Взаимодействие процессов: синхронизация, тупики

Параллельные процессы

Параллельные процессы – процессы, выполнение 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, когда каждый из процессов некоторое время владеет ресурсом

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

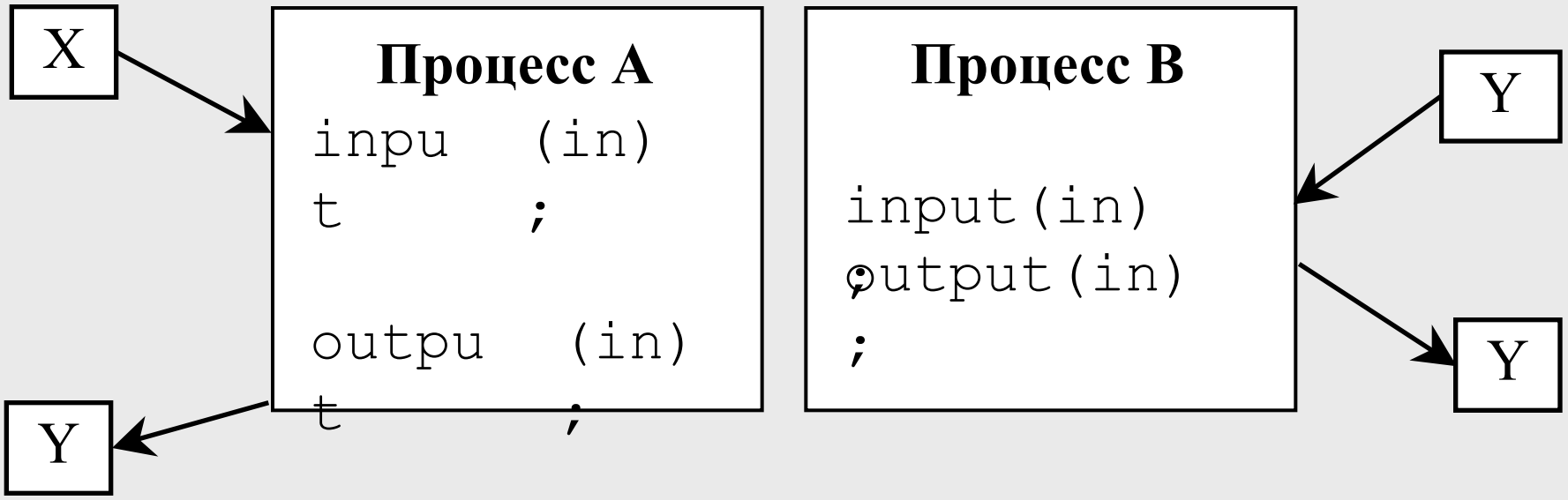
Важнейшие задачи

- Распределение ресурсов между процессами
- Организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов

Требование мультипрограммирования

Результат выполнения процессов не должен зависеть от порядка переключения выполнения между процессами, т.е. от соотношения скорости выполнения данного процесса со скоростями выполнения других процессов

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



Взаимное исключение

Гонки (race conditions) между процессами

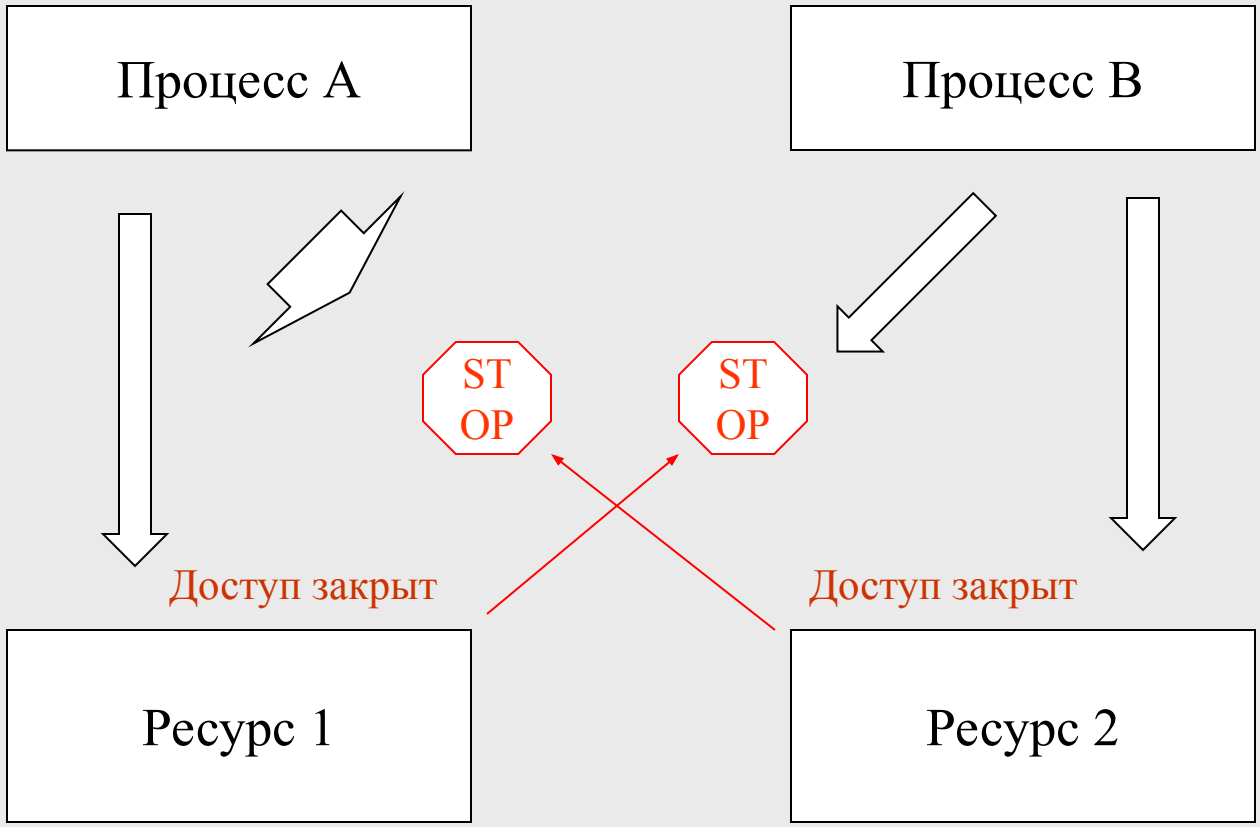
Взаимное исключение – такой способ работы с разделяемым ресурсом, при котором в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

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

Проблемы организации взаимного исключения

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

Тупики (deadlocks)



Способы реализации взаимного исключения

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

Семафоры Дейкстры

S – переменная целого типа

Операции над S

- Down(S) (или P(S))
- Up(S) (или V(S))

Использование двоичного семафора для организации взаимного исключения

Д***воичный семафор -*** семафор, начальное (и 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

Монитор - языковая конструкция, т.е. некоторое средство, предоставляемое языком программирования и поддерживаемое компилятором. Монитор – это 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

Средство, решающее проблему синхронизации

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

Обмен сообщениями

send (destination, message)

receive (source, message)

- Синхронизация
  - Операции посылки/приема сообщения могут быть блокирующими и неблокирующими.
- Адресация
  - -Прямая (ID процесса)
  - -Косвенная (почтовый ящик, или очередь сообщений)
- Длина сообщения

Классические задачи синхронизации процессов

«Обедающие философы»

```
#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);
    }
}
```

```
# 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];
```

```
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]);
}
```

Задача «читателей и писателей»

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

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

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

Задача о «спящем парикмахере»

```
#define CHAIRS 5
typedef int semaphore;
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;
```

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

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