Основы современных операционных систем

Взаимодействующие (cooperating) процессы

- *Независимый* процесс – не может влиять на исполнение других процессов и испытывать их влияние.
- *Взаимодействующий (совместный)* процесс – может влиять на исполнение других процессов или испытывать их влияние
- Преимущества взаимодействующих процессов
  - Совместное использование данных
  - Ускорение вычислений
  - Модульность
  - Удобство

Виды процессов

- *Подчиненный* – зависит от процесса-родителя; уничтожается при его уничтожении; процесс-родитель должен ожидать завершения всех подчиненных процессов
- *Независимый* – не зависит от процесса-родителя; исполняется независимо от него (например, *процесс-демон: cron, smbd* и др.)
- *Сопроцесс (co-process, co-routine)* – хранит свое текущее *локальное управление* (program counter); взаимодействует с другим сопроцессом *Q* с помощью операций *resume (Q)*. Операция *detach* переводит сопроцесс в пассивное состояние (SIMULA-67). *Пример*: взаимодействие *итератора* с циклом

Проблема “производитель-потребитель” (producer – consumer)

- Одна из парадигм взаимодействия процессов: процесс-производитель (*producer*) генерирует информацию, которая используется процессом-потребителем (*consumer*)
  - *Неограниченный буфер (unbounded-buffer)* – на размер используемого буфера практически нет ограничений
  - *Ограниченный буфер (bounded-buffer)* – предполагается определенное ограничение размера буфера
  - Схема с ограниченным буфером, с точки зрения security, представляет опасность *атаки “buffer overruns”*. При заполнении буфера необходимо проверять его *размер*.

Ограниченный буфер – реализация с помощью общей памяти

- **Общие данные**

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- **Решение правильно, но могут использоваться только (BUFFER_SIZE-1) элементов**

Ограниченный буфер: процесс-производитель

```
item nextProduced;

while (1) {
        while (((in + 1) % BUFFER_SIZE)
== out)
            ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Ограниченный буфер: процесс-потребитель

```
item nextConsumed;

while (1) {
    while (in == out)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

Коммуникация процессов

Механизм для коммуникации процессов и синхронизации их действий.

***Система сообщений* – процессы взаимодействуют между собой без обращений к общим переменным.**

Средства взаимодействия между процессами обеспечивают две операции вида:

***send (message)* –отправка сообщения message; размер сообщения постоянный или переменный**

***receive (message)* — получение сообщенияв буфер message.**

Коммуникация процессов

Если *P* и *Q* требуется взаимодействовать между собой, им необходимо:

Установить связь (communication link) друг с другом

Обменяться сообщениями вида send/receive

Реализация связи

Физическая (общая память, аппаратная шина)

Логическая (например, логические свойства)

Реализация коммуникации процессов

- Как устанавливается связь?
- Можно ли установить связь более чем двух процессов?
- Сколько связей может быть установлено между двумя заданными процессами?
- Какова пропускная способность линии связи?
- Является ли длина сообщения по линии связи постоянной или переменной?
- Является ли связь ненаправленной или двунаправленной (дуплексной)?

Прямая связь (direct communication)

- **Процессы именуют друг друга явно:**
  - **send (*P, message*)** – послать сообщение процессу P
  - **receive(*Q, message*)** – получить сообщение от процесса Q
- **Свойства линии связи**
  - **Связь устанавливается автоматически.**
  - **Связь ассоциируется только с одной парой взаимодействующих процессов.**
  - **Между каждой парой процессов всегда только одна связь.**
  - **Связь может быть ненаправленной, но, как правило, она двунаправленная.**

Косвенная связь (indirect communication)

- Сообщения направляются и получаются через почтовые ящики (порты) – mailboxes; ports
  - Каждый почтовый ящик имеет уникальный идентификатор.
  - Процессы могут взаимодействовать, только если они имеют общий почтовый ящик.
- Свойства линии связи
  - Связь устанавливается, только если процессы имеют общий почтовый ящик
  - Связь может быть установлена со многими процессами.
  - Каждая пара процессов может иметь несколько линий связи.
  - Связь может быть ненаправленной или двунаправленной.

Косвенная связь

- **Операции**
  - **Создать новый почтовый ящик**
  - **Отправить (принять) сообщение через почтовый ящик**
  - **Удалить почтовый ящик**
- **Основные операции:**

  ***send (A, message*) – послать сообщение в почтовый ящик A**

  ***receive (A, message*) – получить сообщение из почтового ящика A**

Синхронизация при косвенной связи

- Передача сообщений может выполняться с блокировкой или без блокировки
- Передача с блокировкой - синхронная
- Передача без блокировки - асинхронная
- Основные операции send и receive могут быть с блокировкой или без блокировки

Буферизация

- **С коммуникационной линией связывается *очередь сообщений*, реализованная одним из трех способов:**

1. **Нулевая емкость – 0 сообщений**
   Отправитель должен ждать получателя (рандеву — rendezvous).
2. **Ограниченная емкость – конечная длина очереди: *n* сообщений (предотвратить опасность атаки “buffer overruns”!)**
   Отправитель должен ждать, если очередь заполнена.
3. **Неограниченная емкость – бесконечная длина.**
   Получатель никогда не ждет.

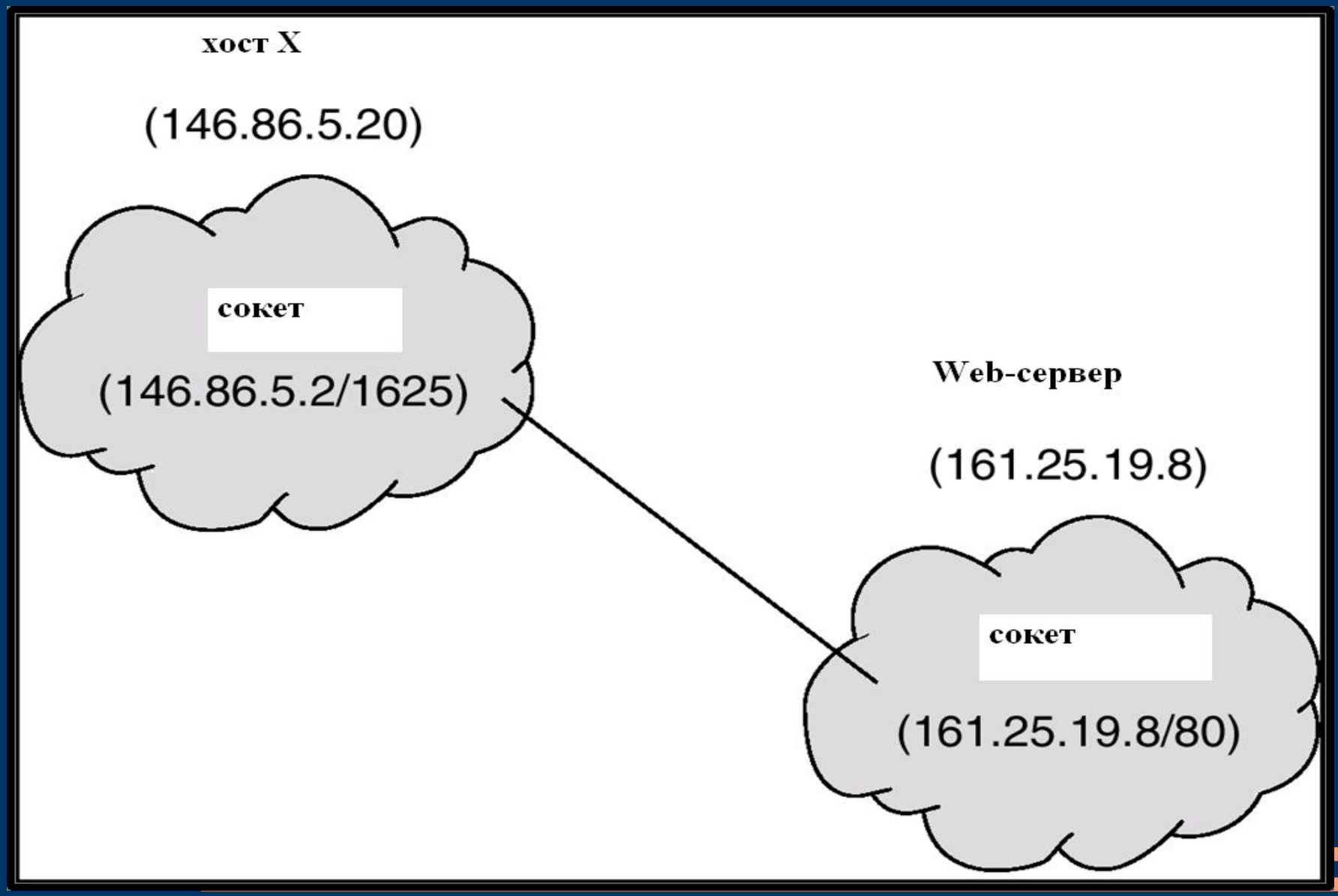
Клиент-серверная взаимосвязь

- **Сокеты (Sockets)**
- **Удаленные вызовы процедур (Remote Procedure Calls – RPC)**
- **Удаленные вызовы методов (Remote Method Invocation – RMI) : Java**

Сокеты (Sockets)

- **Впервые были реализованы в UNIX BSD 4.2**
- **Сокет можно определить как отправную (конечную) точку для коммуникации - *endpoint for communication*.**
- **Конкатенация IP-адреса и порта**
- **Сокет 161.25.19.8:1625 ссылается на порт 1625 на машине (хосте) 161.25.19.8**
- **Коммуникация осуществляется между парой сокетов**

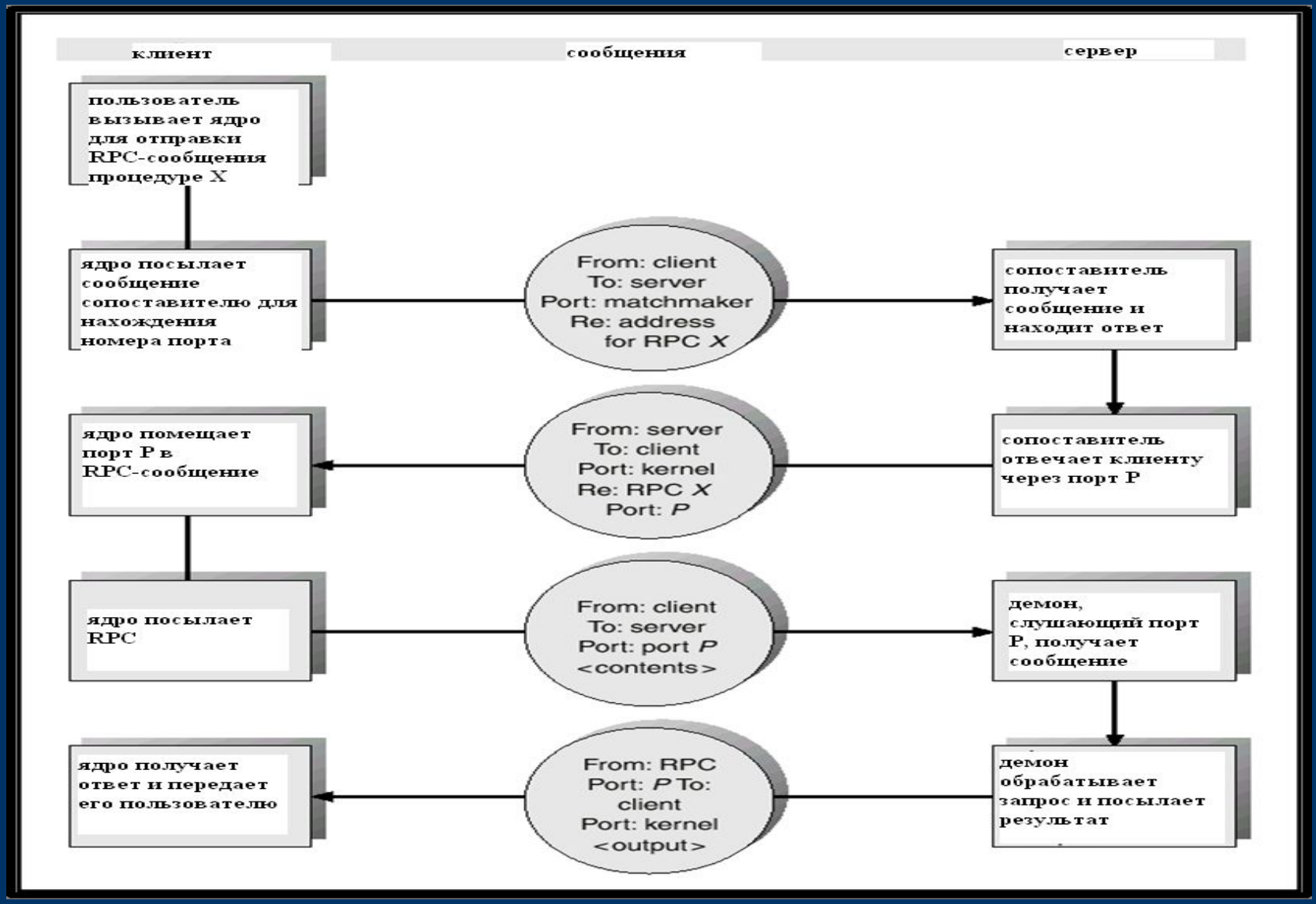
Взаимодействие с помощью сокетов



Удаленные вызовы процедур (RPC)

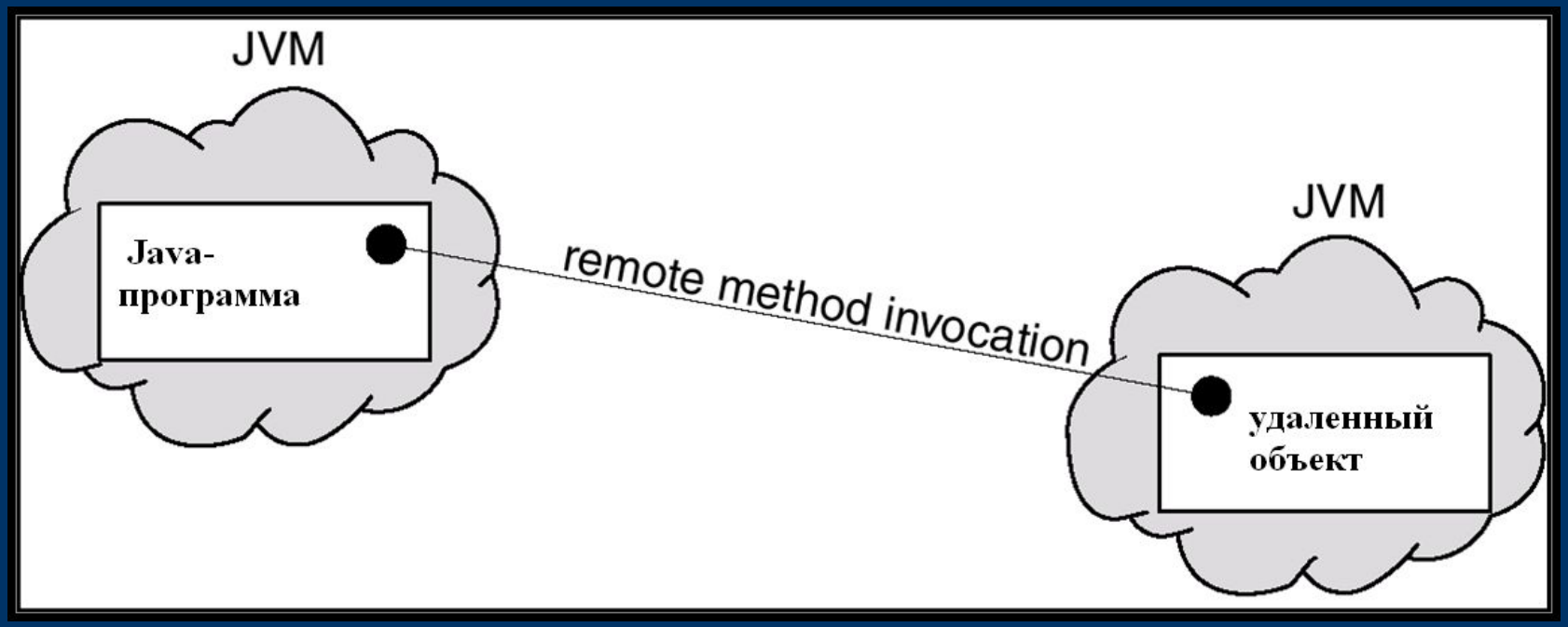
- **RPC впервые предложен фирмой Sun и реализован в ОС Solaris**
- **Удаленный вызов процедуры (RPC) – абстракция вызова процедуры между процессами в сетевых системах**
- **Заглушки (Stubs) – proxy в клиентской части для фактической процедуры, находящейся на сервере**
- **Заглушка в клиентской части находит сервер и выстраивает (*marshals*) параметры.**
- **Заглушка в серверной части принимает это сообщение, распаковывает параметры, преобразует их к нормальному виду и выполняет процедуру на сервере**

Исполнение RPC

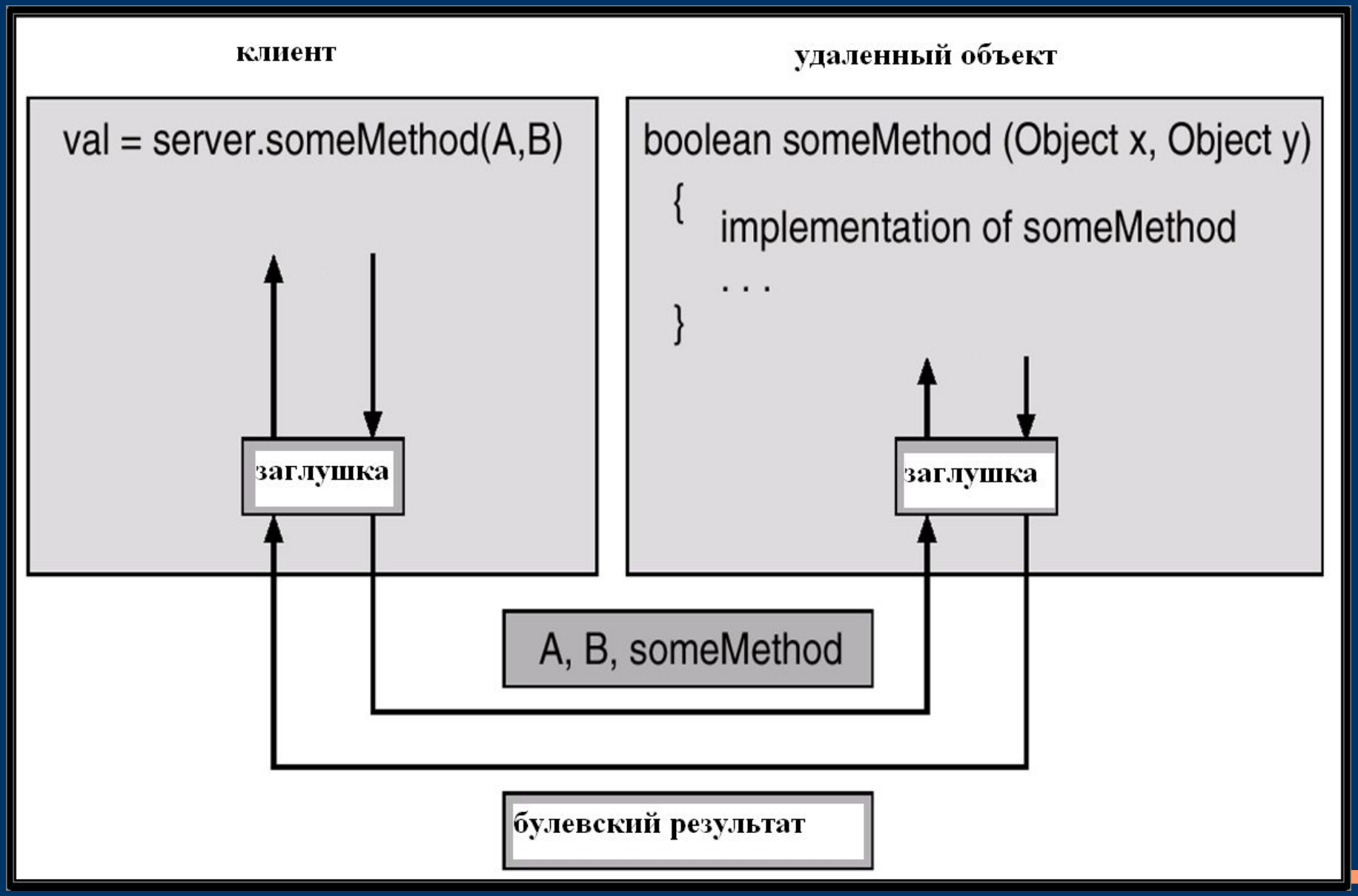


Удаленный вызов метода (RMI) - Java

- **Remote Method Invocation (RMI) – механизм в Java-технологии, аналогичный RPC**
- **RMI позволяет Java-приложению на одной машине вызвать метод удаленного объекта.**



Выстраивание параметров (marshaling)



Синхронизация процессов

- История
- Проблема критической секции
- Аппаратная поддержка синхронизации
- Семафоры
- Классические проблемы синхронизации
- Критические области
- Мониторы
- Синхронизация в Solaris и в Windows

История

- **Совместный доступ к общим данным может привести к нарушению их целостности (inconsistency).**
- **Поддержание целостности общих данных требует механизмов упорядочения работы взаимодействующих процессов (потоков).**
- **Решение проблемы общего буфера с помощью глобальной (общей) памяти допускает, чтобы не более чем *n* – 1 элементов данных могли быть записаны в буфер в каждый момент времени.**
  - **Предположим, что в системе производитель/потребитель мы модифицируем код, добавляя переменную *counter*, инициализируемую 0 и увеличиваемую каждый раз, когда в буфер добавляется новый элемент данных**

Ограниченный буфер: Представление

Общие данные

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Ограниченный буфер: Производитель

- **Процесс-производитель**

```
item nextProduced;
 while (1) {
 while (counter == BUFFER_SIZE)
      ; /* do nothing */
 buffer[in] = nextProduced;
 in = (in + 1) % BUFFER_SIZE;
 counter++;
}
```

Ограниченный буфер: Потребитель

- **Процесс-потребитель**

```
item nextConsumed;
 while (1) {
 while (counter == 0)
       ; /* do nothing */
 nextConsumed = buffer[out];
 out = (out + 1) % BUFFER_SIZE;
 counter--;
}
```

Ограниченный буфер: Атомарность операций над *counter*

- **Операторы**

  counter++;
  counter--;

  должны быть выполнены атомарно (*atomically*).

- **Атомарная операция – такая, которая должна быть выполнена полностью без каких-либо прерываний. При этом, операция, выполняемая одним из процессов, является неделимой, с точки зрения другого процесса**

Ограниченный буфер: Реализация операций над *counter*

Оператор “count++” может быть реализован на языке ассемблерного уровня как:

```
register1 = counter

register1 = register1 + 1
 counter = register1
```

- **Оператор “count—” может быть реализован как:**

```
register2 = counter
register2 = register2 – 1
counter = register2
```

Ограниченный буфер: Совместное обращение (interleaving)

- **Если и производитель, и потребитель пытаются обратиться к буферу совместно (одновременно), то указанные ассемблерные операторы также должны быть выполнены совместно (interleaved).**

- **Реализация такого совместного выполнения зависит от того, каким образом происходит планирование для процессов – производителя и потребителя.**

Ограниченный буфер: Эффект interleaving

- **Предположим, counter вначале равно 5. Исполнение процессов в совместном режиме (interleaving) приводит к следующему:**

  producer: register1 = counter (*register1 = 5*)
  producer: register1 = register1 + 1 (*register1 = 6*)
  consumer: register2 = counter (*register2 = 5*)
  consumer: register2 = register2 – 1 (*register2 = 4*)
  producer: counter = register1 (*counter = 6*)
  consumer: counter = register2 (*counter = 4*)

- **Значение counter может оказаться равным 4 или 6, в то время как правильное значение counter равно 5.**

Конкуренция за общие данные (race condition)

- **Race condition: Ситуация, когда взаимодействующие процессы могут обращаться к общим данным совместно (параллельно). Конечное значение общей переменной зависит от того, какой процесс завершится первым.**

- **Для предотвращения подобных ситуаций процессы следует *синхронизировать*.**

Проблема критической секции

- ***n*** **процессов – каждый может обратиться к общим данным**
- **Каждый процесс имеет участок кода, называемый *критической секцией,* в котором происходит обращение к общим данным.**
- **Проблема – обеспечить, чтобы, если один процесс вошел в свою критическую секцию, никакой другой процесс не мог бы одновременно войти в свою критическую секцию.**

Решение проблемы критической секции

1. ***Взаимное исключение*.** **Если процесс *Pi* исполняет свою критическую секцию, то никакой другой процесс не должен в тот же момент времени исполнять свою.**

2. ***Прогресс*.** **Если в данный момент нет процессов, исполняющих критическую секцию, но есть несколько процессов, желающих начать исполнение критической секции, то выбор системой процесса, которому будет разрешен запуск критической секции, не может продолжаться бесконечно.**

Решение проблемы критической секции

3. *Ограниченное ожидание*. Должно существовать ограничение на число раз, которое процессам разрешено входить в свои критические секции, после того как некоторый процесс сделал запрос о входе в критическую секцию, и до того, как этот запрос удовлетворен.

Предполагается, что каждый процесс исполняется с ненулевой скоростью

Не делается никаких специальных предположений о соотношении скоростей каждого из *n* процессов.

Первоначальные попытки решения проблемы

- Есть только два процесса, *P*0 и *P*1
- Общая структура процесса *Pi*:

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (1);
```

- Процессы могут использовать общие переменные для синхронизации своих действий.

Алгоритм 1

- **Общие переменные:**
  - **int turn;**
    первоначально turn = 0
  - **turn == i процесс *Pi* может войти в критическую секцию**
- **Процесс *Pi* :**

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                remainder section
} while (1);
```

- **Удовлетворяет принципу “взаимное исключение”, но не принципу “прогресс”**

Алгоритм 2

- **Общие переменные**
  - **boolean flag[2];**
    первоначально flag [0] = flag [1] = false.
  - **flag [i] == true ⇒ *Pi* готов войти в критическую секцию**
- **Процесс *Pi* :**

```
do {

    flag[i] := true;
    while (flag[j]) ;

    flag [i] = false;

    remainder section

} while (1);
```

critical section

- **Удовлетворяет принципу “взаимное исключение”, но не принципу “прогресс”**

Алгоритм 3

- Объединяет общие переменные алгоритмов 1 и 2.
- Процесс Pi :

```
do {

    flag [i]:= true;
    turn = j;
    while (flag [j] and turn = j) ;

    critical section

    flag [i] = false;

    remainder section

} while (1);
```

- Удовлетворяет всем трем принципам и решает проблему взаимного исключения.

Алгоритм булочной (bakery algorithm) – L. Lamport

- Происхождение названия: реализована стратегия, подобная стратегии обслуживания клиентов в булочной, где каждому клиенту автоматически присваивается его номер в очереди
- Обозначения: $<$ □ лексикографический порядок
- $(a,b) < (c,d)$ если $a < c$ or if $a = c$ and $b < d$
  - max ($a0$,…, an-1)- число k, такое, что k □ ai for i - 0, …, n – 1
- Общие данные:

```
boolean choosing[n];

int number[n];
```

Структуры данных инициализируются, соответственно, false и 0

Алгоритм булочной

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while ((number[j] != 0) && (number[j] < number[i])) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
```

Аппаратная поддержка синхронизации

- Атомарная операция проверки и модификации значения переменной

.

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}
```

Взаимное исключение с помощью TestAndSet

- Общие данные:
  boolean lock = false;

- Процесс *Pi*

```
do {
        while (TestAndSet(lock)) ;
                critical section
        lock = false;
                remainder section
}
```

Аппаратное решение для синхронизации

- Атомарная перестановка значений двух переменных.

```
void Swap (boolean * a, boolean * b) {
    boolean temp = * a;
    * a = * b;
    * b = temp;
}
```

Взаимное исключение с помощью Swap

- Общие данные (инициализируемые **false**):
  boolean lock;
  boolean waiting[n];

- Процесс *Pi*

```
do {
        key = true;
        while (key == true)
                Swap(&lock, &key);
            critical section
        lock = false;
            remainder section
}while (1)
```

Общие семафоры – counting semaphores (по Э. Дейкстре)

- Семафоры
- Средство синхронизации, не требующее активного ожидания.
- (Общий) семафор *S* – целая переменная
- Может использоваться только для двух атомарных операций:

wait (*S*):

```
while S≤ 0 do no-op;
    S--;
```

signal (*S*):

```
S++;
```

Критическая секция для N процессов

- Общие данные:

  semaphore mutex; //initially *mutex* = 1

- Процесс *Pi:*

```
do {
    wait(mutex);
        critical section

    signal(mutex);
        remainder section
} while (1);
```

Реализация семафора

- Определяем семафор как структуру:

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- Предполагаем наличие двух простейших операций:
  - **block** – задерживает исполнение процесса, исполнившего данную операцию.
  - **wakeup(*P*)** возобновляет исполнение приостановленного процесса **P**.

Реализация

- Определим семафорные операции следующим образом:

```
wait(S):
        S.value--;

 if (S.value < 0) {

            добавить  текущий процесс к S.L;
            block;

        }
 signal(S):
        S.value++;

 if (S.value <= 0) {

            удалить процесс P из S.L;
            wakeup(P);

        }
```

Семафоры как общее средство синхронизации

- Исполнить действие *B* в процессе *P*j только после того, как действие *A* исполнено в процессе *Pi*
- Использовать семафор *flag* , инициализированный 0
- Код:

Pi	*Pj*
□	□
A	*wait*(*flag*)
signal(*flag*)	*B*

Два типа семафоров

- *Общий семафор (Counting* semaphore) – целое значение, теоретически неограниченное
- *Двоичный семафор (Binary* semaphore) – целое значение, которое может быть только 0 или 1; возможно, проще реализуется (семафорный бит – как в Burroughs и “Эльбрусе”)
- Общий семафор S может быть реализован с помощью двоичного семафора.

Вариант операции wait(S) для системных процессов (“Эльбрус”)

- Для системного процесса лишние прерывания нежелательны и может оказаться важным удерживать процессор за собой на какое-то время

- Операция *ЖУЖ(S);* (вместо *ЖДАТЬ(S);*) – процесс не прерывается и “жужжит” на процессоре, пока семафор S не будет открыт операцией *ОТКРЫТЬ*

Реализация общего семафора S с помощью двоичных семафоров

- Структуры данных:
  binary-semaphore S1, S2;
  int C:
- Инициализация:
  S1 = 1
  S2 = 0
  C = начальное значение общего семафора **S**

Реализация операций над семафором S

Операция *wait:*

```
wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

Операция *signal*:

```
wait(S1);
C ++;
if (C <= 0)
    signal(S2);
}
signal(S1);
```

Классические задачи синхронизации

- Задача “ограниченный буфер” (Bounded-Buffer Problem)
- Задача “читатели-писатели” (Readers and Writers Problem)
- Задача “обедающие философы” (Dining-Philosophers Problem)

Задача “ограниченный буфер”

- Общие данные:

semaphore full = n;

semaphore empty =0;

semaphore mutex =1;

Процесс-производитель ограниченного буфера

```
do {
        …
    сгенерировать элемент в nextp
        …
    wait(empty);
    wait(mutex);
        …
    добавить nextp к буферу
        …
    signal(mutex);
    signal(full);
} while (1);
```

Процесс-потребитель ограниченного буфера

```
do {
    wait(full)
    wait(mutex);
        …
    взять (и удалить) элемент из буфера в nextc
        …
    signal(mutex);
    signal(empty);
        …
    использовать элемент из nextc
        …
} while (1);
```

Задача “читатели-писатели”

- Общие данные:

```
semaphore mutex = 1;
semaphore wrt = 1;
int readcount = 0;
```

Процесс-писатель

wait(wrt);

…

выполняется запись

…

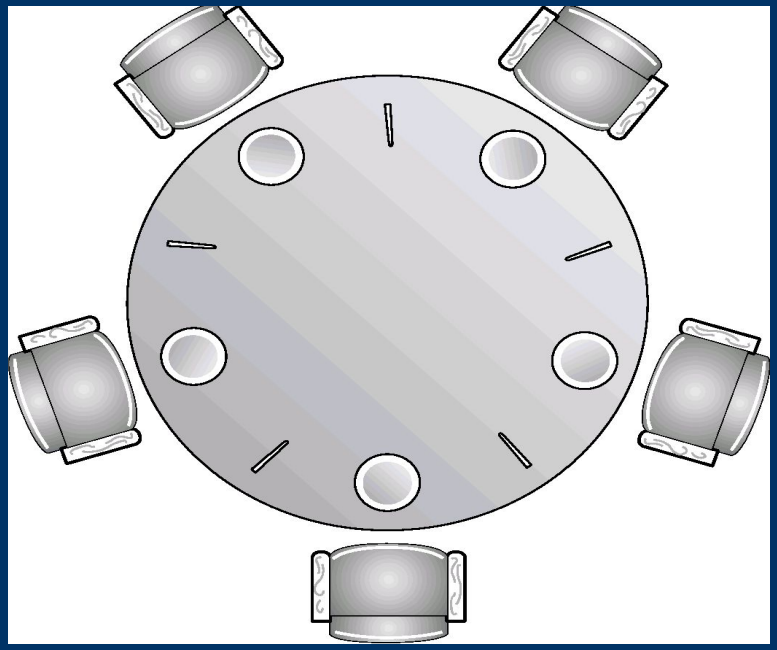
signal(wrt);

Процесс-читатель

```
wait(mutex);
readcount++;
if (readcount == 1)
      wait(rt);
signal(mutex);
            …
     выполняется чтение
            …
wait(mutex);
readcount--;
if (readcount == 0)
      signal(wrt);

signal(mutex):
```

Задача “обедающие философы”



- Общие данные

  semaphore chopstick[5] = {1, 1, 1, 1, 1};

Первоначально все значения равны 1

Задача “обедающие философы”

- **Философ *i*:**

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        …
      dine
        …
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        …
      think
        …
} while (1);
```

Критические области (critical regions)

- Высокоуровневая конструкция для синхронизации
- Общая переменная v типа *T*, определяемая следующим образом:

  v: shared T

- К переменной v доступ возможен только с помощью специальной конструкции:

  region v when B do S

  где *B* – булевское выражение.

- Пока исполняется оператор *S*, больше ни один процесс не имеет доступа к переменной *v*.

Пример: ограниченный буфер

- Общие данные:

```
struct buffer {
    int pool[n];
    int count, in, out;
}
```

Процесс-производитель

- Процесс-производитель добавляет **nextp** к общему буферу

```
region buffer when (count < n)
{
        pool[in] = nextp;
        in:= (in+1) % n;
        count++;
    }
```

Процесс-потребитель

- Процесс-потребитель удаляет элемент из буфера и запоминает его в **nextc**

```
region buffer when (count > 0) {
        nextc = pool[out];
        out = (out+1) % n;
        count--;
 }
```

Реализация оператора region *x* when *B* do *S*

- Свяжем с общей переменной *x* следующие переменные:

semaphore mutex, first-delay, second-delay;
int first-count, second-count;

- Взаимное исключение доступа к критической секции обеспечивается семафором **mutex**.

- Если процесс не может войти в критическую секцию, т.к. булевское выражение **B** ложно, он ждет на семафоре **first-delay**; затем он “перевешивается” на семафор **second-delay,** до тех пор, пока ему не будет разрешено вновь вычислить *B*.

Реализация

- Число процессов, ждущих на **first-delay** и **second-delay**, хранится, соответственно, в **first-count** и **second-count**.
- Алгоритм предполагает упорядочение типа FIFO процессов в очереди к семафору.
- Для произвольной дисциплины обслуживания очереди требуется более сложный алгоритм.

Мониторы (C. A. R. Hoare)

Высокоуровневая конструкция для синхронизации, которая позволяет синхронизировать доступ к абстрактному типу данных.

```
monitor monitor-name
    {
        описания общих переменных
        procedure body P1 (…) {
                    . . .
            }
        procedure body P2 (…) {
                    . . .
            }
        procedure body Pn (…) {
                     . . .
            }
            {
            код инициализации
            }
            }
```

Мониторы: условные переменные

Для реализации ожидания процесса внутри монитора, вводятся условные переменные:

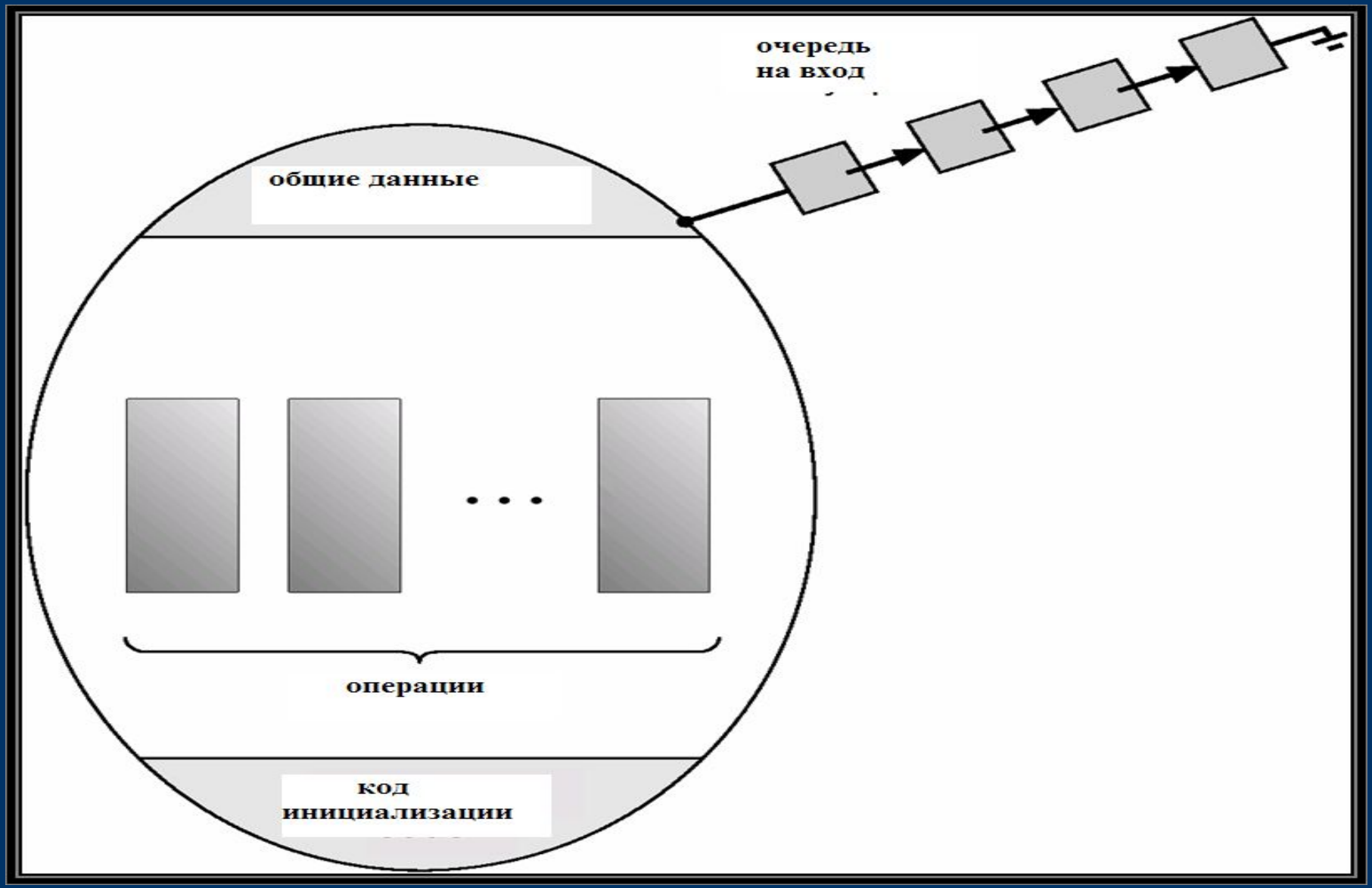
condition x, y;

Условные переменные могут использоваться только в операциях wait и signal.

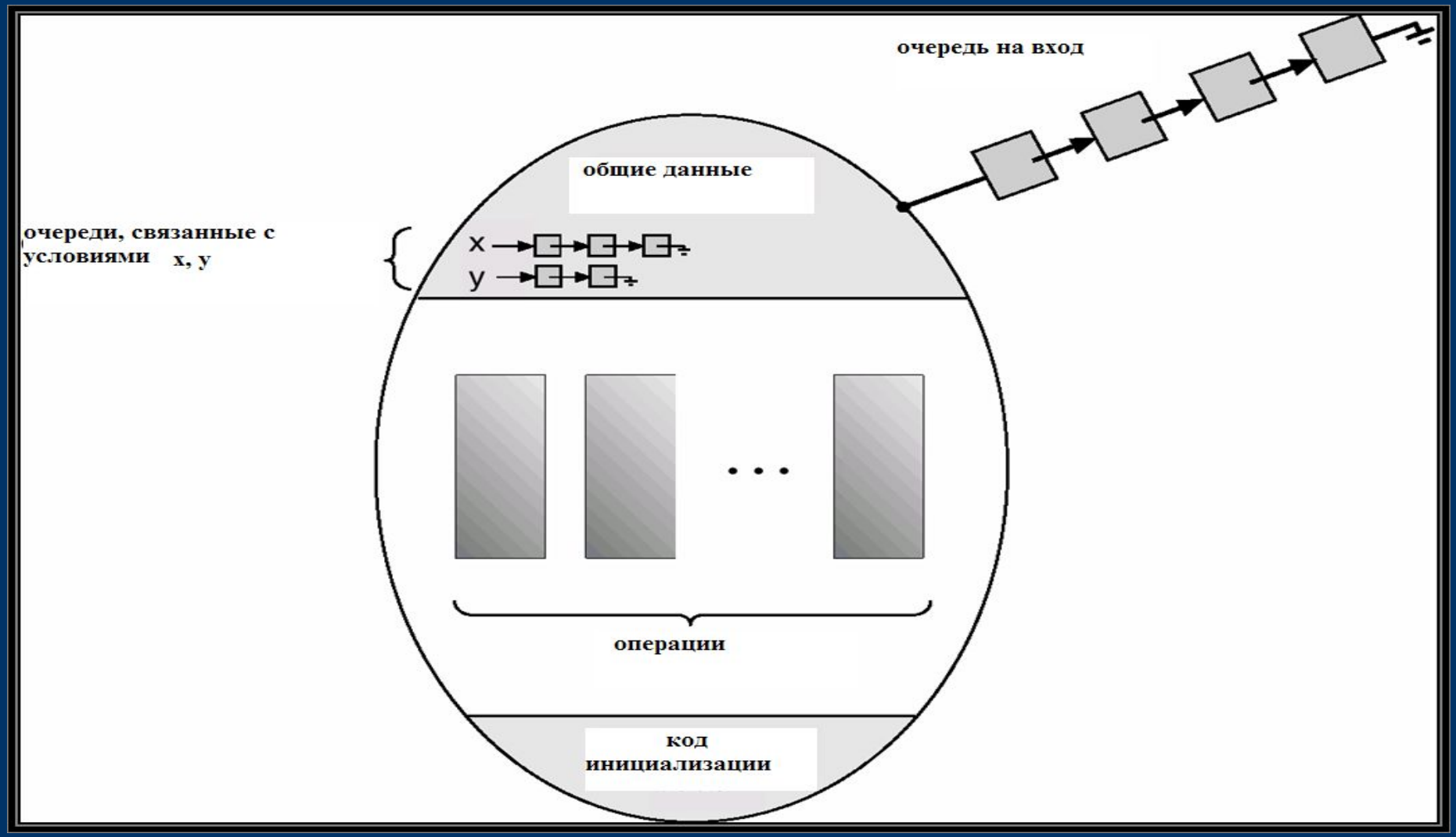
Операция:
x.wait();
означает, что выполнивший ее процесс задерживается до того момента, пока другой процесс не выполнит операцию:
x.signal();

Операция x.signal возобновляет ровно один приостановленный процесс. Если приостановленных процессов нет, эта операция не выполняет никаких действий.

Схематическое представление монитора



Монитор с условными переменными



Пример: обедающие философы

```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)      // following slides
    void putdown(int i)    // following slides
    void test(int i)          // following slides
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```

Обедающие философы: реализация операций pickup и putdown

```
void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
        self[i].wait();
}

void putdown(int i) {
    state[i] = thinking;
    // test left and right neighbors
    test((i+4) % 5);
    test((i+1) % 5);
}
```

Обедающие философы: реализация операции test

```
void test(int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)) {
            state[i] = eating;
            self[i].signal();
void int () {
        for (int i = 0; i < 5; i++)  {
        state [i] = thinking;
        }
}
```

Реализация мониторов с помощью семафоров

Переменные

semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next-count = 0;

Каждая внешняя процедура ***F*** заменяется на:

wait(mutex);
…
тело *F*;
…
if (next-count > 0)
signal(next)
else
signal(mutex);

Обеспечивается взаимное исключение внутри монитора.

Реализация мониторов

- Для каждой условной переменной ***x***:

  semaphore x-sem; // (initially = 0)

  int x-count = 0;

- Реализация операции **x.wait**:

```
x-count++;
if (next-count > 0)
      signal(next);
else
      signal(mutex);
wait(x-sem);
x-count--;
```

Реализация мониторов

- Реализация операции **x.signal**:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```

Реализация мониторов

- Конструкция *conditional-wait*: x.wait(c);
  - c – целое выражение, вычисляемое при исполнении операции wait.
  - значение c (*приоритет*) сокраняется вместе с приостановленным процессом.
  - когда исполняется x.signal, первым возобновляется процесс с меньшим значением приоритета .

Реализация мониторов

Для обеспечения корректности работы системы проверяются два условия:

Пользовательские процессы должны всегда выполнять вызовы операций монитора в правильной последовательности.

Необходимо убедиться, что никакой процесс не игнорирует вход в монитор и не пытается обратиться к ресурсу непосредственно, минуя протокол, предоставляемый монитором .