Синхронизация процессов

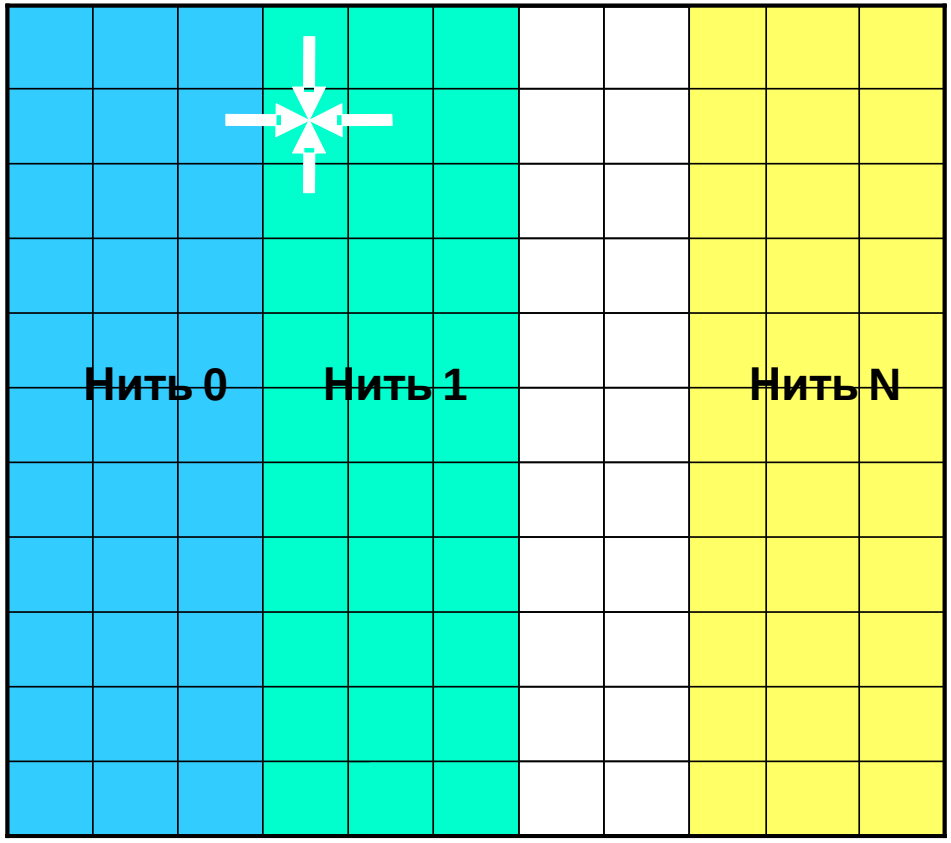
Метод последовательной верхней релаксации (SOR)

```
float  A[ L1 ][ L2 ];

 for(int i = 1; i < L1-1; i++)
    for(int j = 1; j < L2-1; j++)
       A[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1]) / 4;
```

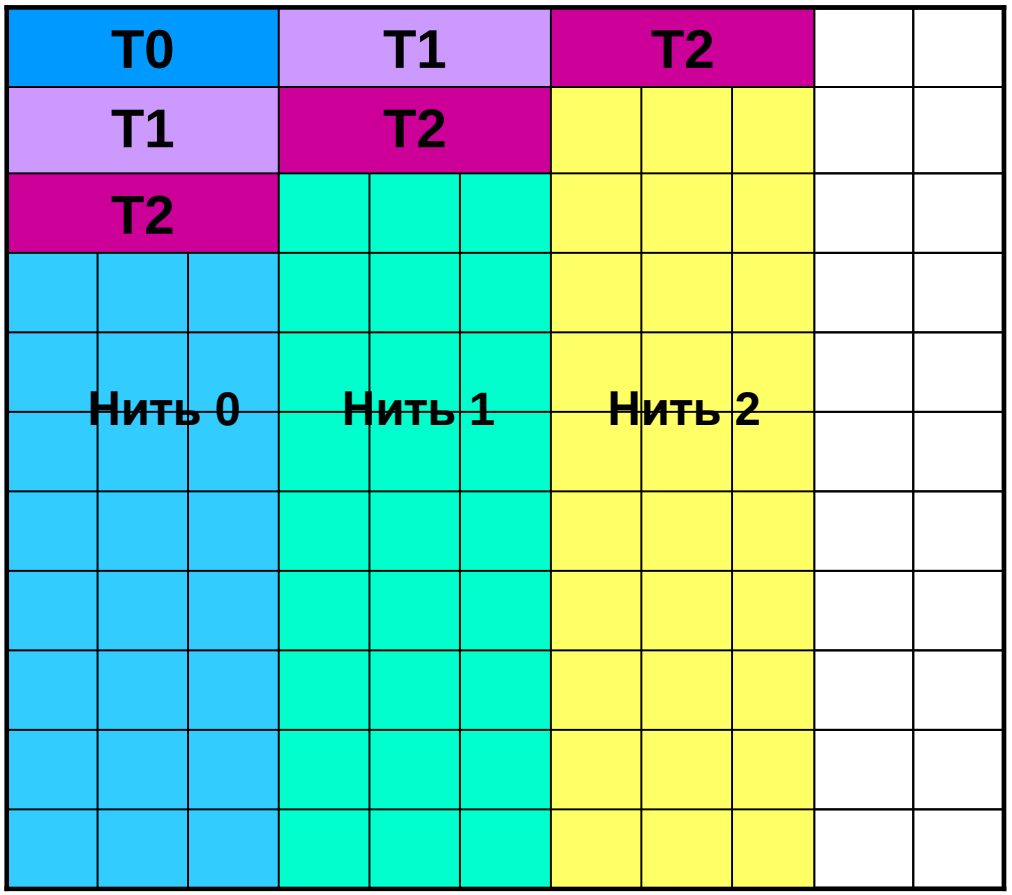
Метод последовательной верхней релаксации (SOR)

```
for(int i = 1; i < L1-1; i++)
    for(int j = 1; j < L2-1; j++)
      A[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1]) / 4;
```



Метод последовательной верхней релаксации (SOR)

```
for(int i = 1; i < L1-1; i++)
  for(int j = 1; j < L2-1; j++)
    A[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1]) / 4;
```



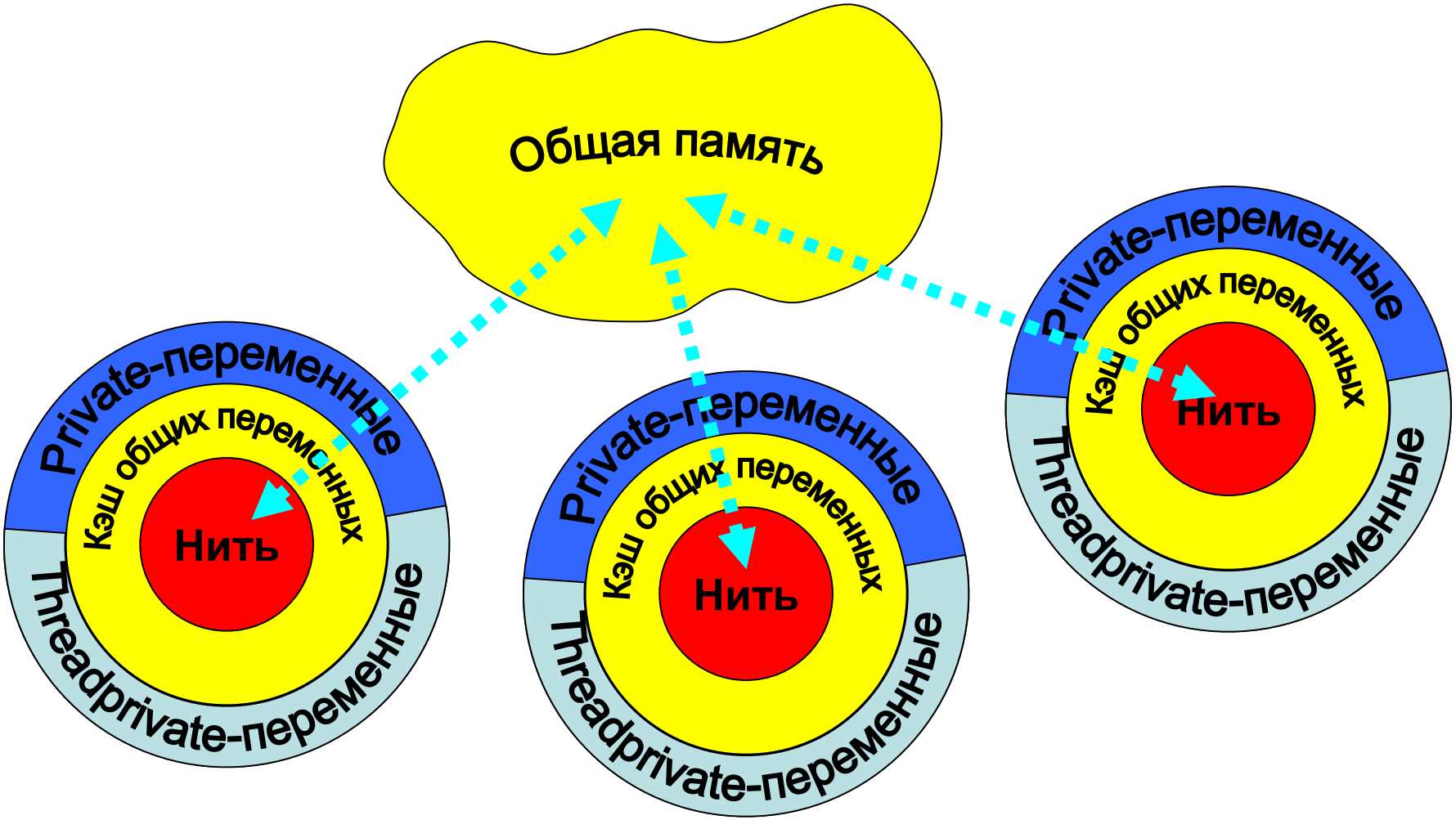
Метод последовательной верхней релаксации (SOR)

```
int iam, numt, limit;
int sync[NUM_THREADS];
#pragma omp parallel
private(iam,numt,limit)
  {
  iam = omp_get_thread_num ();
  numt = omp_get_num_threads ();
  limit=min(numt-1,L2-3);
  isync[iam]=0;
#pragma omp barrier
  for (int i=1; i<L1-1; i++) {
    if ((iam>0) && (iam<=limit)) {
       for (;isync[iam-1]==0;);
       isync[iam-1]=0;
}
#pragma omp for schedule(static) nowait
    for (int j=1; j<L2-1; j++) {
      A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] +
               A[i][j+1])/4;
    }
    if (iam<limit) {
       isync[iam]=1;
    }
  }
}
```

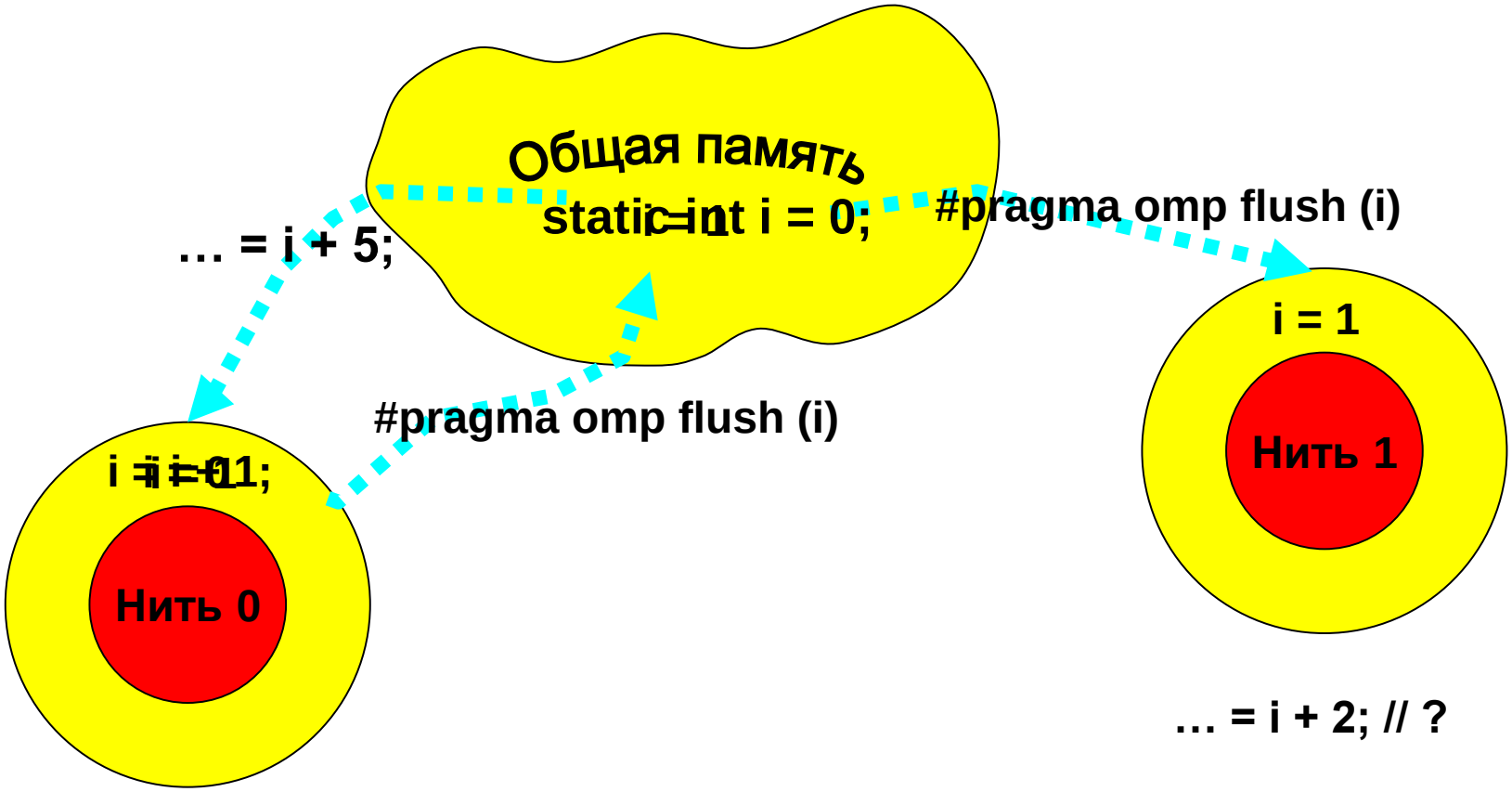
Метод последовательной верхней релаксации (SOR)

```
int iam, numt, limit;
int sync[NUM_THREADS];
#pragma omp parallel
private(iam,numt,limit)
  {
  iam = omp_get_thread_num ();
  numt = omp_get_num_threads ();
  limit=min(numt-1,L2-3);
  isync[iam]=0;
#pragma omp barrier
  for (int i=1; i<L1-1; i++) {
    if ((iam>0) && (iam<=limit)) {
       for (;isync[iam-1]==0;);
       isync[iam-1]=0;
}
#pragma omp for schedule(static) nowait
    for (int j=1; j<L2-1; j++) {
      A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] +
               A[i][j+1])/4;
    }
    if (iam<limit) {
       for (;isync[iam]==1;);
       isync[iam]=1;
    }
  }
}
```

Модель памяти в OpenMP



Модель памяти в OpenMP



Консистентность памяти в OpenMP

Корректная последовательность работы нитей с переменной:

- Нить0 записывает значение переменной – write (var)
- Нить0 выполняет операцию синхронизации – flush (var)
- Нить1 выполняет операцию синхронизации – flush (var)
- Нить1 читает значение переменной – read (var)

```
A = 1
flush(A)
. . .
flush(A)
... = A
```

Консистентность памяти в OpenMP

#pragma omp flush *[(список переменных)]*

По умолчанию все переменные приводятся в консистентное состояние **(#pragma omp flush**):

- при барьерной синхронизации;
- при входе и выходе из конструкций **parallel**, **critical** и **ordered;**
- при выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**), если не указана клауза **nowait;**
- при вызове **omp_set_lock** и **omp_unset_lock;**
- при вызове **omp_test_lock**, **omp_set_nest_lock**, **omp_unset_nest_lock** и **omp_test_nest_lock**, если изменилось состояние семафора.

При входе и выходе из конструкции **atomic** выполняется **#pragma omp flush(x)**, где x – переменная, изменяемая в конструкции **atomic**.

Консистентность памяти в OpenMP

1. Если пересечение множеств переменных, указанных в операциях flush, выполняемых различными нитями не пустое, то результат выполнения операций flush будет таким, как если бы эти операции выполнялись в некоторой последовательности (единой для всех нитей).
2. Если пересечение множеств переменных, указанных в операциях flush, выполняемых одной нитью не пустое, то результат выполнения операций flush, будет таким, как если бы эти операции выполнялись в порядке определяемом программой.
3. Если пересечение множеств переменных, указанных в операциях flush, пустое, то операции flush могут выполняться независимо (в любом порядке).

Метод последовательной верхней релаксации (SOR)

```
int iam, numt, limit;
int sync[NUM_THREADS];
#pragma omp parallel
private(iam,numt,limit)
  {
  iam = omp_get_thread_num ();
  numt = omp_get_num_threads ();
  limit=min(numt-1,L2-3);
  isync[iam]=0;
#pragma omp barrier
  for (int i=1; i<L1-1; i++) {
    if ((iam>0) && (iam<=limit)) {
       for (;isync[iam-1]==0;) {
          #pragma omp flush (isync)
       }
       isync[iam-1]=0;
       #pragma omp flush (isync)
    }
#pragma omp for schedule(static) nowait
    for (int j=1; j<L2-1; j++) {
      A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] +
               A[i][j+1])/4;
    }
    if (iam<limit) {
       for (;isync[iam]==1;) {
          #pragma omp flush (isync)
       }
       isync[iam]=1;
       #pragma omp flush (isync)
     }
  }
}
```

Метод последовательной верхней релаксации (SOR)

```
#pragma omp parallel
  {
    int iam = omp_get_thread_num ();
    int numt = omp_get_num_threads ();
    for (int newi=1; newi<L1 -1 + numt - 1; newi++) {
      int i = newi - iam;
      #pragma omp for
        for (int j=1; j<L2 - 1; j++) {
          if ((i >= 1) && (i< L1-1)) {
            A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1])/4;
          }
        }
    }
  }
```

Метод последовательной верхней релаксации (SOR)

```
#pragma omp parallel for ordered(2) shared(a)
 for (int i=1; i<L1-1; i++)
     for (int j=1; j<L2-1; j++) {
         #pragma omp ordered depend (sink: i - 1, j) depend (sink: i, j - 1)
             A[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4;
         #pragma omp ordered depend (source)
     }

    /* OpenMP 4.5*/
```

Механизм событий

События – это переменные, показывающие, что произошли определенные события.

Для объявления события служит оператор

POST(имя переменной),

для ожидания события –

WAIT (имя переменной),

для чистки (присваивания нулевого значения) - оператор

CLEAR(имя переменной).

Варианты реализации - не хранящие информацию (по оператору POST из ожидания выводятся только те процессы, которые уже выдали WAIT), однократно объявляемые (нет оператора чистки).

Метод последовательной верхней релаксации (SOR)

```
float  A[ L1 ][ L2 ];
struct event  s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
    for ( j = 0; j < L2; j++)   { clear( s[ i ][ j ]) };
for ( j = 0; j < L2; j++)   { post( s[ 0 ][ j ]) };
.............
.............
parfor ( i = 1; i < L1-1; i++)
    for ( j = 1; j < L2-1; j++)
    {
        wait( s[ i-1 ][ j ]);
        A[ i ][ j ] = (A[ i-1 ][ j ] + A[ i+1][ j ] + A[ i ][ j-1 ] + A[ i ][ j+1 ]) / 4;
        post( s[ i ][ j ]);
    }
```

Метод последовательной верхней релаксации (SOR)

```
float  A[ L1 ][ L2 ];
semaphore  s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
    for ( j = 0; j < L2; j++)   { P( s[ i ][ j ]) };
for ( j = 0; j < L2; j++)   { V( s[ 0 ][ j ]) };
.............
.............
parfor ( i = 1; i < L1-1; i++)
    for ( j = 1; j < L2-1; j++)
    {
            P( s[ i-1 ][ j ]);
            A[ i ][ j ] = (A[ i-1 ][ j ] + A[ i+1][ j ] + A[ i ][ j-1 ] + A[ i ][ j+1 ]) / 4;
            V( s[ i ][ j ]);
    }
```

Метод последовательной верхней релаксации (SOR)

```
float  A[ L1 ][ L2 ];
struct event  s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
    for ( j = 0; j < L2; j++)   { clear( s[ i ][ j ]) };
for ( j = 0; j < L2; j++)   { post( s[ 0 ][ j ]) };
for ( i = 0; i < L1; i++)   { post( s[ i ][ 0 ]) };
.............
.............
parfor ( i = 1; i < L1-1; i++)
    parfor ( j = 1; j < L2-1; j++)
    {
            wait( s[ i-1 ][ j ]);
            wait( s[ i ][ j-1 ]);
            A[ i ][ j ] = (A[ i-1 ][ j ] + A[ i+1][ j ] + A[ i ][ j-1 ] + A[ i ][ j+1 ]) / 4;
            post( s[ i ][ j ]);
    }
```

Обмен сообщениями (message passing)

- **Хоар** (Xoare) 1978 год, "Взаимодействующие последовательные процессы". Цели - избавиться от проблем разделения памяти и предложить модель взаимодействия процессов для распределенных систем.

  send (destination, &message, msize);

  receive ([source], &message, msize);

- Адресат - процесс.
- Отправитель - может не специфицироваться (любой).
- С буферизацией (почтовые ящики) или нет (рандеву - Ада, Оккам).

  Пайпы ОС UNIX - почтовые ящики, заменяют файлы и не хранят границы сообщений (все сообщения объединяются в одно большое, которое можно читать произвольными порциями).

Производитель-потребитель

```
semaphore  mutex = 1;
semaphore  full = 0;
semaphore  empty = N;
```

```
producer()
{
    int item;
    while (TRUE)
    {
      produce_item(&item);
      P(empty);
      P(mutex);
      enter_item(item);
      V(mutex);
      V(full)

    }
}
```

```
consumer()
{
      int item;
      while (TRUE)
      {

         P(full);
         P(mutex);
         remove_item(&item);
         V(mutex);
         V(empty);
         consume_item(item);
      }
 }
```

Производитель-потребитель

```
#define  N   100                    /* максимальное число сообщений */
#define  msize  4                   /* размер сообщения*/
typedef  int message[msize];
```

```
producer()
{
    message m;
    int item;

    while (TRUE)
    {
      produce_item(&item);
      receive(consumer, &m, msize);
      build_message(&m, item);
      send(consumer, &m, msize);
     }
 }
```

```
consumer()
{
    message m;
    int item, i;

    for (i = 0; i < N; i ++)
        send (producer, &m, msize);
        while (TRUE)
        {
            receive(producer, &m, msize);
            extract_item(&m, item);
            send(producer, &m, msize);
            consume_item(item);
        }
 }
```

producer() AND consumer() **/* запустили 2 процесса */**