Управление процессами

3. Взаимодействие процессов: синхронизация, тупики
   - 3.1. Разделение ресурсов
   - 3.2. Взаимное исключение
     - 3.2.1. Проблемы реализации взаимного исключения
     - 3.2.2. Способы реализации взаимного исключения
       - 3.2.2.1. Семафоры Дейкстры
       - 3.2.2.2. Мониторы
       - 3.2.2.3. Обмен сообщениями
   - 3.3. Примеры

Параллельные процессы

Параллельные процессы — процессы, выполнение (обработка) которых хотя бы частично перекрывается по времени.

- **Независимые процессы** используют независимое множество ресурсов
- **Взаимодействующие процессы** используют ресурсы совместно, и выполнение одного процесса может оказать влияние на результат другого

Разделение ресурсов

Разделение ресурса — совместное использование несколькими процессами ресурса ВС.

Критические ресурсы — разделяемые ресурсы, которые должны быть доступны в текущий момент времени только одному процессу.

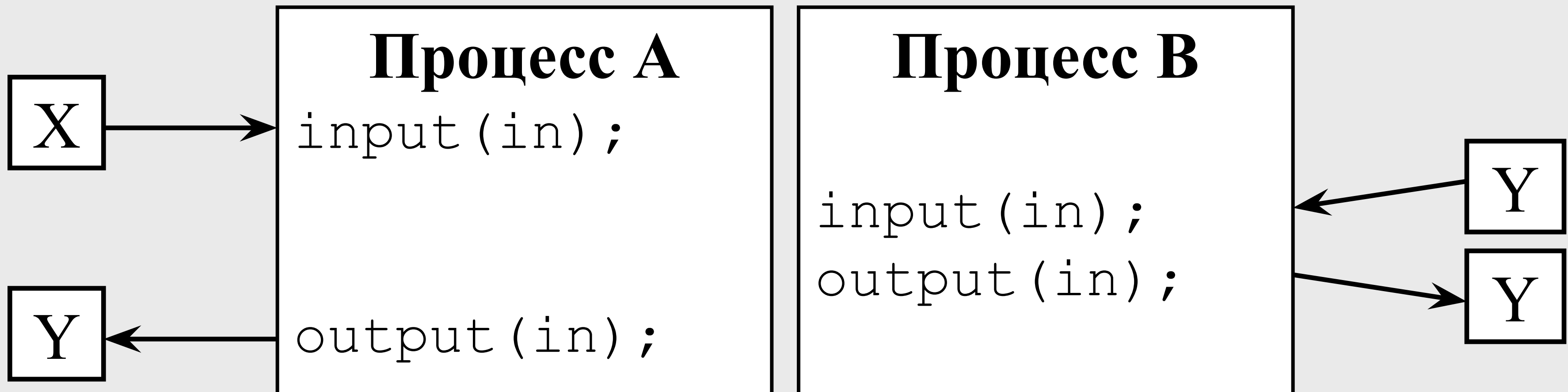
Критическая секция или **критический интервал** часть программы (фактически набор операций), в которой осуществляется работа с критическим ресурсом.

Требование мультипрограммирования

Результат выполнения процессов не должен зависеть от порядка переключения выполнения между процессами.

```
void echo ()
{
    char in;
    input ( in ) ;
    output ( in ) ;
}
```

Гонки (race conditions) между процессами.



Взаимное исключение

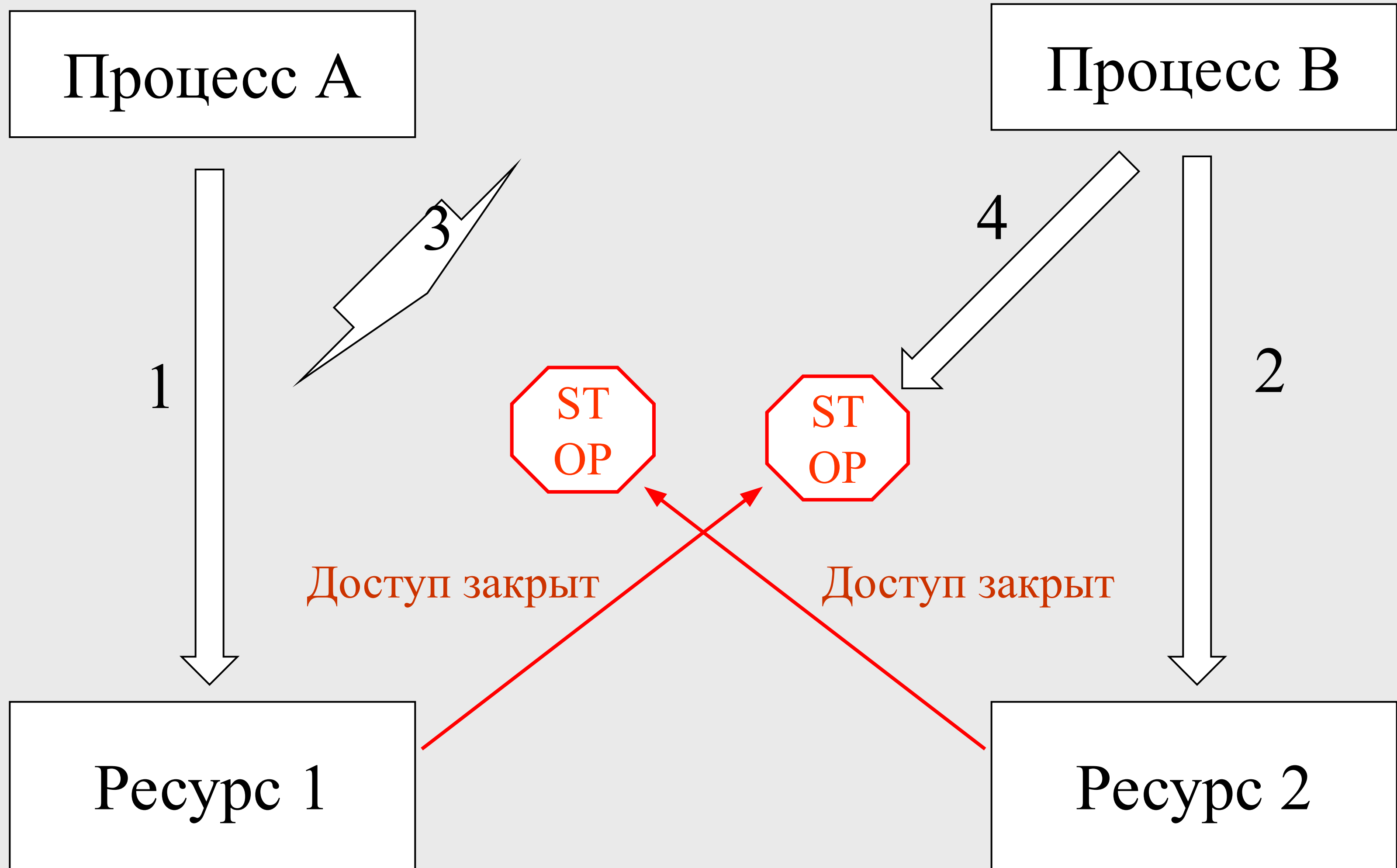
Взаимное исключение — способ работы с разделяемым ресурсом, при котором в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

- **Тупики (deadlocks)**
- **Блокирование (дискриминация)**

Тупик — ситуация, при которой из-за некорректной организации доступа и разделения ресурсов происходит взаимоблокировка.

Блокирование — доступ одного из процессов к разделяемому ресурсу не обеспечивается из-за активности других, более приоритетных процессов.

Тупики (Deadlocks)



Способы реализации взаимного исключения

Способы, которые позволяют работать с разделяемыми ресурсами. Существуют как аппаратные, так и алгоритмические модели.

- Запрещение прерываний и специальные инструкции
- Алгоритм Петерсона
- Активное ожидание
- **Семафоры Дейкстры**
- **Мониторы Хоара**
- **Обмен сообщениями**

Семафоры Дейкстры

Семафоры Дейкстры — формальная модель синхронизации, предложенная голландским учёным Эдсгером В. Дейкстрой

Операции, определённые над семафорами

- Down (S) или P (S) – **P**roberen (проверить)
- Up (S) или V (S) – **V**erhogen (увеличить)

Использование двоичного семафора для организации взаимного исключения

Двоичный семафор — семафор, максимальное значение которого равно 1.

процесс 1

```
int semaphore;
…
down ( semaphore ) ;
/*критическая секция
процесса 1*/
…
up ( semaphore ) ;
…
```

процесс 2

```
int semaphore;
…
down ( semaphore ) ;
/*критическая секция
процесса 2*/
…
up ( semaphore ) ;
…
```

Мониторы Хоара

Монитор Хоара — совокупность процедур и структур данных, объединенных в программный модуль специального типа.

- Структуры данных монитора доступны только для процедур, входящих в этот монитор
- Процесс «входит» в монитор по вызову одной из его процедур
- В любой момент времени внутри монитора может находиться не более одного процесса

Обмен сообщениями

Синхронизация и передача данных:

- для однопроцессорных систем и систем с общей памятью
- для распределенных систем (когда каждый процессор имеет доступ только к своей памяти)

Функции:

- send (destination, message)
- receive (source, message)

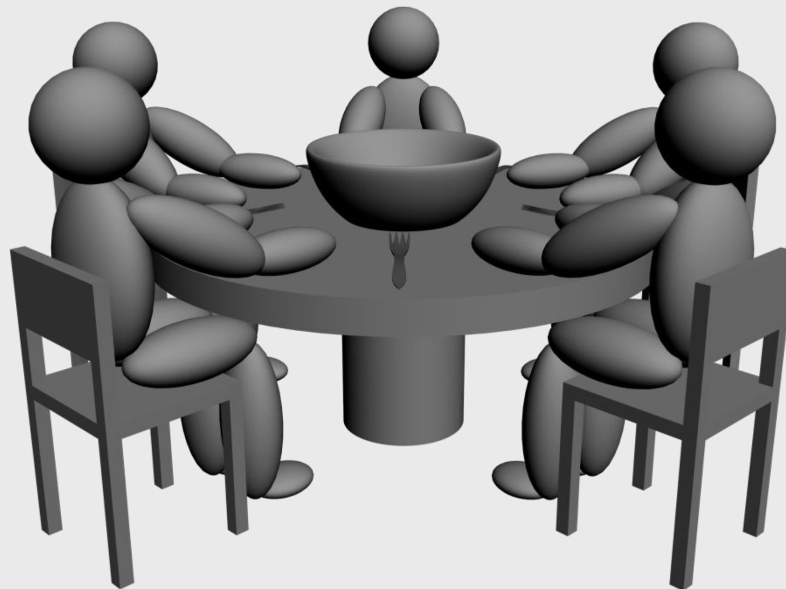
Обмен сообщениями

- Синхронизация
  - send/receive блокирующие
  - send/receive неблокирующими
- Адресация
  - Прямая (ID процесса)
  - Косвенная (почтовый ящик, или очередь сообщений)

Классические задачи синхронизации процессов

1. Обедающие философы
2. Задача о читателях и писателях
3. Задача о спящем парикмахере

«Обедающие философы» (доступ равноправных процессов к разделяемому ресурсу)



«Обедающие философы»

Простейшее решение

```
#define N 5

void philosopher ( int i )
{
   while (TRUE)
   {
    think () ;
    take_fork ( i ) ;
    take_fork ( ( i + 1 ) % N ) ;
    eat () ;
    put_fork ( i ) ;
    put_fork ( ( i + 1 )  % N ) ;
   }
   return;
}
```

«Обедающие философы»

Правильное решение с использованием семафоров

```
# define  N  5
# define  LEFT  ( i - 1 ) % N
# define  RIGHT  ( i + 1 ) % N
# define  THINKING  0
# define  HUNGRY  1
# define  EATING  2

typedef  int  semaphore ;
int state [ N ] ;
semaphore mutex = 1 ;
semaphore s [ N ] ;
```

«Обедающие философы»

Правильное решение с использованием семафоров

```
void philosopher ( int i )
{  while ( TRUE )
   {
   think () ;
    take_forks ( i ) ;
    eat ();
    put_forks ( i ) ;
   }
}
```

```
void take_forks ( int i )
{
   down ( & mutex ) ;
   state [ i ] =  HUNGRY ;
   test ( i ) ;
   up( & mutex ) ;
   down ( & s [ i ] ) ;
}
```

```
void put_forks ( int i )
{
   down ( & mutex ) ;
   state[i] = THINKING ;
   test ( LEFT ) ;
   test ( RIGHT ) ;
   up ( & mutex ) ;
}
```

```
void test ( int i )
{
   if ( ( state [ i ] == HUNGRY )
    && ( state [ LEFT ] != EATING )
   && ( state [ RIGHT ] != EATING ))
   {
    state [ i ] = EATING ;
    up ( & s [ i ] ) ;
   }
}
```

«Читатели и писатели» (задача резервирования ресурса)

```
typedef int semaphore ;
int rc = 0 ;
semaphore mutex = 1 ;
semaphore db = 1 ;

void reader ( void )
{
 while ( TRUE )
 {
   down ( & mutex ) ;
   rc++ ;
   if ( rc == 1 ) down ( & db )
 ;
   up ( & mutex ) ;
   read_data_base () ;
   down ( & mutex ) ;
   rc-- ;
   if ( rc ==0 ) up ( & db ) ;
   up ( & mutex ) ;
   use_data_read () ;
 }
}

void writer ( void )
{
  while ( TRUE )
  {
      think_up_data () ;
      down ( & db ) ;
      write_data_base   ()
  ;
      up ( & db ) ;
  }
}
```

«Спящий парикмахер» (клиент-сервер с ограничением на длину очереди)

```c
#define CHAIRS 5
typedef int semaphore ;
semaphore customers = 0 ;

void barber ( void )
{
   while ( TRUE )
   {
    down ( & customers ) ;
    down ( & mutex ) ;
    waiting = wating - 1 ;
    up ( & barbers ) ;
    up ( & mutex ) ;
    cut_hair () ;
   }
}
```

```c
semaphore barbers = 0 ;
semaphore mutex = 1 ;
int waiting = 0 ;

void customer ( void )
{
   down ( & mutex ) ;
   if ( waiting < CHAIRS )
   {
        waiting = waiting + 1
   ;
        up ( & customers ) ;
        up ( & mutex ) ;
        down ( barbers ) ;
        get_haircut () ;
   }
   else
   {
        up ( & mutex ) ;
   }
}
```