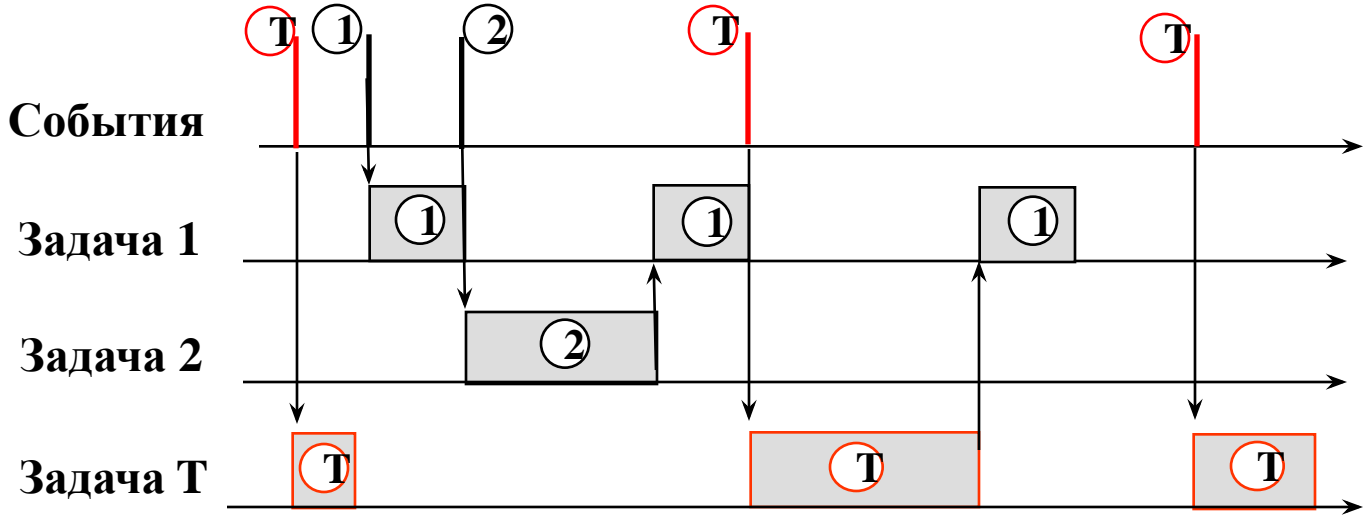
История вопроса, «Концепция процесса»

Дейкстра «Взаимодействующие последовательные процессы»
«Языки программирования» под ред. Ф. Женюи, М., "Мир", 1972

Дейкстра Эдсгер 1930-2002

Сложность монопольных приложений - наряду с циклическими задачами надо обеспечивать своевременную реакцию на асинхронные события.



1. Прерывание от таймера – запуск периодической задачи T
2. Событие «1» - прерывается T, активизируется задача 1
3. Событие «2» - прерывается 1, активизируется задача 2
4. Задача 2 заканчивается, возобновляется задача 1
5. И т. д. . . . . .

Асинхронное выполнение задач

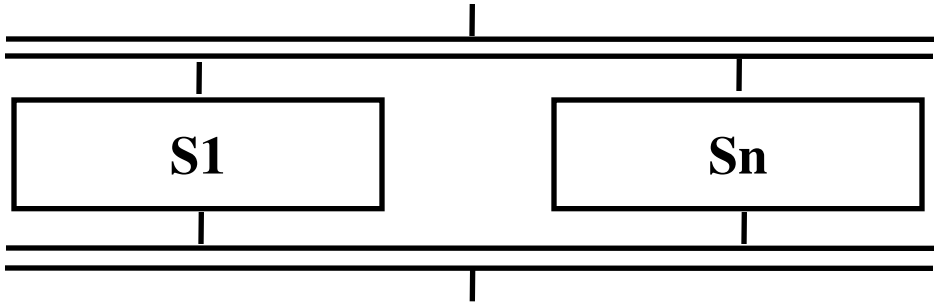
«Виртуальная машина» – вычислительная среда – процессоры + ОС

Число процессоров M > N; M < N; M=N

Условия реализации :

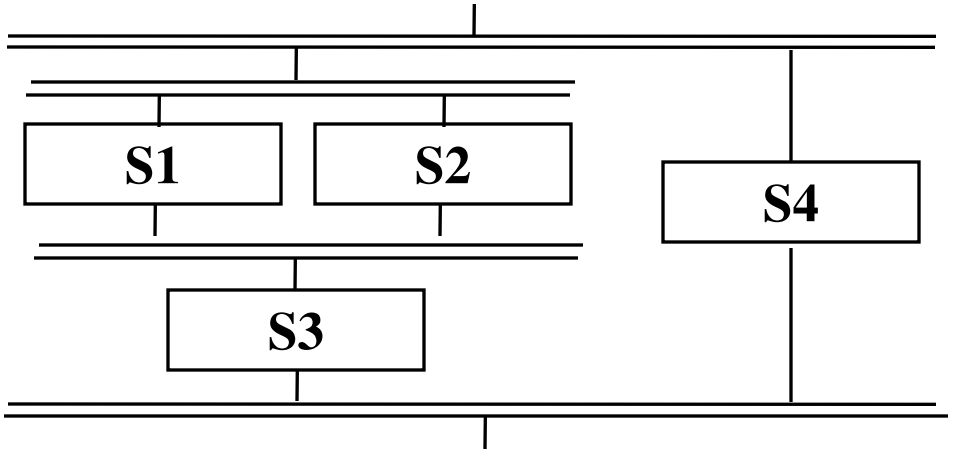
- Задача (процесс) – это действия «виртуальной машины» при выполнении программы
- Несколько задач (процессов) могут выполняться «машиной» параллельно во времени; реализация нас не беспокоит
- На длительность процесса и скорость его развития не накладывается никаких ограничений («длинные процессы», «короткие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

Описание



Parbegin S1; . . . Sn Parend;

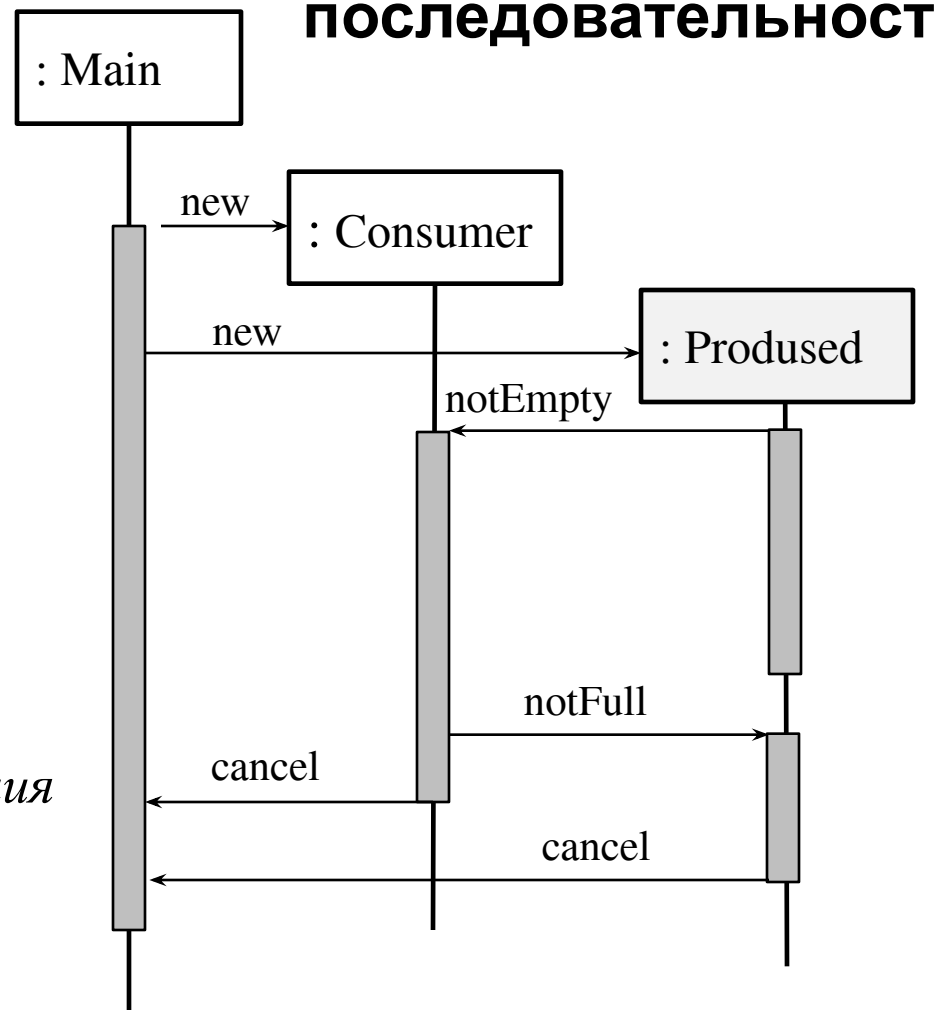
```
Parbegin
    begin
        Parbegin
            S1;S2;
        Parend;
        S3;
    end;
    S4;
Parend;
```



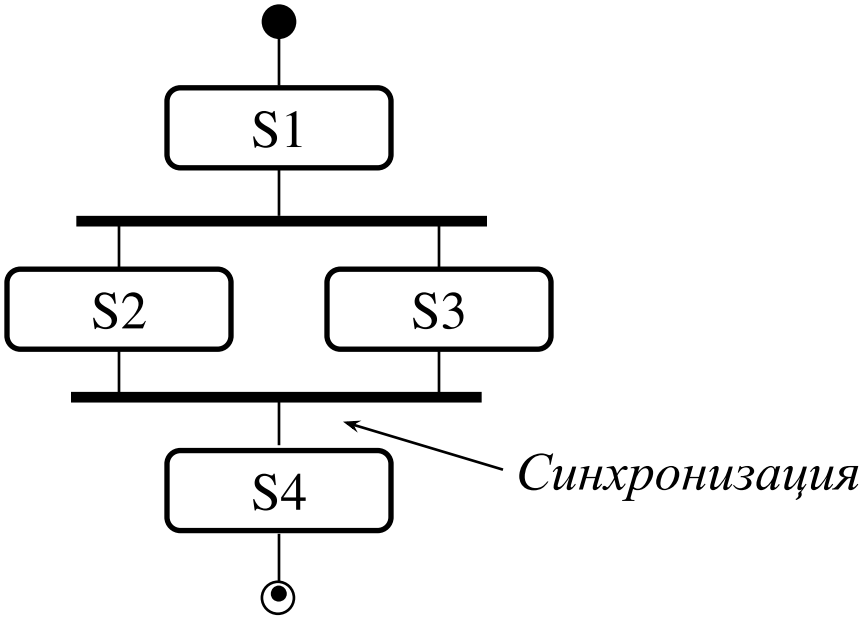
задача (task) для «ADA-машины», нитка (thread) для «Java – машины», процесс и поток многозадачной ОС

Развитие описания, пример - UML диаграммы

Диаграммы последовательности

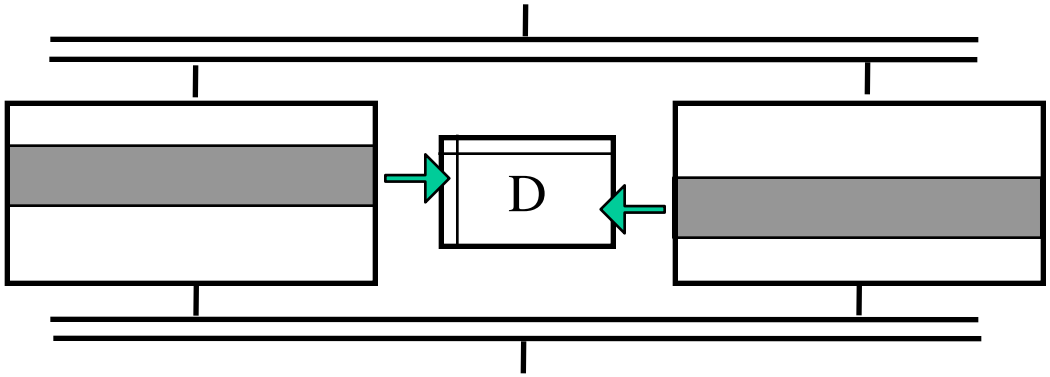


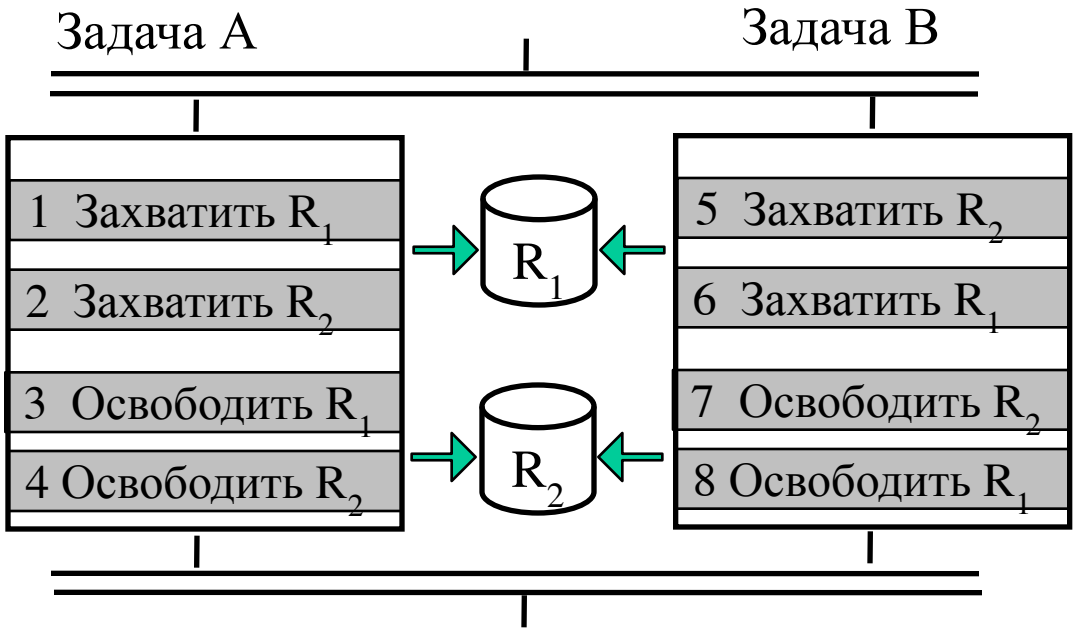
Диаграммы активности

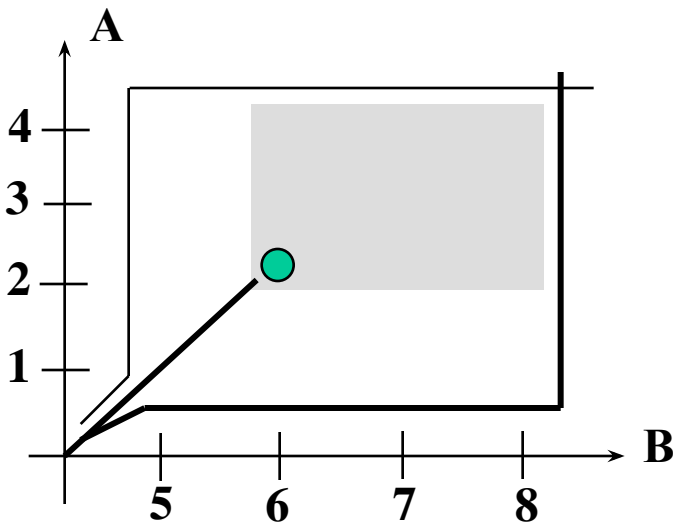


Проблемы

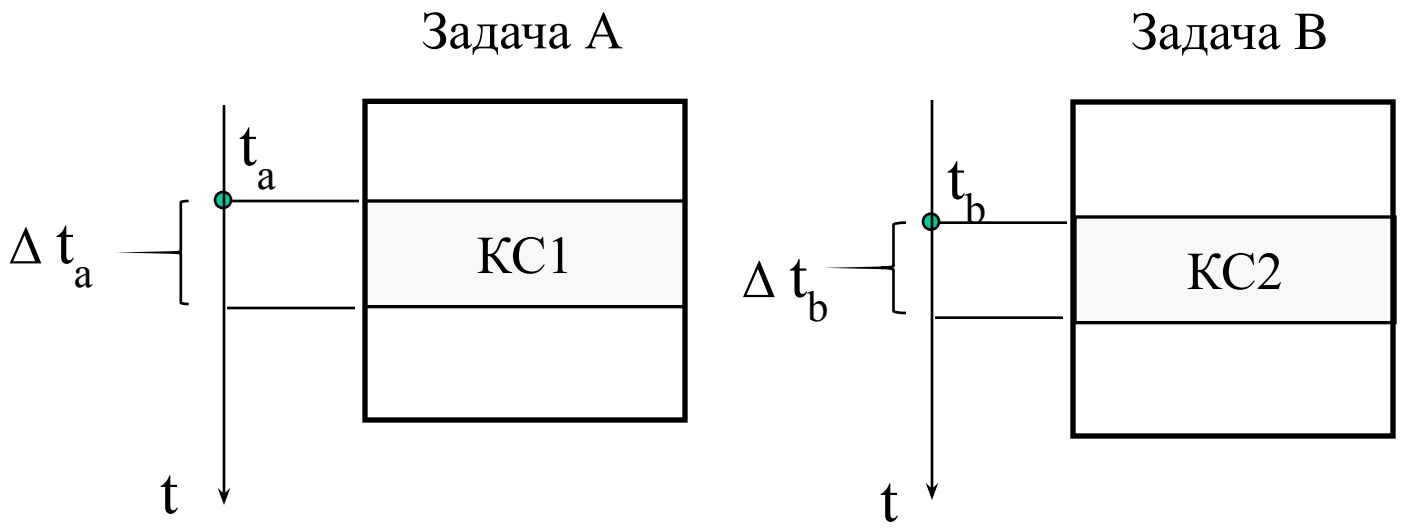
- **Критическая секция**
- **Взаимная блокировка (тупики, deadlock)**







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

```
integer Очередь = 1;

  Parbegin
    while(true){
      while(Очередь == 2){}
      КС_1;
      Очередь = 2;
      Остальное_1;
    }
```

```
    while(true){
      while(Очередь == 1){}
      КС_2;
      Очередь = 1;
      Остальное_2;
    }
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`**

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

Алгоритм Деккера - вспомогательная схема 2

```
integer c1 = 1, //занят
        c2 = 0; //свободен;

Parbegin
  while(true){
    c1= 1; {*}
    while(c2==1){}
    КС_1;
    c1 = 0;
    Остальное_1;
  }
```

```
  while(true){
    c2:= 1; {*}
    while(c1==1){}
    КС_2;
    c2 = 0;
    Остальное_2;
  };
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

```
c1 = 1; c2 = 0; Очередь = 1
Parbegin
  while(true){
    c1:= 1; {*}
    while(c2==1){
      if(Очередь == 2){
{**}    c1 = 0;
      }
      while(c2 == 1){}
      c1 = 1;
    };
    КС_1;
    c1 = 0; Очередь = 2
    Остальное_1;
  };
```

```
  while(true){
    c2:= 1; {*}
    while(c1==1){
      if(Очередь == 1){
{**}    c2 = 0
      }
      while(c1 == 1){};
      c2 = 1;
    }
    КС_2;
    c2 = 0; Очередь = 1
    Остальное_2;
  };
Parend.
```

Алгоритм Деккера, реализация

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

```
while(c=занят) A end.
```

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**

Механизм семафоров
Двоичный семафор

```
class ДвСемафор {

    private integer значение = 1;

    void P{

        if(значение = = 0) then
            ждать(значение>0);
        значение = 0;
    }
    void V{
        значение = 1;
    }
}
```

Конструкция динамическая:

```
ДвСемафор S =
        new ДвСемафор();
```

Двоичный семафор

- При создании объекта **семафор S** с ним связывается очередь заблокированных процессов; очередь обслуживается в соответствии с определенной дисциплиной, не допускающей бесконечного ожидания процесса (например, FIFO)
- При **значение=0** операция **ждать(значение>0)** блокирует процесс, вызвавший **P**–операцию и и ставит его в очередь до момента выполнения условия **значение>0;**
- При выполнении **V**–операции активизируется первый процесс из очереди; продолжение этого процесса происходит с действия **S=0** Приостановленной **P**–операции
- **P** и **V** операции над семафором **S** являются неделимыми (могут выполняться только одним процессом)

Обеспечение взаимного исключения

```
ДвСемафор S =
      new ДвСемафор();

Ri:   loop
          S.P();
          КС_I;
          S.V();
          Остальное_I;
       endloop;

Parbegin
   R1;. . .Rn;
Parend.
```

- **P** и **V** операции неделимы; т.е. выполнять **S.P()** может только один процесс ;
- Если один из процессов выполнил S.**P()** и вошел в свою **КС**, то **S.значение = 0** и все остальные процессы, выполняющие **S.P()**, будут блокироваться. Т.е взаимное исключение обеспечено
- При выполнении **S.V()** соблюдается очередность, т.е. никто не будет ждать бесконечно долго

Общий (счетный) семафор

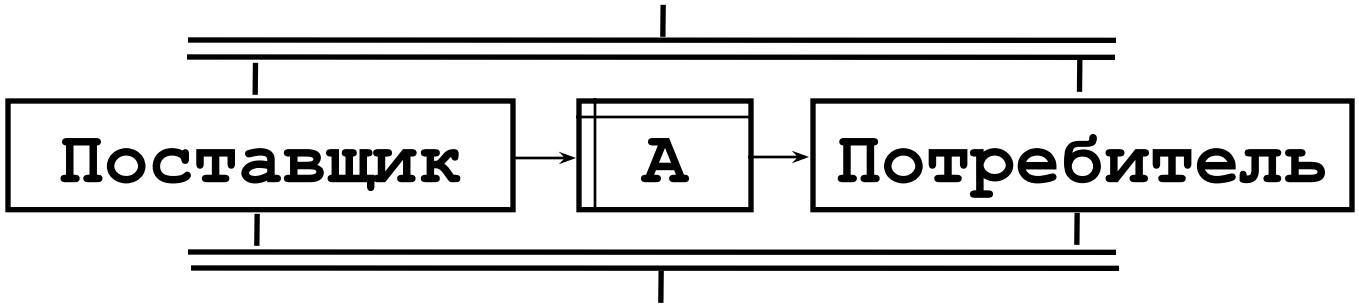
```
class ОбСемафор {

    private integer значение;
    private integer maxЗначение;
    ОбСемафор(integer n, m) {
        значение = n; maxЗначение = m;
    }
    void P(){
        if(значение = = 0) then
            ждать(значение>0);
        значение--;
    }
    void V(){
        if(значение<maxЗначение) значение++;
    }
}
```

Конструкция динамическая:

```
ОбСемафор S =
        new ОбСемафор(N,M);
```

Пример: «Поставщик-Потребитель»



Поставщик : производит единицы информации и записывает их в буфер **A**, размер которого **N**

Потребитель : читает и обрабатывает информацию из буфера **A**

Требуется реализовать процессы Поставщик и Потребитель таким образом, чтобы выполнялись условия:

а) **Поставщик** не может записывать в полный буфер;

б) **Потребитель** не может читать из пустого буфера;

в) одновременные запись и чтение недопустимы

«Поставщик-Потребитель» - реализация

```
ДвСемафор Доступ = new ДвСемафор();
ОбСемафор Полон  = new ОбСемафор(N,N);
ОбСемафор Пуст   = new ОбСемафор(0,N);
```

Поставщик:	Потребитель:
loop	loop
Производство;	Пуст.P();
(*) Полон.P();	Доступ.P();
(**) Доступ.P();	Чтение_из_буфера;
Запись_в_буфер;	Доступ.V();
Доступ.V();	Полон.V();
Пуст.V();	Обработка;
endloop.	endloop.

```
Parbegin  Поставщик; Потребитель  Parend;
```

Недостатки механизма семафоров

- Низкий уровень, слабая защищенность от ошибок при программировании
    - Возможность использования P() и V() над одним семафором из разных задач (семафор **S** может быть установлен в 0 в одной задаче, а в 1 – в другой, см. задачу «Поставщик-Потребитель»)
    - Возможность вызова V-операции над семафором без предварительного вызова P-операции
- **Не позволяет реализовать протокол наследования приоритетов, т. к. не определен процесс-«владелец» семафора**

Пример сервисов, поддерживающих семафоры в RTEK

KC_CloseSema	– end the use of dynamic semaphore
KC_DefSemaCount	– define a semaphore count
KC_DefSemaName	– define the name of a previously opened dynamic semaphore
KC_GetSemaCount	– get the current semaphore count
KC_GetSemaName	– get the name of a semaphore
KC_GetSemaProp	- get the properties of a semaphore
KC_GetSemaWaiters	– get the number and list of tasks waiting on semaphore
KC_InitSemaClassProp	– initialize the semaphore object class properties

Пример сервисов, поддерживающих семафоры в RTEK

KC_LookupSema	– look up a semaphore's name to get its handle
KC_OpenSema	– allocate and name a dynamic semaphore
KC_SignalSema	– signal a semaphore
KC_SignalSemaM	– signal multiply semaphores
KC_TestSema	– test a semaphore
KC_TestSemaT	– test a semaphore and wait for a specified time if the semaphore is not DONE
KC_TestSemaW	- test a semaphore and wait if the semaphore is not DONE

Семафоры в QNX

Неименованные семафоры

```
#include <semaphore.h>
(sem_t  mySem;)

int sem_init( sem_t * sem,  int  pshared,  unsigned value );
```

***sem** – *указатель на объект, который мы хотим инициировать*

pshared – *не «0», если мы хотим, чтобы sem разделялся процессами через разделяемую память*

value – *значение семафора (> 0, если «0», то семафор блокирован)*

int sem_destroy(sem_t * *sem*);

int sem_wait(sem_t * *sem*);

int sem_post(sem_t * *sem*) – *инкремент значения семафора;*

int sem_trywait(sem_t * *sem*) - *неблокирующий декремент значения семафора*

Именованные семафоры – *межпроцессная и сетевая поддержка; примитивы другие.*