + 1);
  last_row = last_row / p - 1;

  /* В силу нулевых краевых условий на границе,
     слагаемые, соответствующие граничным узлам = 0 */
  /* Адрес начала рассматриваемой части блока */
  addr = n2 + 1;
  if (first_row == 0)
```

```
  first_row = 1;
if (last_row == n1)
  last_row = n1 - 1;

/* Количество участвующих строк в середине блока */
rows = last_row - first_row + 1;

for (i1 = 0, s1_fr = 0., s1_Aru = 0., s1_Arr = 0.;
     i1 < rows; i1++)
  {
    /* в первом элементе слагаемые = 0 в силу
       краевых условий */
    addr++;
    for (i2 = 1, s2_fr = 0., s2_Aru = 0., s2_Arr = 0.;
         i2 < n2; i2++, addr++)
      {
        /* Вычисляем (f, r) */
        s2_fr += f[addr] * r[addr];
        /* Вычисляем Ar */
        Ar = - (r[addr - (n2 + 1)] - 2 * r[addr]
                + r[addr + (n2 + 1)]) * hh1
             - (r[addr - 1] - 2 * r[addr]
                + r[addr + 1]) * hh2;
        /* Вычисляем (Ar, u) */
        s2_Aru += Ar * u[addr];
        /* Вычисляем (Ar, r) */
        s2_Arr += Ar * r[addr];
      }
    s1_fr  += s2_fr;
    s1_Aru += s2_Aru;
    s1_Arr += s2_Arr;
    /* в последнем элементе слагаемые = 0 в силу
       краевых условий */
    addr++;
  }
```

```
  a[0] = s1_fr;
  a[1] = s1_Aru;
  a[2] = s1_Arr;

  /* Вычислить суммы по всем процессам */
  MPI_Allreduce (a, b, 3, MPI_DOUBLE, MPI_SUM,
                 MPI_COMM_WORLD);

  /* Вычислить ответ */
  return (b[0] - b[1]) / b[2];
}
```

Заголовочный файл `laplace.h`:

```
int laplace_solve (unsigned int n1, unsigned int n2,
  double h1, double h2, unsigned int max_it, double prec,
  double *f, double *u, double *r, int my_rank, int p);
```

В файле `laplace.c` находится функция `laplace_solve`, осуществляющая итерационный процесс (10.8), (10.10) с матрицами (10.7) и (10.9) по описанным в разделе 10.6 расчетным формулам. Файл `laplace.c`:

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#include "init.h"
#include "operators.h"
#include "laplace.h"


/* Решить задачу в процессе с номером my_rank из общего
   количества p.
   Возвращает количество потребовавшихся итераций. */
int laplace_solve (
  unsigned int n1,      /* число точек по x */
  unsigned int n2,      /* число точек по y */
```

```
  double h1,            /* шаг сетки по x */
  double h2,            /* шаг сетки по y */
  unsigned int max_it,  /* максимальное число итераций */
  double prec,          /* величина падения невязки */
  double *f,            /* правая часть */
  double *u,            /* решение */
  double *r,            /* невязка */
  int my_rank,          /* номер процесса */
  int p)                /* всего процессов */
{
  unsigned int first_row, last_row, rows, width, len;
  unsigned int addr;
  /* Норма невязки на первом шаге */
  double norm_residual_1;
  /* Норма невязки на очередном шаге */
  double norm_residual;
  /* Норма невязки на предыдущем шаге */
  double norm_residual_prev;
  unsigned int it;      /* Номер итерации */
  double tau;           /* Итерационный параметр */

  /* Первая участвующая строка */
  first_row = (n1 + 1) * my_rank;
  first_row /= p;
  /* Последняя участвующая строка */
  last_row = (n1 + 1) * (my_rank + 1);
  last_row = last_row / p - 1;
  /* Количество участвующих строк */
  rows = last_row - first_row + 1;
  /* Первый и последний блоки имеют по rows + 1 строк,
     остальные по rows + 2 */
  width = rows + 2;
  if (first_row == 0)
    width--;
  if (last_row == n1)
```

```
  width--;
/* Длина блока */
len = width * (n2 + 1);

/* Инициализируем правую часть */
init_f (n1, n2, h1, h2, f, my_rank, p);

/* Начальное приближение = 0 */
for (addr = 0; addr < len; addr++)
  u[addr] = 0.;

/* Невязка при нулевом начальном приближении = f */
norm_residual_1 = norm_residual = norm_residual_prev
  = get_residual (n1, n2, h1, h2, f, my_rank, p);

if (my_rank == 0)
  printf ("Residual = %le\n", norm_residual_1);

/* Итерационный процесс */
for (it = 1; it < max_it; it++)
  {
    /* Шаг итерационного процесса */

    /* Вычисляем r = Au */
    get_operator (n1, n2, h1, h2, u, r, my_rank, p);

    /* Вычисляем невязку r = f - r = f - Au */
    for (addr = 0; addr < len; addr++)
      r[addr] = f[addr] - r[addr];

    /* Вычисляем норму невязки */
    norm_residual = get_residual (n1, n2, h1, h2, r,
                                  my_rank, p);
    if (my_rank == 0)
      printf ("It # =%2.2d, residual=%11.4e, \
```

```
convergence=%6.3f, average convergence=%6.3f\n",
          it, norm_residual,
          norm_residual / norm_residual_prev,
          pow (norm_residual / norm_residual_1, 1./it));

      /* Проверяем условие окончания процесса */
      if (norm_residual < norm_residual_1 * prec)
        break;

      /* Обращение предобуславливателя r = B^{-1} r */
      get_preconditioner (n1, n2, h1, h2, r, my_rank, p);

      /* Вычисление итерационного параметра
         tau = ((b, r) - (Ar, u)) / (Ar, r) */
      tau = get_optimal_tau (n1, n2, h1, h2, f, u, r,
                             my_rank, p);

      /* Построение очередного приближения u+=tau * r */
      for (addr = 0; addr < len; addr++)
        u[addr] += tau * r[addr];

      norm_residual_prev = norm_residual;
    }

  return it;
}
```

Источники дополнительной информации

Этот раздел является аналогом раздела «список литературы», а несколько необычное название связано с тем, что практически вся перечисленная ниже документация существует только в электронном виде.

Основным источником дополнительной информации о всех описанных функциях, их аргументах, поведении в различных ситуациях, о связи с другими функциями можно получить из установленной в UNIX справочной системы (man pages). Для этого надо набрать команду:

```
man <имя функции>
```

или

```
man -a <имя функции>
```

если имя встречается в нескольких разделах справочника (тогда будут выданы все вхождения).

Для того чтобы по команде `man` можно было получать информацию о MPI-функциях, необходимо установить на компьютере подсистему MPI. Существует несколько свободно распространяемых реализаций MPI.

1. Реализация `mpich`, разработанная Argonne National Lab и Mississippi State University, доступна по адресу

   ```
   ftp://info.mcs.anl.gov/pub/mpi
   ```

2. Реализация LAM, разработанная Ohio Supercomputer Center, доступна по адресу

   `ftp://ftp.osc.edu/pub/lam`

3. Реализация CHIMP, разработанная Edinburgh Parallel Computing Center, доступна по адресу

   `ftp://ftp.epcc.ed.ac.uk/pub/chimp/release`

Стандарт MPI версии 1 доступен по адресу

`ftp://ftp.netlib.org`

как Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5, 1994. Стандарт MPI версии 1.1 доступен по адресу

`ftp://ftp.mcs.anl.gov`

Программа курса

1. CISC- и RISC-процессоры. Основные черты RISC-архитектуры.
2. Повышение производительности процессоров за счет конвейеризации. Условия оптимального функционирования конвейера.
3. Суперконвейерные и суперскалярные процессоры. Выделение независимо работающих устройств: IU, FPU, MMU, BU.
4. Методы уменьшения негативного влияния инструкций перехода на производительность процессора.
5. Повышение производительности процессоров за счет введения кэш-памяти. Кэши: единый, Гарвардский, с прямой записью, с обратной записью.
6. Организация кэш-памяти. Алгоритмы замены данных в кэш-памяти. Специальные кэши.
7. Согласование кэшей в мультипроцессорных системах с общей памятью.
8. Виды многопроцессорных архитектур.
9. Поддержка многозадачности и многопроцессорности специальными инструкциями процессора. Организация данных во внешней памяти.
10. Программа, процессор, процесс. Основные составляющие процесса, состояния процесса. Стек, виртуальная память, механизмы трансляции адреса.

11. Механизмы взаимодействия процессов. Разделяемая память, семафоры, сигналы, почтовые ящики, события. Задачи (threads). Сравнение с процессами. Ресурсы, приоритеты. Параллельные процессы. Связывание. Статическое и динамическое связывание.
12. Виды ресурсов: аппаратные, программные, активные, пассивные, локальные, разделяемые, постоянные, временные, некритичные, критичные.
13. Типы взаимодействия процессов: сотрудничающие и конкурирующие процессы. Критические секции, взаимное исключение процессов (задач).
14. Проблемы, возникающие при синхронизации задач и идеи их разрешения.
15. Состояния процесса и механизмы перехода из одного состояния в другое.
16. Стандарты на UNIX-системы.
17. Управление процессами. Функции `fork`, `execl`, `execv`, `waitpid`. Примеры использования.
18. Работа с сигналами. Функция `signal`. Пример использования.
19. Разделяемая память. Функции `shmget`, `shmat`, `shmctl`. Примеры использования.
20. Семафоры. Функции `semget`, `semop`, `semctl`. Примеры использования.
21. События. Примитивные операции. Организация взаимодействия клиент–сервер с помощью событий.
22. Очереди сообщений. Функции `msgget`, `msgsnd`, `msgrcv`, `msgctl`. Примеры использования.
23. Управление задачами (threads). Функции `pthread_create`, `pthread_join`, `sched_yield`. Примеры использования.
24. Объекты синхронизации типа mutex. Функции `pthread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`, `pthread_mutex_destroy`. Примеры использования.

25. Объекты синхронизации типа condvar. Функции `pthread_cond_init`, `pthread_cond_signal`, `pthread_cond_broadcast`, `pthread_cond_wait`, `pthread_cond_destroy`. Примеры использования.
26. Пример multithread-программы умножения матрицы на вектор.
27. Message Passing Interface (MPI). Общая структура MPI-программы. Функции `MPI_Init`, `MPI_Finalize`. Сообщения и их виды.
28. Коммуникаторы. Функции `MPI_Comm_size`, `MPI_Comm_rank`. Примеры использования.
29. Попарный обмен сообщениями. Функции `MPI_Send`, `MPI_Recv`. Примеры использования.
30. Операции ввода–вывода в MPI-программах. Примеры.
31. Дополнительные возможности для попарного обмена сообщениями. Функции `MPI_Sendrecv`, `MPI_Sendrecv_replace`.
32. Дополнительные возможности для попарного обмена сообщениями. Функции `MPI_Isend`, `MPI_Irecv`, `MPI_Test`, `MPI_Testany`, `MPI_Wait`, `MPI_Waitany`
33. Коллективный обмен сообщениями. Функции `MPI_Barrier`, `MPI_Abort`, `MPI_Bcast`.
34. Коллективный обмен сообщениями. Функции `MPI_Reduce`, `MPI_Allreduce`, `MPI_Op_create`, `MPI_Op_free`. Пример MPI-программы, вычисляющей определенный интеграл.
35. Время в MPI-программах. Функции `MPI_Wtime`, `MPI_Wtick`. Пример использования.
36. Пример MPI-программы умножения матрицы на вектор.
37. Дополнительные возможности для коллективного обмена массивами данных. Функции `MPI_Gather`, `MPI_Allgather`, `MPI_Scatter`,
38. Пересылка структур данных. Создание нового MPI-типа данных с помощью `MPI_Type_struct`. Функции `MPI_Address`, `MPI_Type_commit`, `MPI_Type_free`. Пример использования.

39. Упаковка/распаковка разнородных данных для блочной пересылки с помощью функций `MPI_Pack`, `MPI_Unpack`. Функция `MPI_Pack_size`. Пример использования.
40. Гетерогенные и однородные вычислительные установки. Пересылка структур данных в однородных параллельных ЭВМ. Примеры.
41. Пересылки строк и столбцов матриц. Создание нового MPI-типа данных с помощью `MPI_Type_vector`. Пример использования.
42. Ограничение коллективного обмена на подмножество процессов. Функции `MPI_Comm_group`, `MPI_Group_incl`, `MPI_Comm_create`, `MPI_Comm_free`. Примеры использования.

Список задач

1. Написать программу, вычисляющую сумму элементов n последовательностей вещественных чисел, находящихся в n файлах, имена которых заданы массивом a. Ответ должен быть записан в файл `res.txt`. Программа запускает n процессов, вычисляющих ответ для каждого из файлов и прибавляющих его к результату, находящемуся в выходном файле. Взаимное исключение при доступе к файлу обеспечивается с помощью семафора.
2. Написать функцию, получающую в качестве аргументов целое число n и массив длины n с именами файлов, содержащих единую последовательность вещественных чисел неизвестной длины, и возвращающую количество участков постоянства этой последовательности. Функция возвращает -1, -2 и т. д., если она не смогла открыть какой-либо файл, прочитать элемент и т. д. Функция должна запускать n процессов, обрабатывающих свой файл и передающих результаты в основной процесс через очередь сообщений для формирования ответа для последовательности в целом.
3. Написать программу, осуществляющую мониторинг и перезапуск в случае завершения работы заданного количества приложений. Приложения задаются массивом строк, являющихся их полным путевым именем, и не имеют аргументов.
4. Написать реализацию стека строк в разделяемой памяти. При запуске программа создает блок разделяемой памяти

(если его еще нет) или присоединяется к существующему блоку. Программа должна обеспечивать основные функции работы со стеком (добавить и удалить элемент) и возможность запуска себя во многих экземплярах.

5. Написать реализацию набора конфигурационных параметров для многих одновременно работающих экземпляров программы. Набор параметров задается некоторой структурой данных и хранится в разделяемой памяти. Блок разделяемой памяти создается при запуске первого экземпляра программы и заполняется из файла. Перед окончанием работы последнего экземпляра набор параметров сохраняется в файле, а блок разделяемой памяти удаляется. Программа должна обеспечивать основные функции работы с набором параметров (прочитать и изменить элемент), причем в случае изменения данных одним из экземпляров он оповещает все остальные экземпляры с помощью сигнала.
6. Написать multithread-функцию, получающую в качестве аргументов $n \times n$ массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и возвращающую ненулевое значение, если массив a симметричен (т. е. $a_{ij} = a_{ji}$), 0 в противном случае. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту функцию и выводить на экран результат ее работы.
7. Написать multithread-подпрограмму, получающую в качестве аргументов $n \times n$ массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую матрицу a на ее транспонированную. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

8. Написать multithread-подпрограмму, получающую в качестве аргументов $n \times n$ массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую матрицу a на матрицу $(a + a^t)/2$, где a^t — транспонированная матрица a. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.
9. Написать multithread-подпрограмму, получающую в качестве аргументов массив a вещественных чисел, целое число n, являющееся длиной этого массива, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент массива (для которого это возможно) на среднее арифметическое соседних элементов. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.
10. Написать multithread-подпрограмму, получающую в качестве аргументов $n \times n$ массив a вещественных чисел, вспомогательный массив b вещественных чисел длины n (в каждом thread свой), целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+1,j} + a_{i-1,j} + a_{i,j+1} + a_{i,j-1} - 4a_{i,j}$. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданным формулам), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.
11. Написать multithread-подпрограмму, получающую в качестве аргументов $n \times n$ массив a вещественных чисел, вспомогательный массив b вещественных чисел длины $2n$ (в каж-

дом thread свой), целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+2,j} + a_{i-2,j} + a_{i,j+2} + a_{i,j-2} - 4a_{i,j}$. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданным формулам), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

12. Написать MPI-функцию, получающую в качестве аргументов соответствующую часть (блок) $n \times n$ массива a вещественных чисел, целое число n, номер процесса k, общее количество процессов p, и возвращающую ненулевое значение, если массив a симметричен (т. е. $a_{ij} = a_{ji}$), 0 в противном случае. При этом должна быть обеспечена равномерная загрузка всех процессов. Основная программа должна начинать работу с MPI, вводить число n и массив a (из файла или по заданной формуле), вызывать эту функцию и выводить на экран результат ее работы.
13. Написать MPI-подпрограмму, получающую в качестве аргументов соответствующую часть (блок) $n \times n$ массива a вещественных чисел, целое число n, номер процесса k, общее количество процессов p, и заменяющую матрицу a на ее транспонированную. При этом должна быть обеспечена равномерная загрузка всех процессов. Основная программа должна начинать работу с MPI, вводить число n и массив a (из файла или по заданной формуле), вызывать эту подпрограмму и выводить на экран результат ее работы.
14. Написать MPI-подпрограмму, получающую в качестве аргументов соответствующую часть (блок) $n \times n$ массива a вещественных чисел, целое число n, номер процесса k, общее количество процессов p, и заменяющую матрицу a на матрицу $(a + a^t)/2$, где a^t — транспонированная матрица a. При этом должна быть обеспечена равномерная загрузка всех процессов. Основная программа должна начинать работу с MPI. вводить число n и массив a (из файла или по

заданной формуле), вызывать эту подпрограмму и выводить на экран результат ее работы.

15. Написать MPI-подпрограмму, получающую в качестве аргументов соответствующую часть (блок) массива a вещественных чисел, целое число n, являющееся длиной этого массива, номер процесса k, общее количество процессов p, и заменяющую каждый элемент массива (для которого это возможно) на среднее арифметическое соседних элементов. При этом должна быть обеспечена равномерная загрузка всех процессов. Основная программа должна начинать работу с MPI, вводить число n и массив a (из файла или по заданной формуле), вызывать эту подпрограмму и выводить на экран результат ее работы.
16. Написать MPI-подпрограмму, получающую в качестве аргументов соответствующие части (блоки) $n \times n$ массива a вещественных чисел, вспомогательный массив b вещественных чисел длины n, целое число n, номер процесса k, общее количество процессов p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+1,j} + a_{i-1,j} + a_{i,j+1} + a_{i,j-1} - 4a_{i,j}$. При этом должна быть обеспечена равномерная загрузка всех процессов. Основная программа должна начинать работу с MPI, вводить число n и массив a (из файла или по заданной формуле), вызывать эту подпрограмму и выводить на экран результат ее работы.
17. Написать MPI-подпрограмму, получающую в качестве аргументов соответствующие части (блоки) $n \times n$ массива a вещественных чисел, вспомогательный массив b вещественных чисел длины $2n$, целое число n, номер процесса k, общее количество процессов p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+2,j} + a_{i-2,j} + a_{i,j+2} + a_{i,j-2} - 4a_{i,j}$. При этом должна быть обеспечена равномерная загрузка всех процессов. Основная программа должна начинать работу с MPI, вводить число n и массив a (из файла или по заданной формуле), вызывать эту подпрограмму и выводить на экран результат ее работы.

Указатель русских терминов

Указатель английских терминов

Список иллюстраций

Список таблиц

Список примеров