Примитивы Синхронизации

(Анатомия параллелизма)

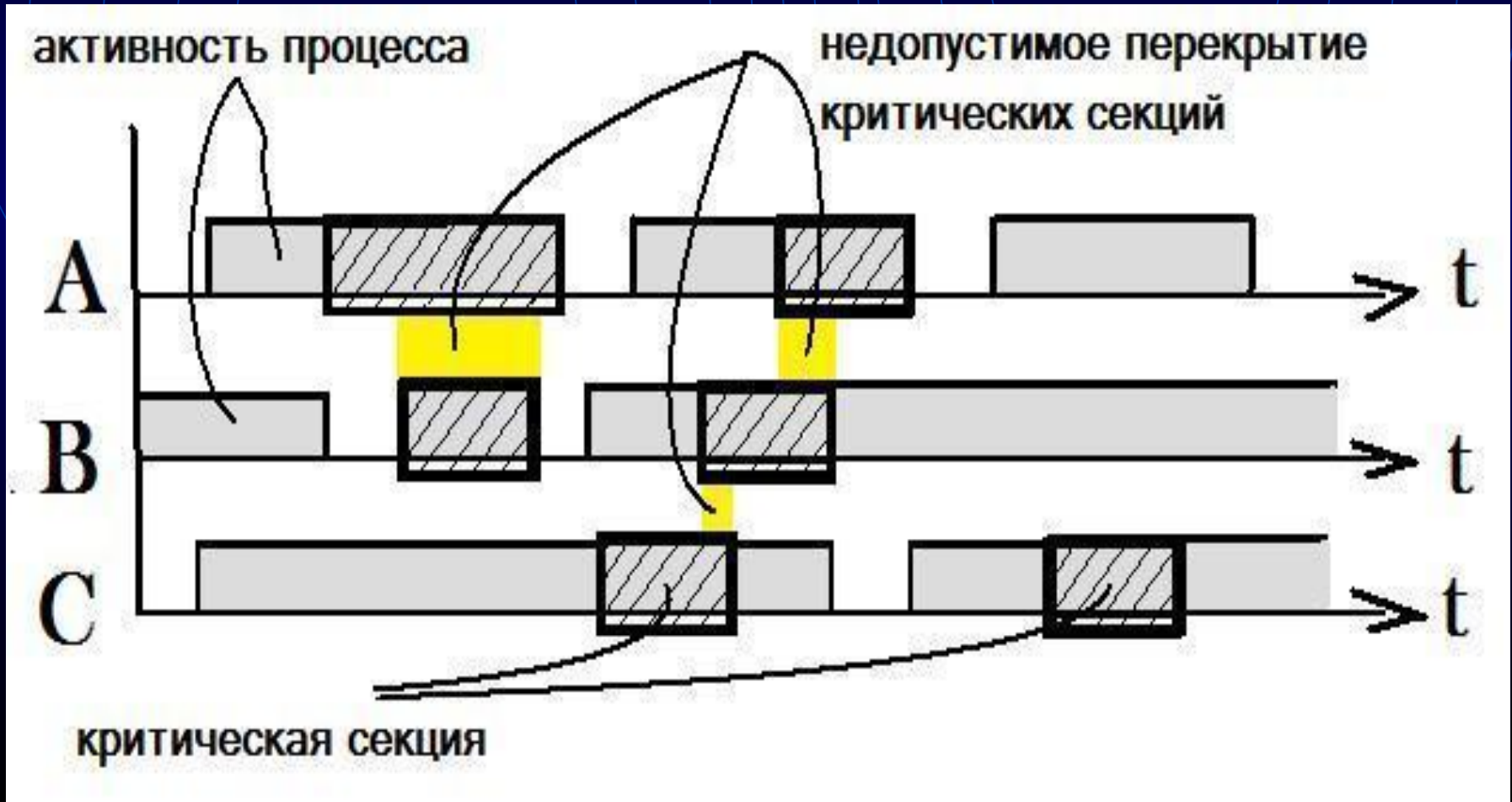
Типы синхронизирующих примитивов (системные средства ОС и надстройки)

критическая секция
семафор
сигнал
барьер
мьютекс
условные переменные
атомарные операции
рандеву
монитор

• • •

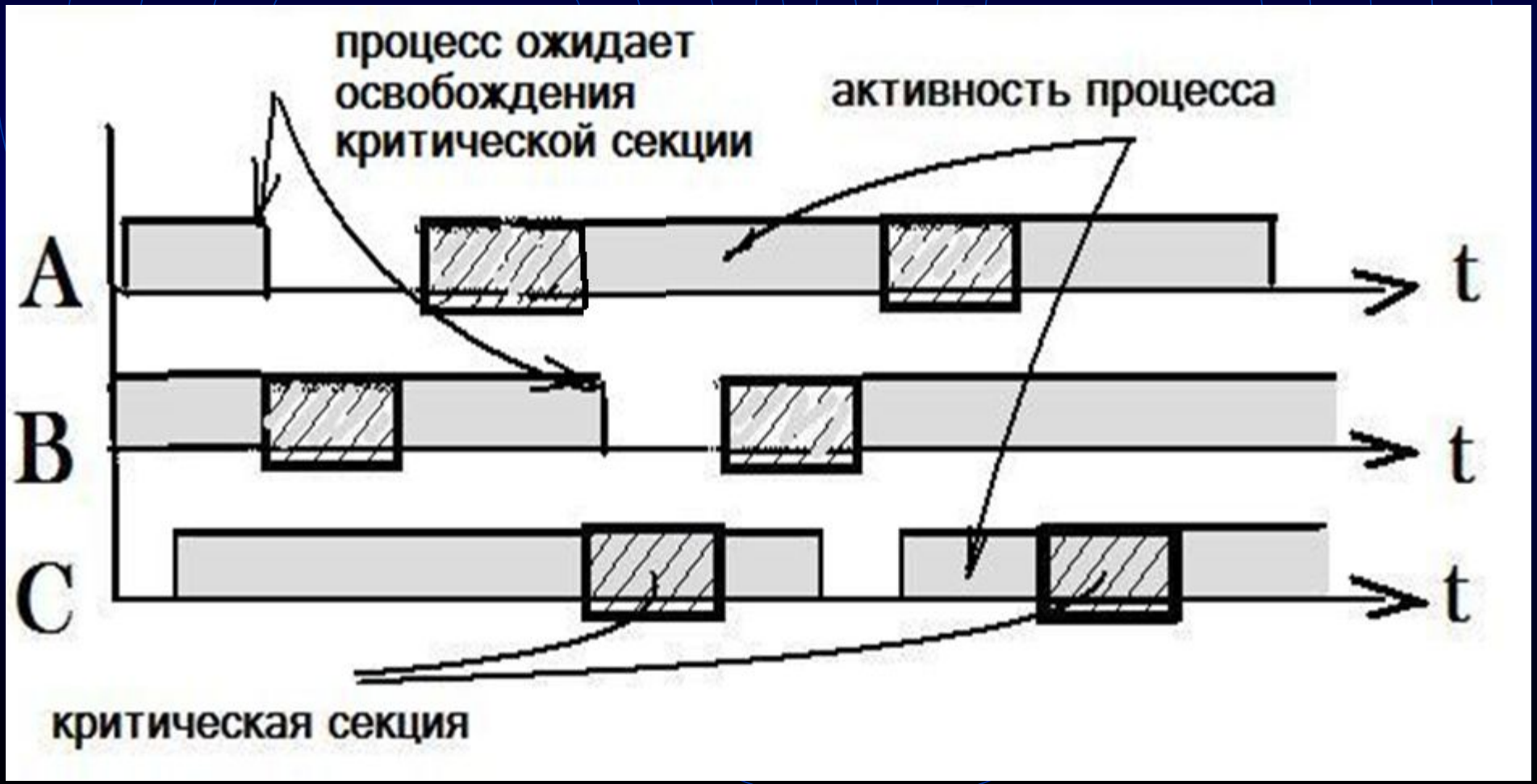
Критическая секция

В **любой** момент времени в критическом интервале может находиться не более одного процесса



Критическая секция

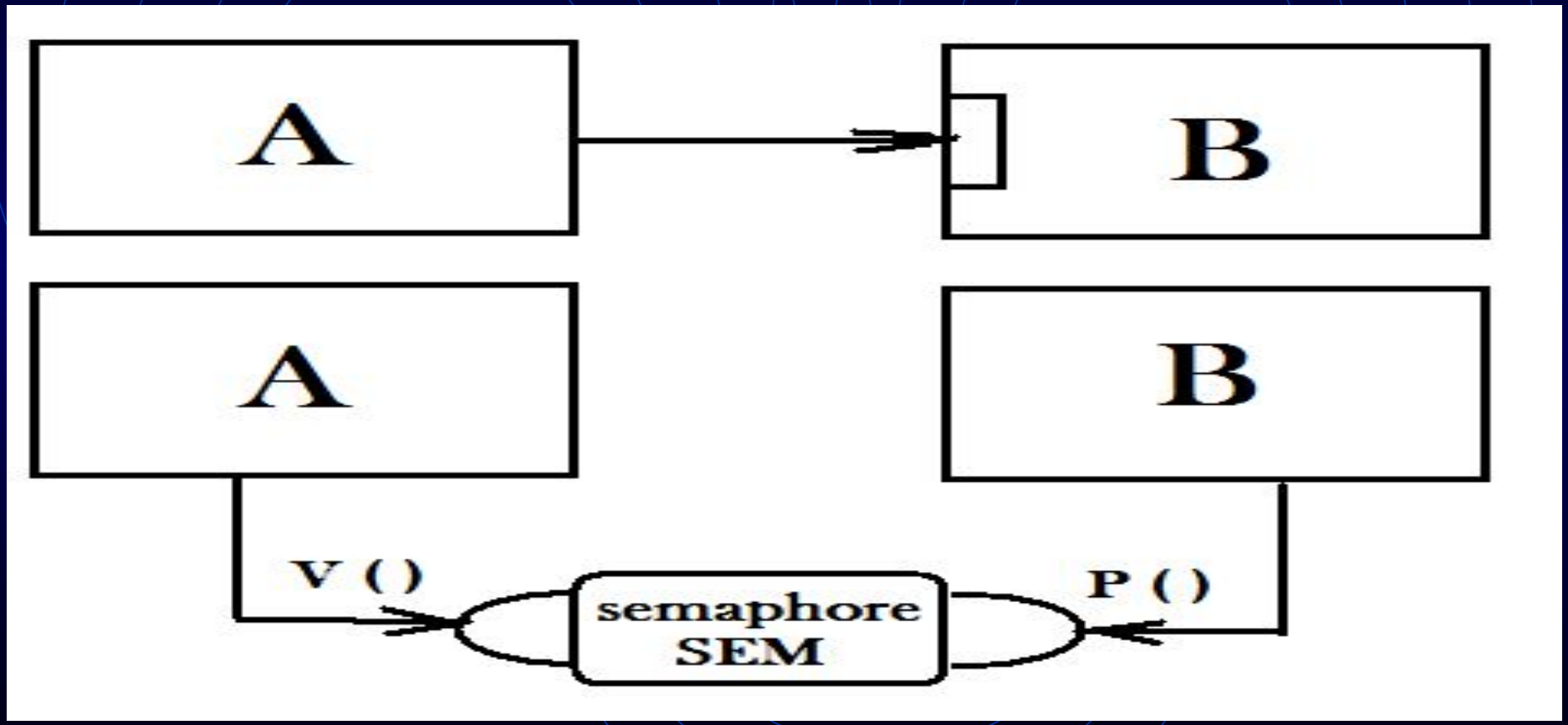
В **любой** момент времени в критическом интервале может находиться не более одного процесса



Семафор двоичный

```
classTSemaphore {
private:
        int count;  // локальный счетчик
public:
      void P(){while( ! count );    count =0;}
      void V(){count =1;}
      TSemaphore(int start)  {
            if(start) count =1;    else count =0;
      }
     ~TSemaphore();
};
```

Семафор



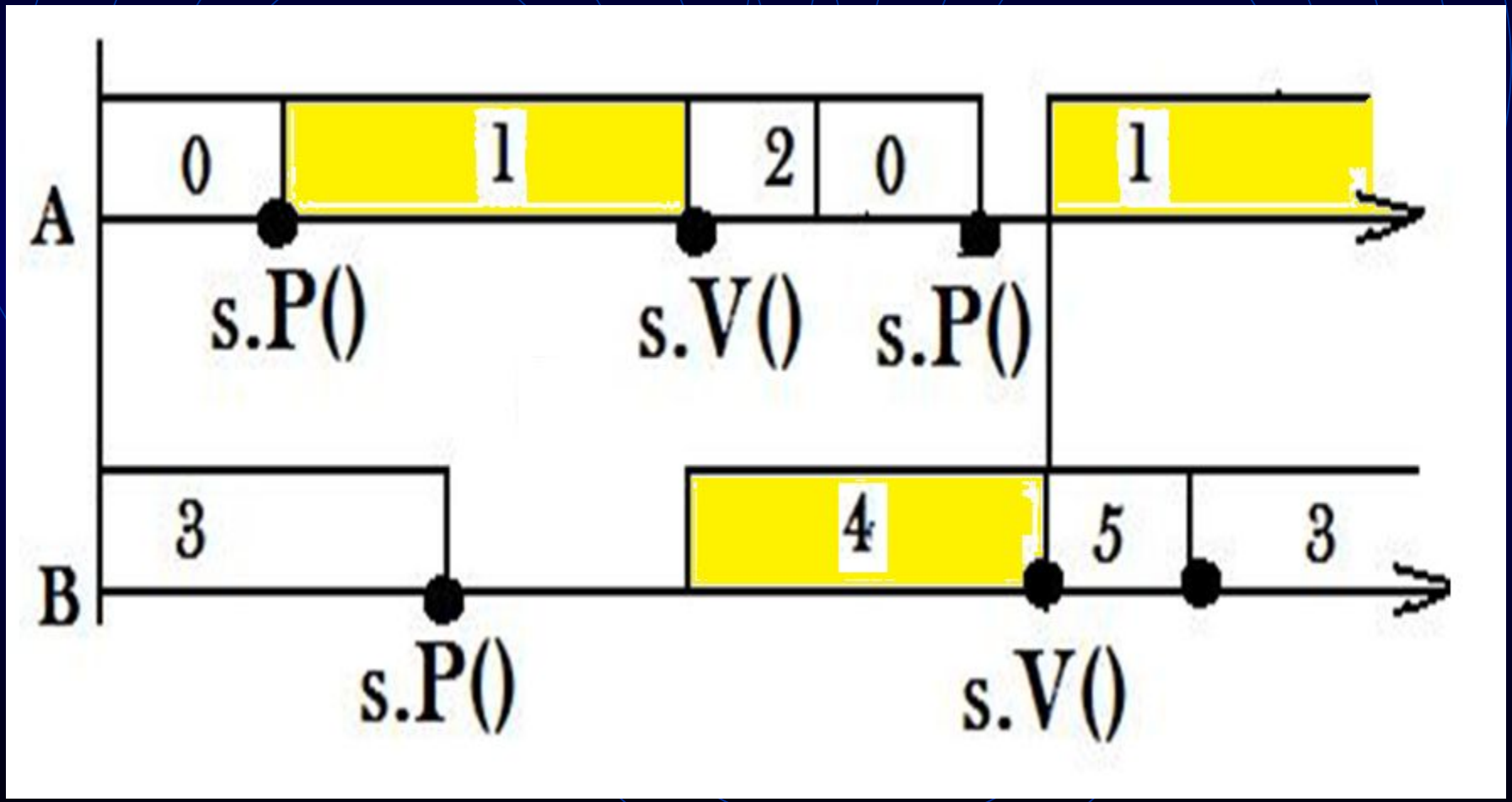
Семафор двоичный

TSemaphore s = new TSemaphore (1);
// семафор, видимый из двух процессов

// процесс A	**// процесс B**

while (true){
<действие 0>
S.P();
<действие 1>
S.V();
<действие 2>
}
	while (true){
	<действие 3>
	S.P();
	<действие 4>
	S.V();
	<действие 5>
	}

Семафор двоичный



Семафор двоичный

TSemaphore s = new TSemaphore (1);
// семафор, видимый из двух процессов

// процесс A	// процесс B

while (true){
<генерирует информацию>
S.P();
<запись информации в buf>
S.V();
}
	while (true){
	S.P();
	<Забирает данные из buf в локальную память>
	S.V();
	<Обрабатывает данные >
	}

Семафор двоичный

TSemaphore free = new TSemaphore(1);
TSemaphore empty = new TSemaphore(0);

// процесс A	// процесс B

while (true){
<генерирует информацию>
free.P();
<запись данных в buf>
empty.V();
}
	while (true){
	empty.P();
	<Забирает данные из buf>
	free.V();
	<Обрабатывает данные >
	}

Общий семафор

```
classTSemaphoreUniversal {
private:
    int count, maxCount;
public:
    void P(){while( ! count );    count --;}
    void V(){count ++;
        if (count > maxCount) count =maxCount ; }
    TSemaphoreUniversal (int start,  int m)  {
            count =start;  maxCount = m; }
~ TSemaphoreUniversal ();
  };
```

Семафор общий

```
 Int  n ;   //n – длина буфера
TSemaphoreUniversal  free  =
     new TSemaphoreUniversal (n, n);
   // общий семафор, в данный момент в буфер
   // можно записать n  порций информации

TSemaphoreUniversal empty  =
     new TSemaphoreUniversal (0, n);
   // общий семафор, из буфера можно будет прочитать
   // n порций,  в начальный момент  буфер пуст

TSemaphore  rw   =  new TSemaphore (1);
     // доступ к критической секции
```

Семафор общий

// процесс A	// процесс B

while (true){
  <Генерация порции>
  free.P();
  rw.P();
  <Запись в буфер>
  rw.V();
  empty.V();
}
	while (true){
	  empty.P();
	  rw.P();
	  <взять из буфера>
	  rw.V();
	  free.V();
	  <обработка >
	}

Семафор общий

Общим семафором можно искусственно ограничить многопоточность.

Слишком много одновременно работающих потоков могут заметно ухудшить производительность системы из-за частых переключений контекстов, поэтому число одновременно активных потоков можно ограничить числом процессоров.

Для этой цели каждый поток при своей активизации должен выполнить функцию P() над общим семафором, а при завершении потока вызывать функцию V().

Мьютекс (mutex)

- Синхронизирующий примитив для исключения доступа к критической секции для ПОТОКОВ одного приложения (локальный семафор).

Перед обращением к общим данным, мьютекс должен быть заблокирован методом lock, а после окончания работы с общими данными — разблокирован методом unlock.

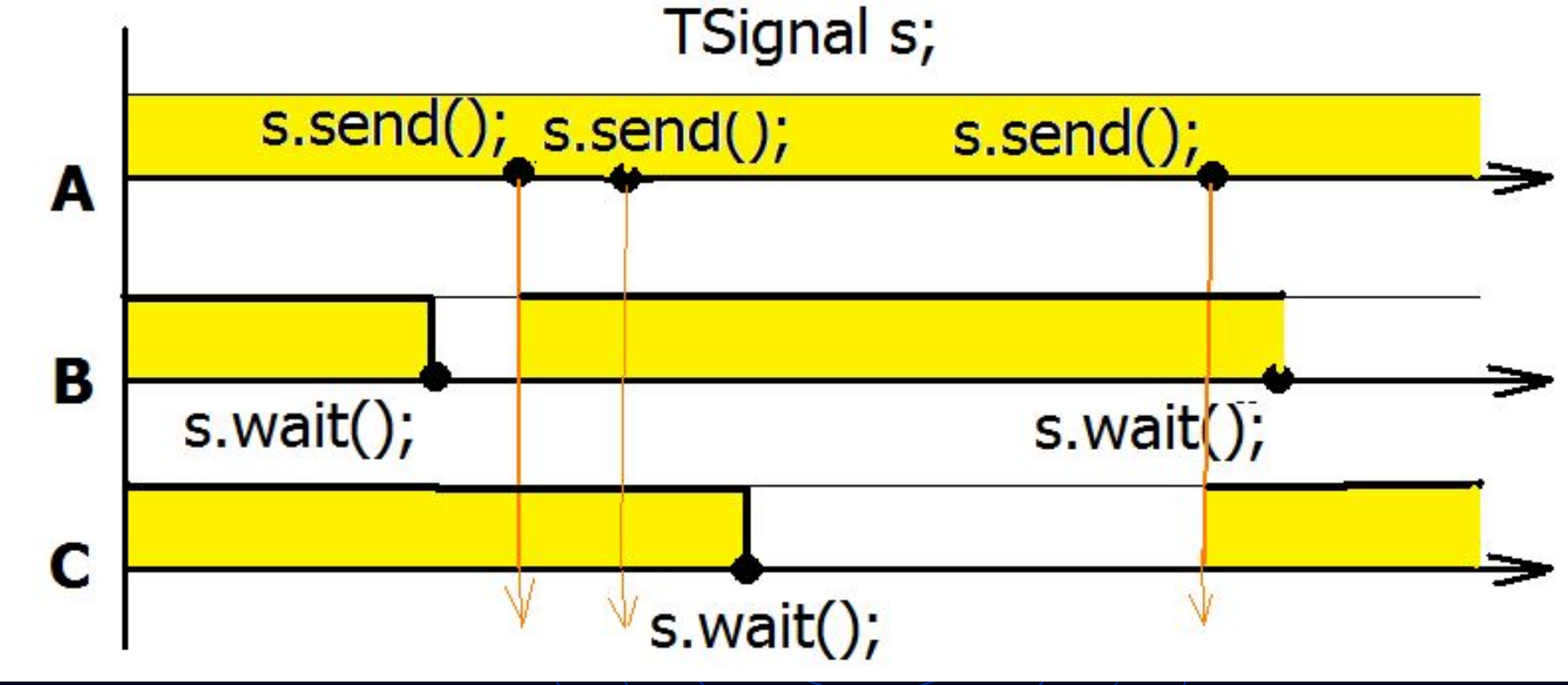
Мьютекс в C++11

```
#include <thread>
#include <mutex>

std::mutex   lockMutex;
    std::vector<T>   elements;

void add(T element)
    {
        lockMutex.lock();
        elements.push_back(element);
        lockMutex.unlock();
    }
```

Сигнал

```
classTSignal {
private:
     bool  flagWait;
public:
    void wait(){flagWait = true;
          while( flagWait; );
    }
    void send(){ flagWait = false; }
    TSignal ()  { flagWait = false;}
    ~ TSignal ();
};
```

Сигнал



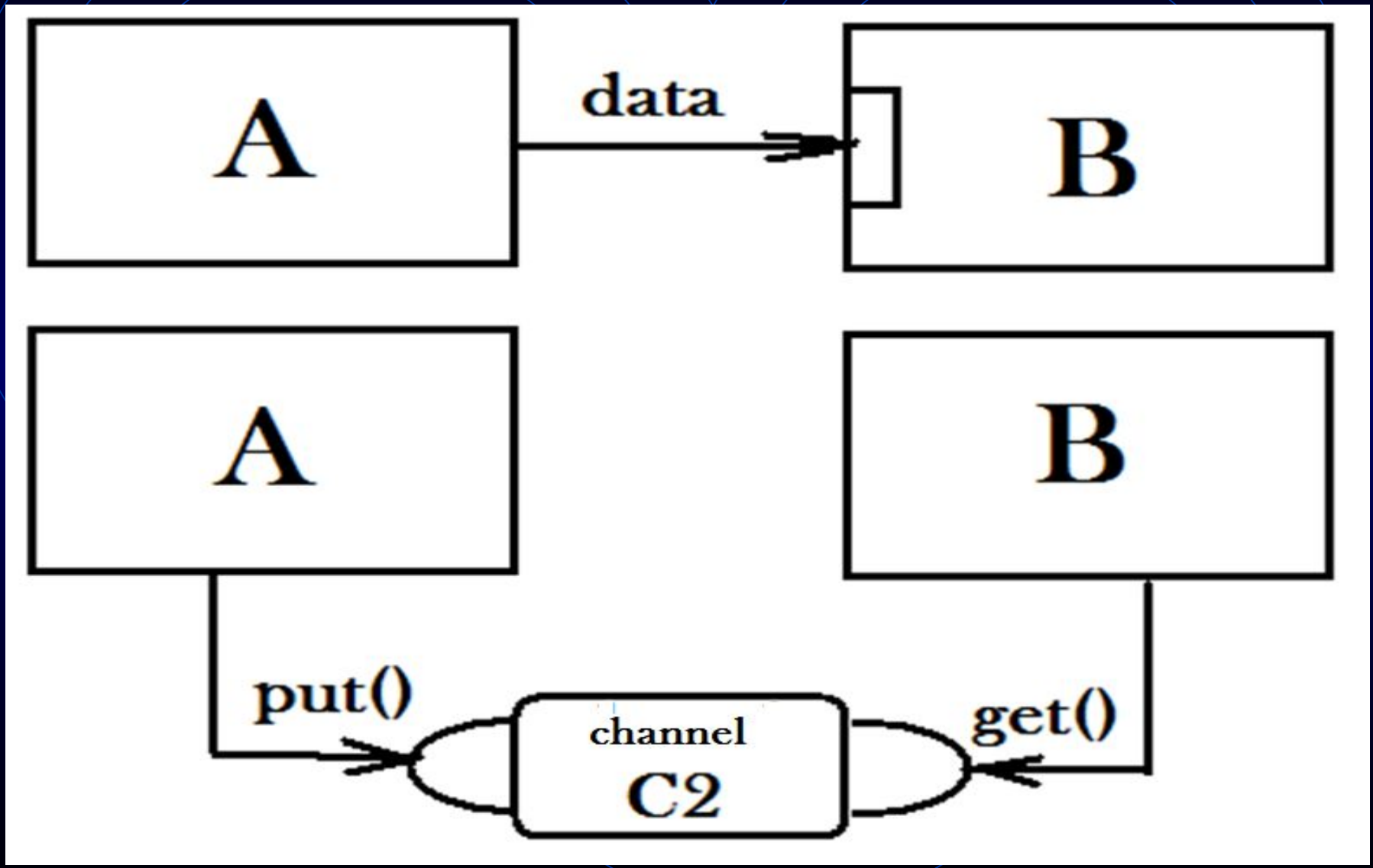
Канал без потерь

```
classTChannel {
private:
    bool free, empty;
    TData data;
private:
    void put(TData  t) ;
        TData get() ;
        TChannel (){free = true; empty =false;}
        ~TChannel(){}
};
```

Канал без потерь

```
void TChannel :: put(TData  t) { while(!free);
    memcpy(&data, &t, sizeof(data));
    empty = true;
}

TData  TChannel :: get() {
    while (!empty); free = true;
    return data;
}
```

Канал без потерь



Канал без потерь

```
classTChannel {
private:
    TSemaphore free;
        TSemaphore empty;
    TData data;
private:
    void put(TData  t);
        TData get(TData * resultData);
        TChannel ();
        ~TChannel(){}
};
```

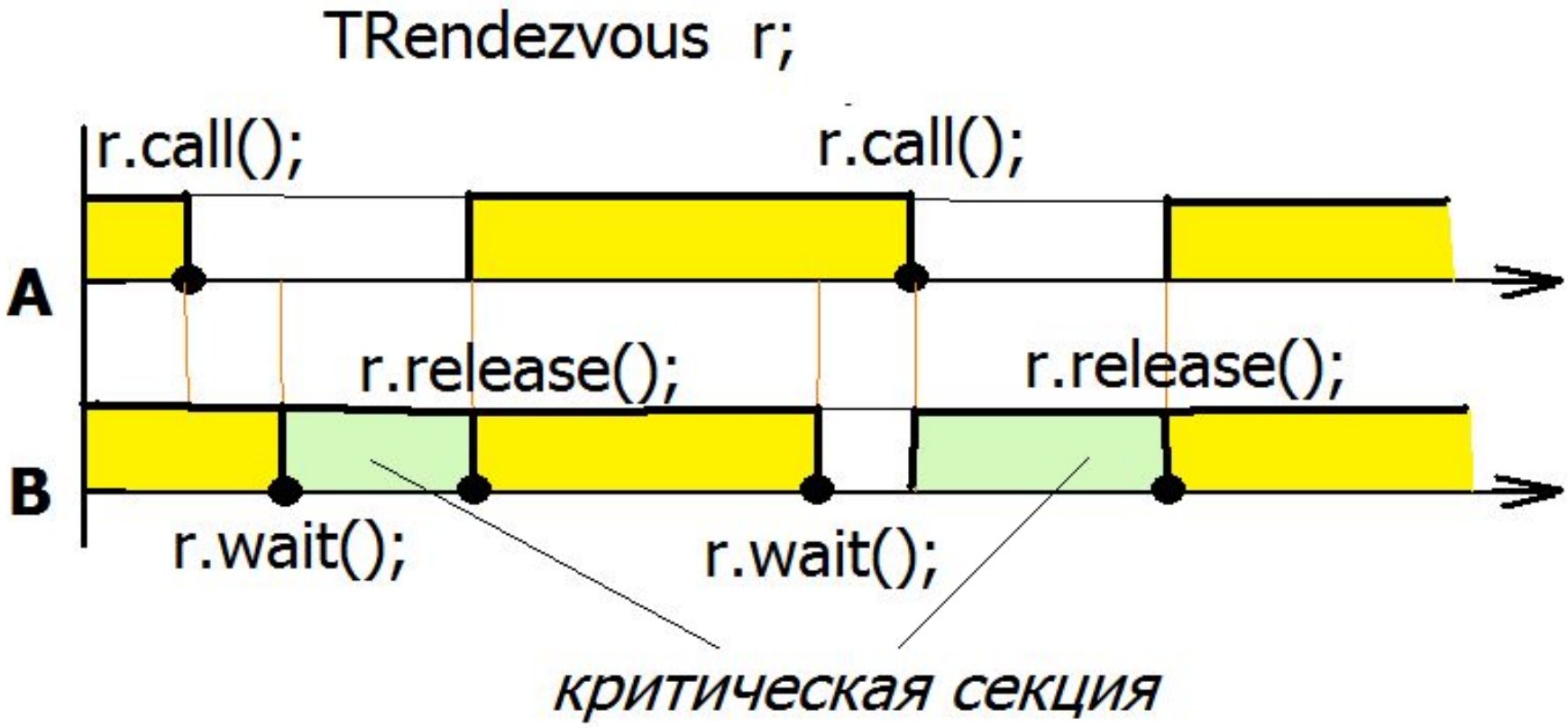
Рандеву

```
classTRendezvous {
private:
    int callFlag,  waitFlag,  releaseFlag;
public:
    void call();
    void  wait();
    void  release();
    TRendezvous () ;
    ~TRendezvous ();
};
```

Рандеву

```
void Trendezvous::call(){
    callFlag =1;
    while( ! waitFlag ); waitFlag =0;
    while( ! releaseFlag ); releaseFlag = 0;
}
void Trendezvous:: wait(){
      waitFlag   =1;
      while( ! callFlag );  callFlag  = 0;
}
void Trendezvous:: release(){   releaseFlag =1;
}
```

Рандеву


TRendezvous r;
r.call();
r.call();
A
r.release();
r.release();
B
r.wait();
r.wait();
критическая секция

Мониторы

Мониторы – это специальные программные модули, которые обеспечивают структурированность кода, являются объектом абстракции данных и обеспечивает набор операций, с помощью которых и только с их помощью обрабатываются внутренние данные, принадлежащие монитору.

Монитор используется для группировки представления и реализации разделяемого ресурса

Свойства монитора

1.монитор содержит методы, которые видимы снаружи, причем эти методы являются единственными видимыми извне именами

2. методы внутри монитора не могут обращаться к переменным вне монитора

3. когда некоторый процесс вызывает метод монитора, он становится активным. Взаимное исключение обеспечивается тем фактом, что в некоторый момент времени может быть активен только один метод

Барьерная синхронизация

Простейшуая реализация барьеров на основе функций API WaitForSingleObject() и WaitForMultipleObject().

Если параллельные обработчики работают не в рамках одного родительского процесса, требуется программная организация барьеров.

Типы барьеров

1. На основе разделяемого счетчика
2. С управляющим процессом
3. Симметричный барьер
   - Объединяющее дерево
   - Барьер – бабочка
   - Барьер с распространением сообщений
   - …

Барьер на основе разделяемого счетчика

```
Process (){
    While(true){     < обработка данных>      }
     // Sem -  семафор для счетчика барьера/
     Sem.p();   count--,   Sem.V();
     bool  flag = true;
     While(flag){
              Sem.p();  // не обязательно – нет гонки
             if ( ! count) flag = false;
             Sem.V();
             Sleep(TIMEWAIT);
     }
}
```

Барьер на основе управляющего процесса

```
Process (){
    While(true){     < обработка данных>      }
     processFlag[i]  =  true;  // нет гонки данных –
                    // у каждого процесса свой индекс
    // Sem -  семафор флага завершения
    bool  flag = true; // Закончил работу!!
    While(flag){    Sem.p();
                if ( GlobalFlag) flag = false;
                 Sem.V();       Sleep(TIMEWAIT);
     }
} // GlobalFlag)  - флаг управляющего процесса
   // ставится при всех processFlag[i]  == true
```

Симметричный барьер (дерево)

Организуем бинарное дерево рабочих процессов. Тогда каждый процесс выполняет код в соответствии со своим положением в дереве процессов

Симметричный барьер – лист дерева

this->processFlag = true;
// процесс с некоторым номером установил
//флаг завершения работы
<wait (this->continue == false)>
// ждем, пока наш флаг continue станет true
this->continue == false;
// дождались разрешения на продолжение
//и сбросили свой флаг

Симметричный барьер - промежуточный узел дерева

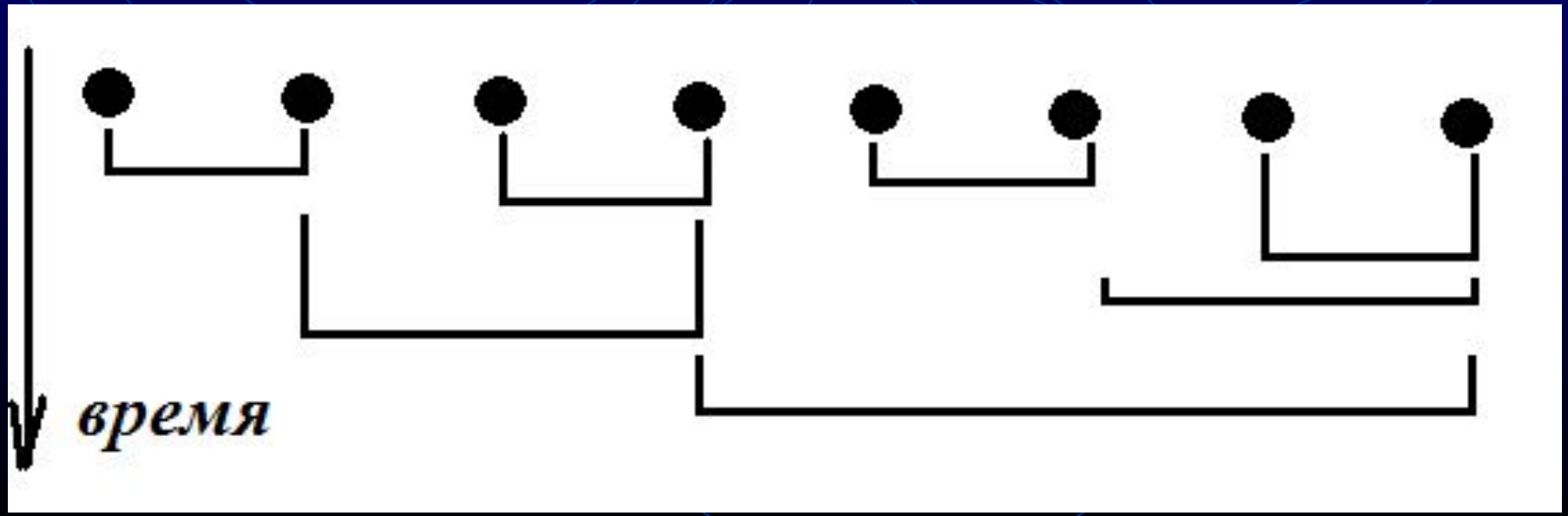
```
<wait (left -> processFlag == false)>
      // ждем, пока левый процесс работает
left -> processFlag  =  false;
 <wait (right -> processFlag  == false)>
left -> processFlag = false;
this->processFlag  =  true;  // свой флаг
<wait (this -> continue  == false)>
      // ждем, пока нам не разрешат продолжать
this -> continue  == false;
left -> continue  == true
right -> continue  == true;
```

Симметричный барьер – корень дерева

```
<wait (left -> processFlag == false)>
left -> processFlag  =  false;
 <wait (right -> processFlag  == false)>
left -> processFlag = false;
this-> continue  =  true;
  // установили свой флаг, если он нужен
   // это не обязательная операция

left -> continue  == true;
right -> continue  == true;
```

Симметричный барьер (бабочка)

Основан на использовании уровней синхронизации, когда некоторая пара процессов устанавливают и сбрасывают флаги друг друга/ Пара процессов модет быть выбрана динамически (не обязательно дерево)



Симметричный барьер (бабочка)

Для каждого уровня барьерной синхронизации используется свой набор флагов, чтобы процесс <i> дождался процесса <j> на нужном уровне синхронизации (возможно использование множества соответствующих значений флагов на каждом уровне).

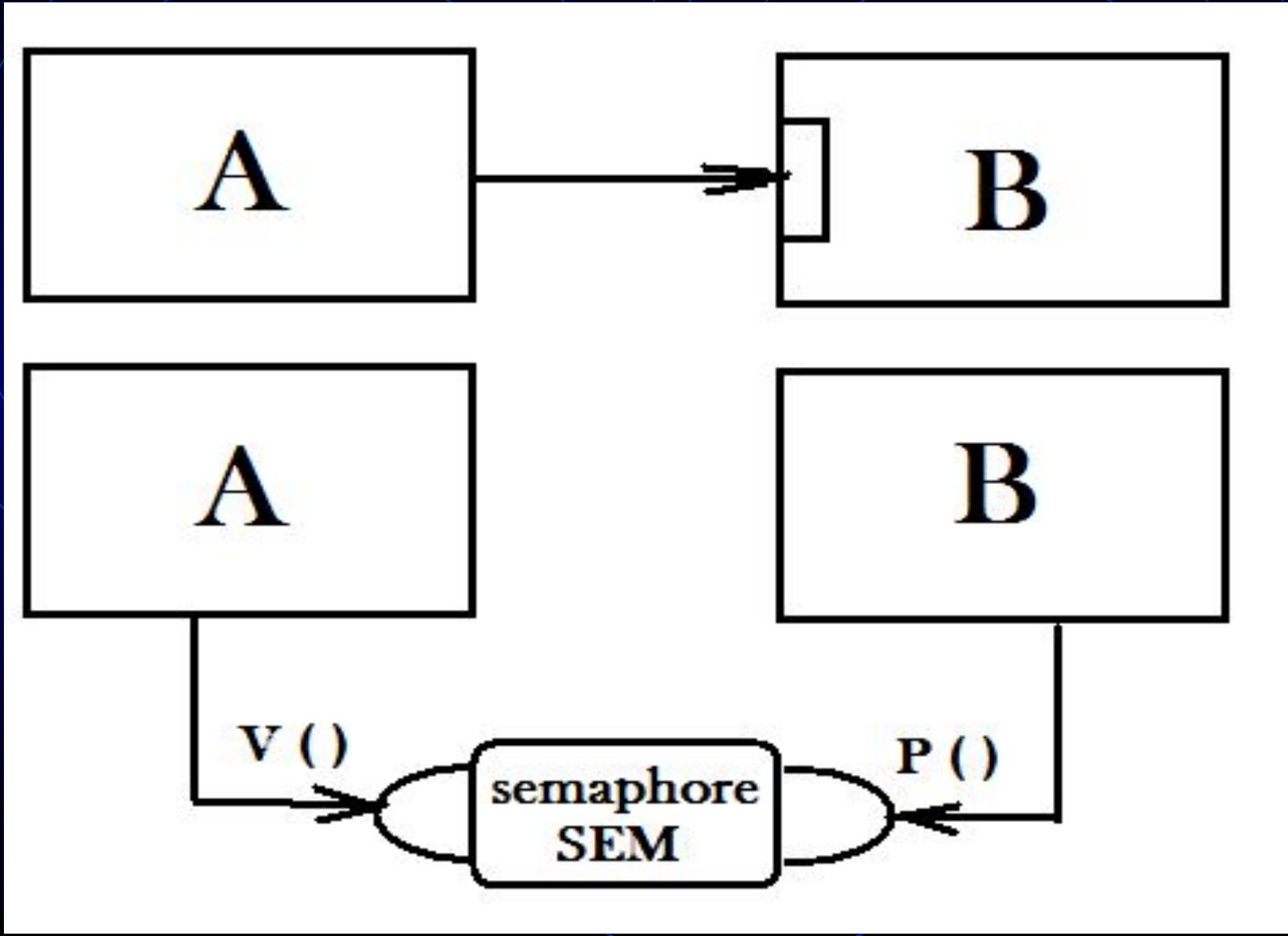
```
// барьер для процесса <i>
<wait (arrayFlag[i]) == 0>
arrayFlaf[i] = 1;
<wait (arrayFlag[j]) == 1>
arrayFlaf[j] = 0;
```

```
// барьер для процесса <j>
<wait (arrayFlag[j]) == 0>
arrayFlaf[j] = 1;
<wait (arrayFlag[i]) == 1>
arrayFlaf[i] = 0;
// для нового уровня
// барьера
```

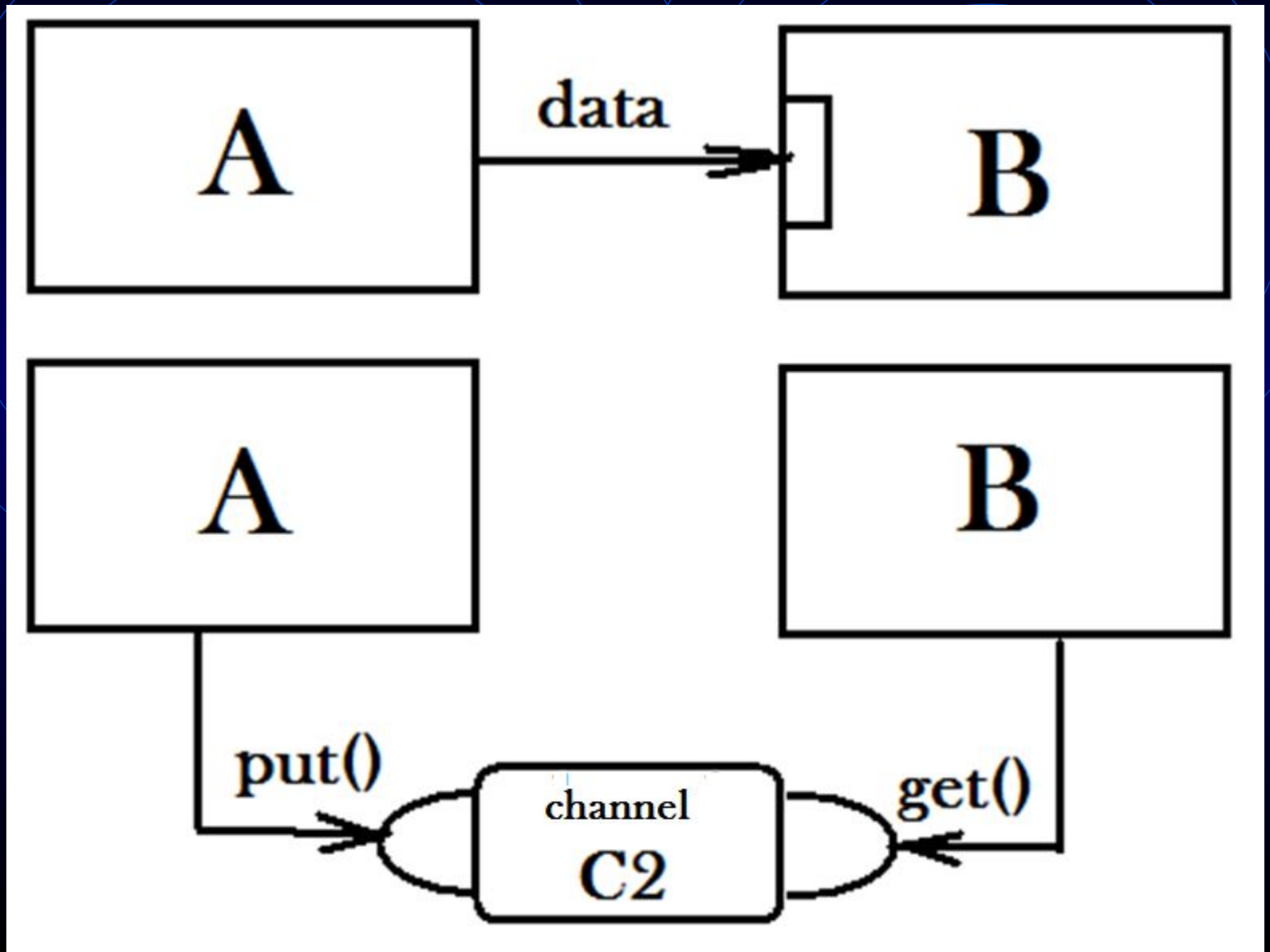
Лабораторная работа № 4

1. Выбрать примитивы синхронизации для организации каждого типа взаимодействия в системе. С осторожностью обращайтесь с примитивами с потерей синхронизирующей информации! Их неаккуратное использование может привести к дедлокам.
2. Перестроить схему в соответствии с выбранными примитивами
3. Построить UML-диаграмму «плавательные дорожки», на которых указаны выбранные примитивы синхронизации

Семафоры



Каналы



Каналы для альтернативного входа

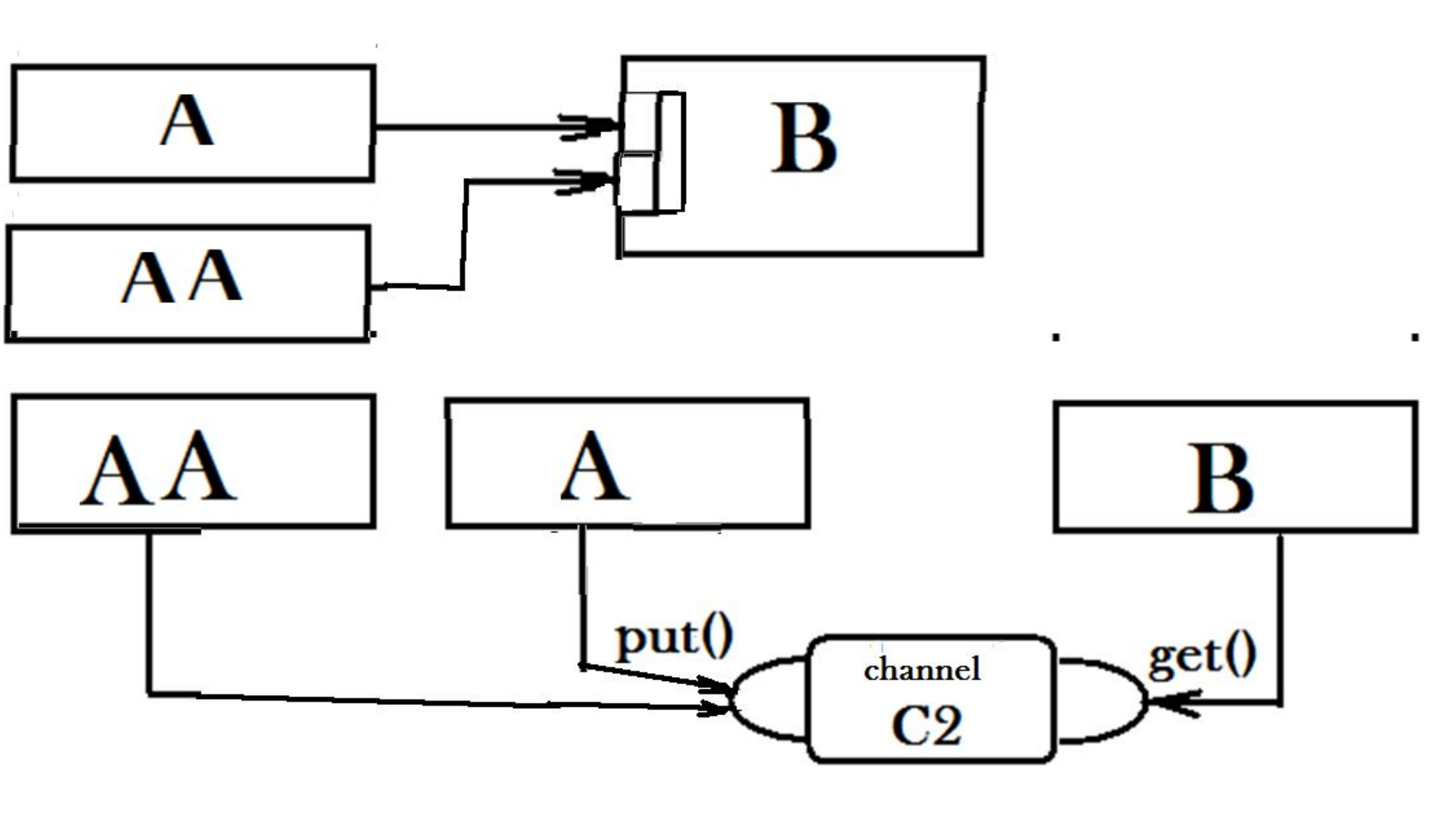


Диаграмма активности с параллельным выполнением

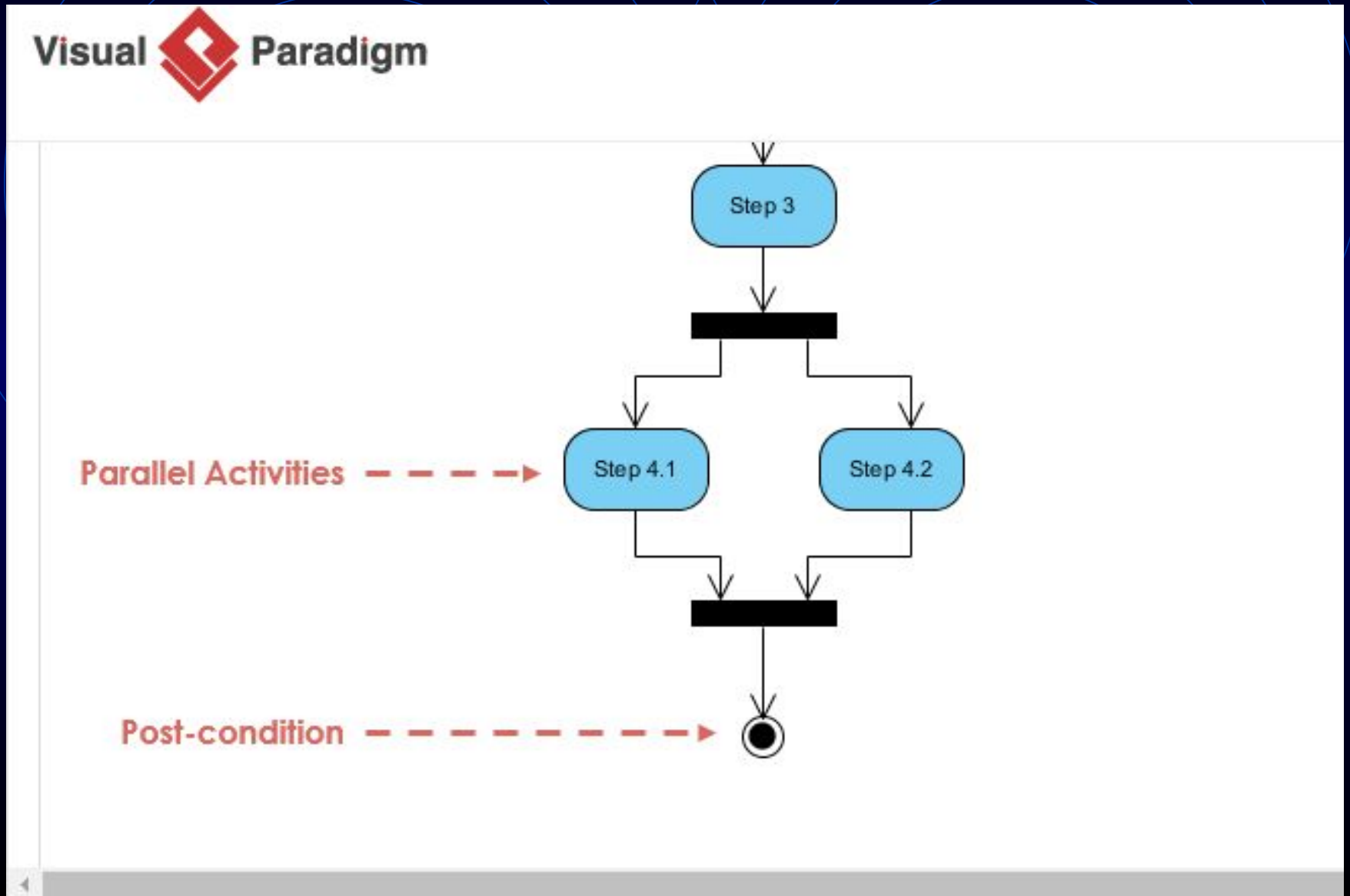


Диаграмма «Плавательные дорожки» - один поток

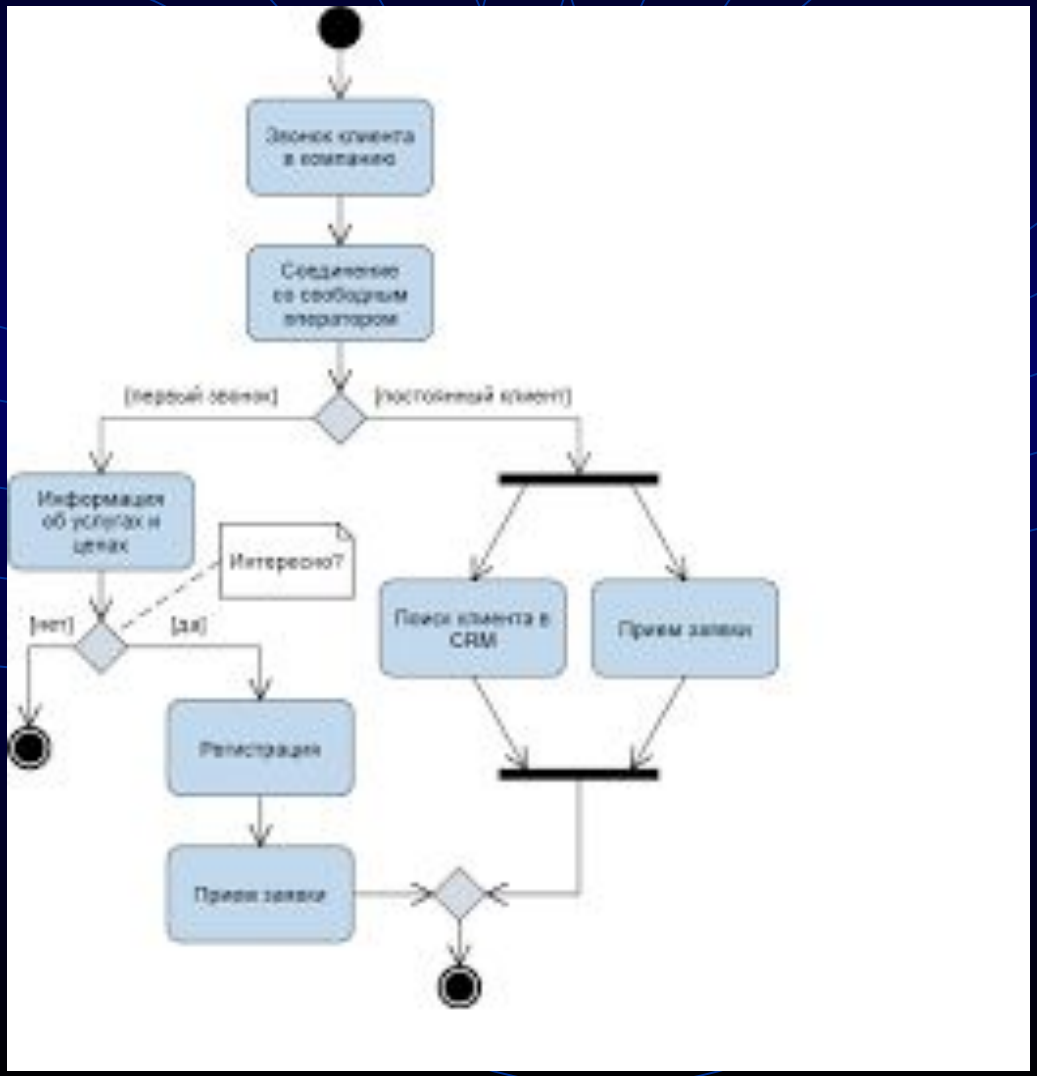


Диаграмма «Плавательные дорожки» взаимодействие потоков

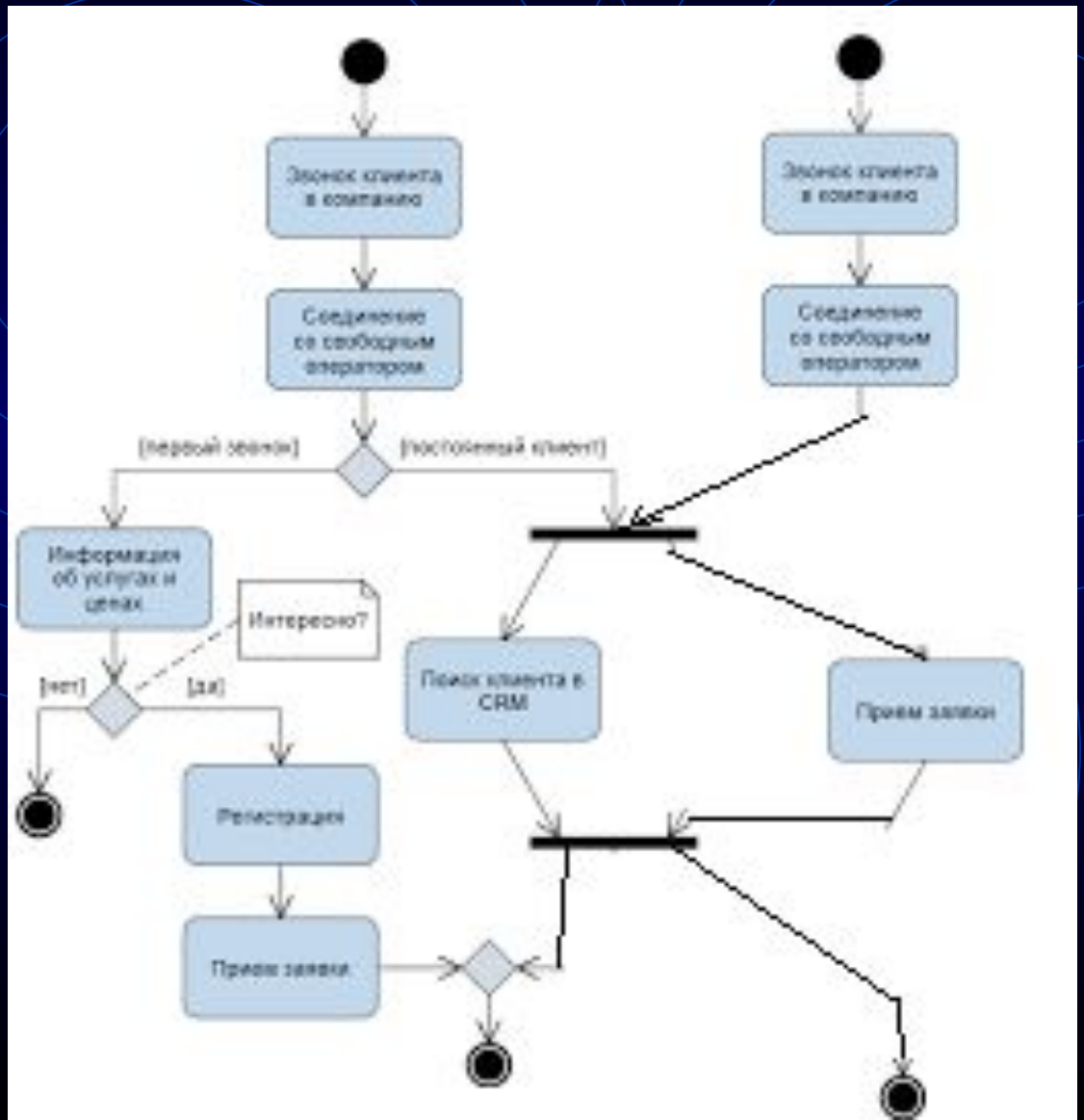
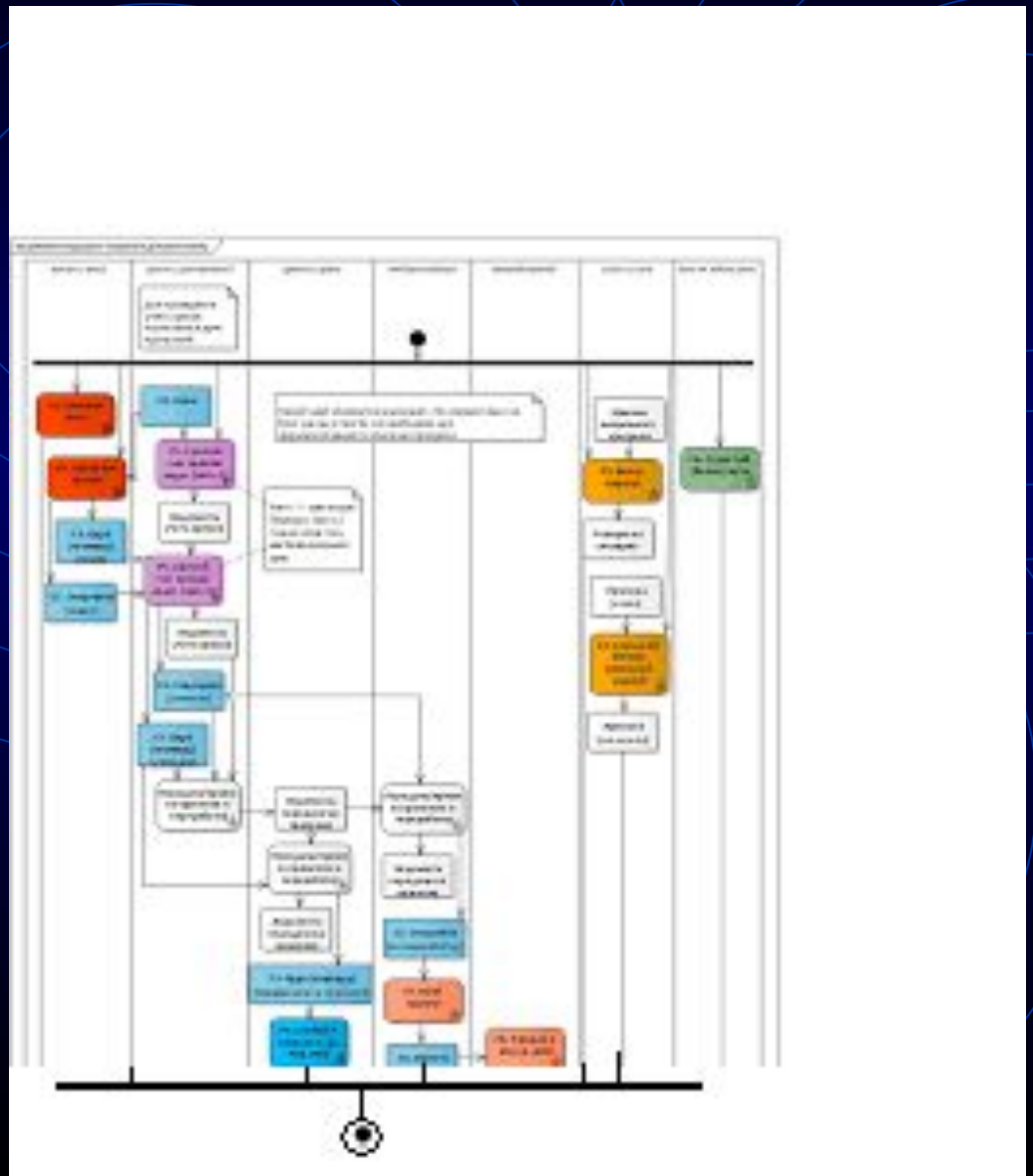


Диаграмма активности с барьером



Выбор семафор/канал:

- Есть передаваемая информация: канал.
- Разрешить продолжение выполнения, ничего не передавая: семафор.
- Альтернативный вход: канал, так как нам нужно знать, на какой подвход альтернативного входа поступил вызов.

Лабораторная работа 5 - будет после рассмотрения объектов ядра

1. Реализовать программу с потоками
2. Реализовать программу с отдельными приложениями