5. Концепция процесса

Цель:

- формализовать процессы, происходящие при выполнении программ реального времени, повысить уровень представления;
- на основе предложенных формализмов создать высокоуровневые средства надежного программирования СРВ

Концепция процесса (2)

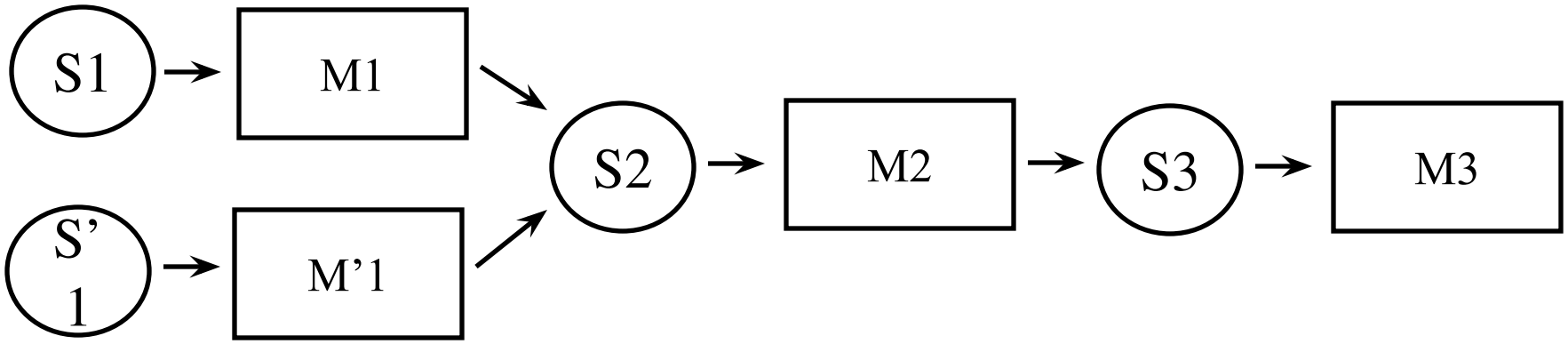
- Процесс – это действия «машины» при выполнении программы
- Несколько процессов могут выполняться «машиной» параллельно во времени
- На длительность процесса и скорость его развития не накладывается никаких ограничений («длинные процессы», «короткие процессы», быстротекущие, медленные .....)

Концепция процесса – систем взглядов, положенная в основу подхода к построению средств реализации СРВ

Концепция процесса (3)

«Машина» (виртуальная машина) – модель, реализующая пользовательское представление некоторой вычислительной среды

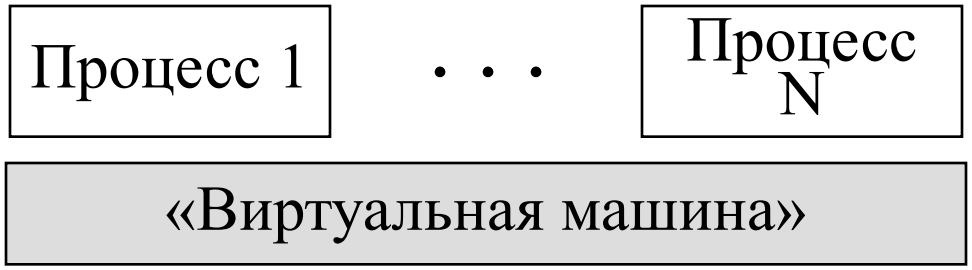
(*Вейдерман, 1971 год – Иерархия виртуальных машин:*



Si– программа на языке Si, (S1 – ЯВУ, S2 – Assembler, S3 – машинный код)

Mi – виртуальная машина, реализующая эту программу - Paskal-машина, Assembler-машина, «железо»)

Концепция процесса (4)



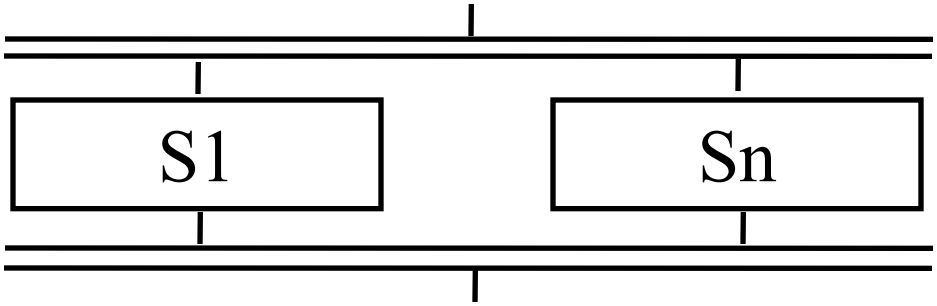
«Виртуальная машина»

- обеспечивает параллельное выполнение процессов

Процесс

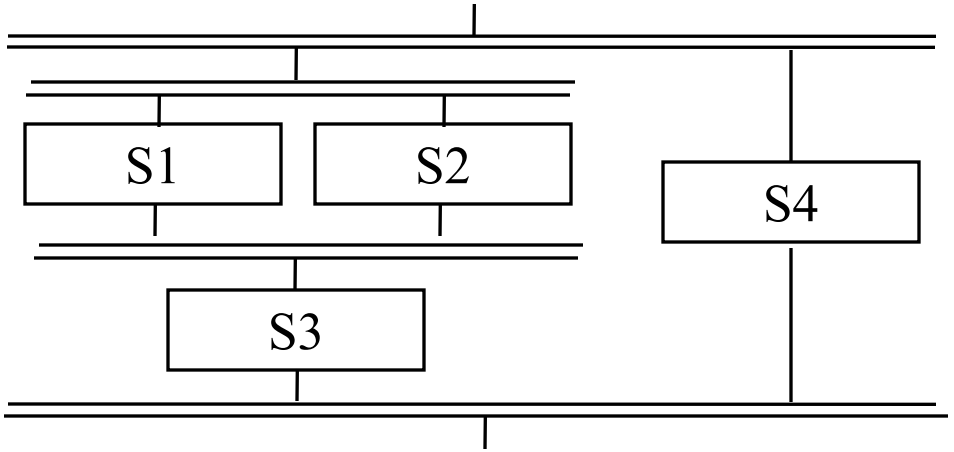
- описывется программой на некотором языке (например – задача (task) для «ADA-машины» , нитка (thread) для «Java – машины», задача многозадачной ОС)

Представление процессов




```
Parbegin  S1; . . . Sn  Parend;
```

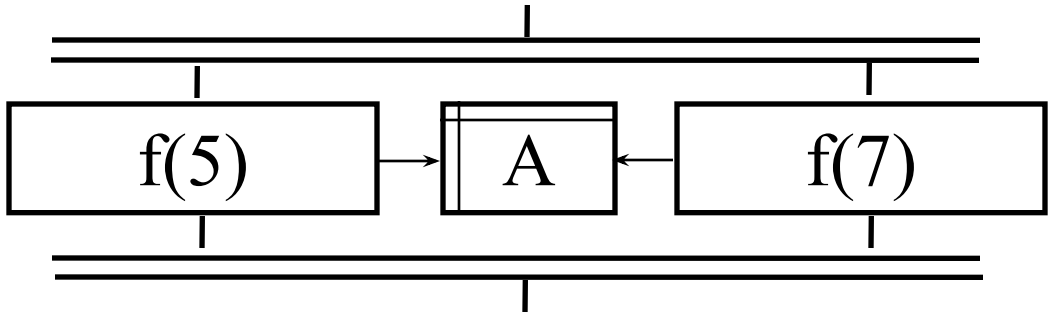
```
Parbegin
   begin
      Parbegin
         S1;S2;
      Parend;
      S3;
   end;
   S4;
Parend;
```



Синхронизация процессов

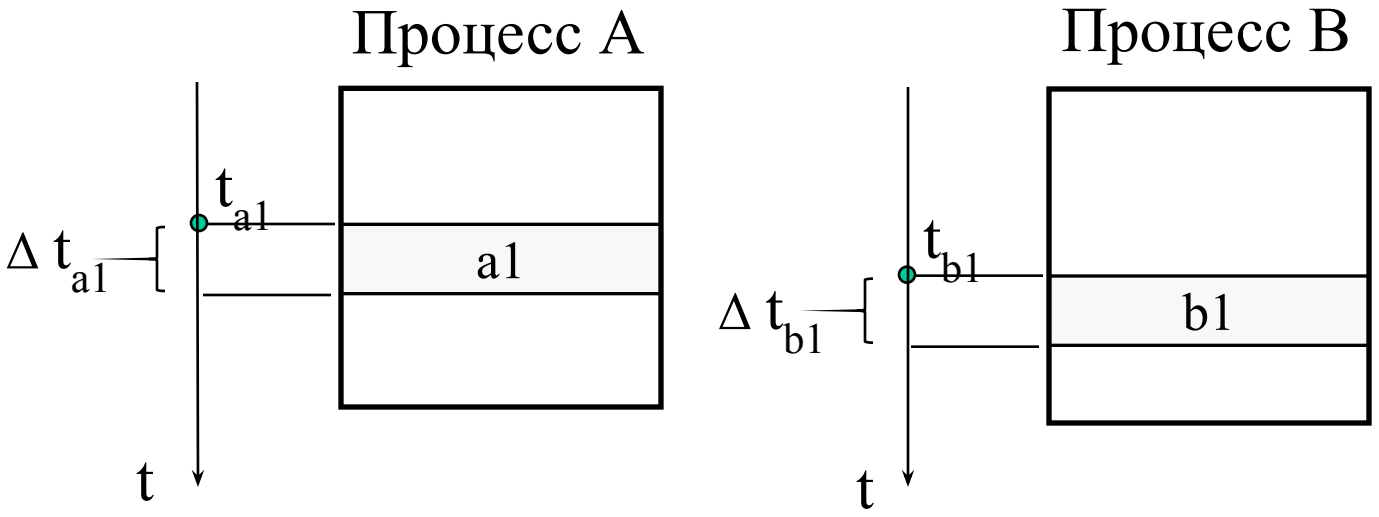
```
procedure f(I: integer);
    var C: integer;
    while(true);
        C:= A;
        doSomething(C); { C:= C+I }
        A:= C;
         doSomethingElse();
    endwhile;
endproc;
```

```
Parbegin
    f(5);
    f(7);
Parend;
```



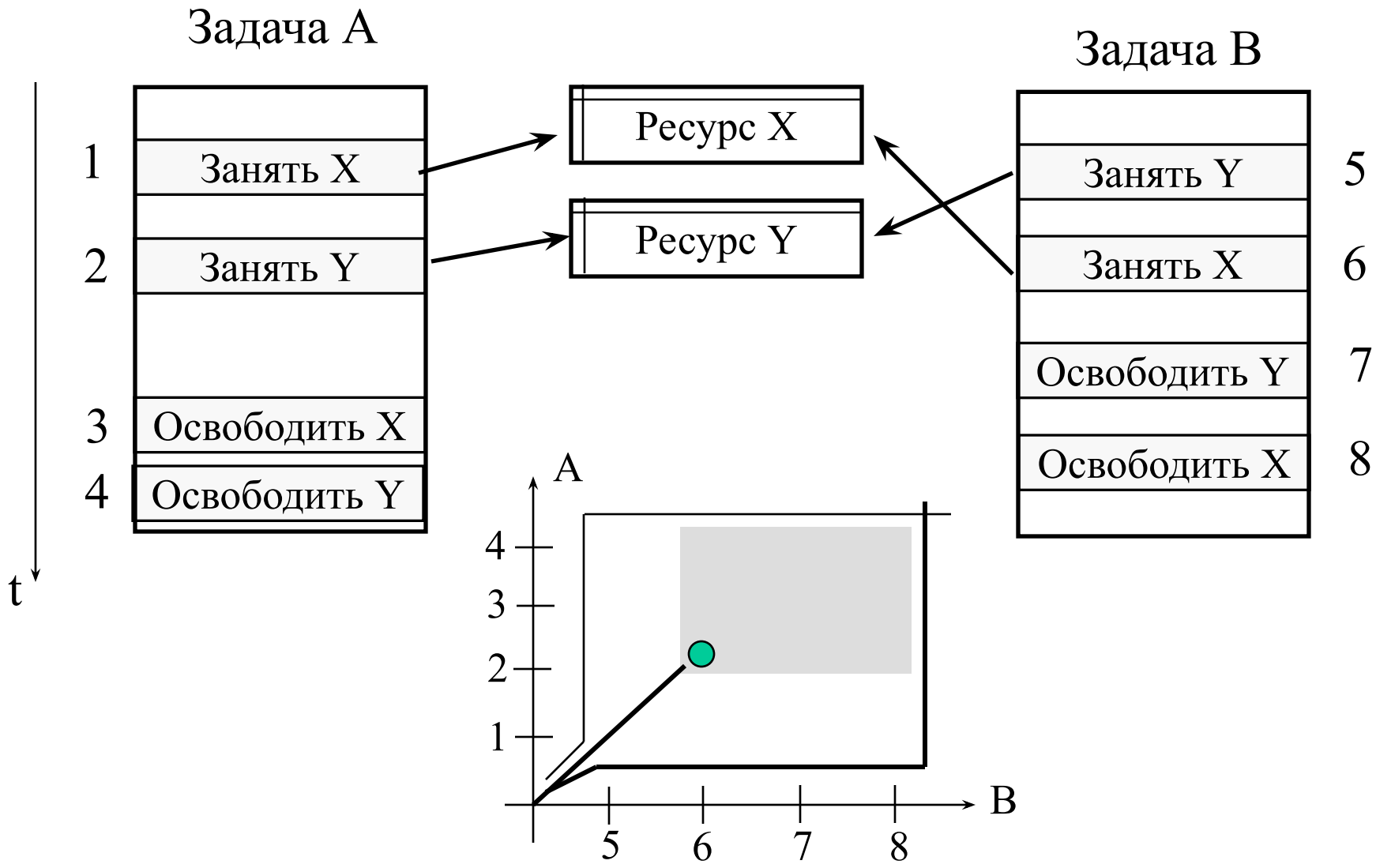
Синхронизация процессов (2)

Синхронизация – выполнение заданных временных соотношений между процессами

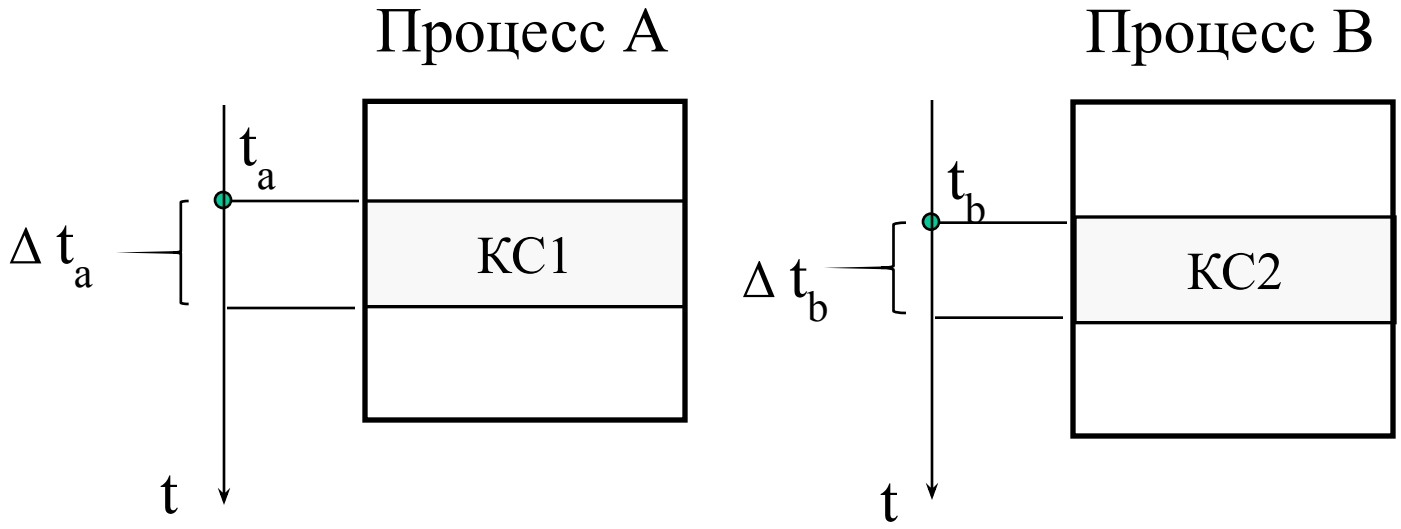


1. $t_{b1} > t_{a1}$ соотношение предшествования
2. $(t_{a1} + \Delta t_{a1}) \exists (t_{b1} + \Delta t_{b1}) = O$ взаимное исключение
3. $t_{b1} = t_{a1}$ рандеву

Взаимная блокировка (тупик, deadlock)



Задача взаимного исключения



а) недопустимо одновременное выполнение критических секций КС1 и КС2;

б) ни одна из задач не должна ждать входа в свою критическую секцию бесконечно долго (блокировка недопустима)

Попытка алгоритмического решения - Алгоритм Деккера

Алгоритм Деккера - вспомогательная схема 1

```
var Очередь: 1,2;
Очередь:= 1;

  Parbegin
    loop
      while(Очередь = 2)
      endwhile;
      КС_1;
      Очередь:= 2;
      Остальное_1;
    endloop;
```

```
    loop
      while(Очередь = 1)
      endwhile;
      КС_2;
      Очередь:= 1;
      Остальное_2;
    endloop;
Parend.
```

Алгоритм Деккера, вспомогательная схема 1

Вспомогательная схема 1: задача не решена

а) при остановке одного из процессов второй процесс блокируется; причина – разделяемая переменная **`Очередь`**.

Однако

б) одновременное выполнение критических секций невозможно, т.к. вход к КСi строго контролируется переменной **`Очередь`**

б') следует обратить внимание на то, что выполнение процессов строго чередуется.

Алгоритм Деккера - вспомогательная схема 2

```
var c1,c2: занят,свободен;
c1:= свободен; c2:= свободен

Parbegin
  loop
    c1:= занят; {*}
    while(c2=занят)
    endwhile;
    КС_1;
    c1:= свободен;
    Остальное_1;
  endloop;
```

```
  loop
    c2:= занят; {*}
    while(c1=занят)
    endwhile;
    КС_2;
    c2:= свободен;
    Остальное_2;
  endloop;
Parend.
```

Алгоритм Деккера, вспомогательная схема 2

Вспомогательная схема 2: задача не решена

а) при одновременном выполнении действий {*} происходит блокировка

Однако

б) одновременное выполнение критических секций невозможно, т.к. при входе одного процесса в свою КС второй будет остановлен в результате того, что переменная **c** примет запрещающее значение - **схема надежна с точки зрения одновременного входа в КС**

Алгоритм Деккера, реализация

```
Parbegin
  loop
    c1:= занят; {*}
    while(c2=занят)
      if(Очередь=2)
        c1:= свободен
      endif;
      while(c2=занят) endwhile;
    c1:= занят;
  endwhile;
  КС_1;
  c1:= свободен; Очередь:=2
  Остальное_1;
endloop;
```

{**}

```
loop
    c2:= занят; {*}
    while(c1=занят)
      if(Очередь=1)
        c2:= свободен
      endif;
      while(c1=занят) endwhile;
    c2:= занят;
  endwhile;
  КС_2;
  c2:= свободен; Очередь:=1
  Остальное_2;
endloop;
Parend.
```

{**}

Алгоритм Деккера, реализация (2)

а) В теле цикла {**} параметр цикла не изменяется, т.е. цикл {**} можно представить как «занятое ожидание» -

```
while(c=занят) A endwhile.
```

В этих условиях рассматриваемый алгоритм аналогичен Схеме 2, которая **обеспечивает взаимное исключение**

б) При одновременном выполнении операторов {*} «работает» переменная **`Очередь`**, позволяющая разблокировать один из процессов. **Недостаток Схемы 2 ликвидирован**

в) Остановка одного из процессов вне критической секции не блокирует другой процесс. **Ни один из процессов не будет ждать бесконечно долго входа в свою КС**

Некоторые выводы

- Задача решена, показана принципиальная возможность обеспечения взаимного исключения алгоритмическим путем, однако, практического значения представленное решение не имеет
- Необходимы МЕХАНИЗМЫ, позволяющие надежно (*обеспечивая невозможность блокировки*) решать проблемы синхронизации (надо усложнять «виртуальную машину»)
- Для СРВ такие механизмы не должны влиять на предсказуемость поведения системы (например, Java-машина не удовлетворяет этому условию)