



# СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

ЛЕКЦИЯ № 7

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.



# РАБОТА С ПОТОКАМИ WINAPI

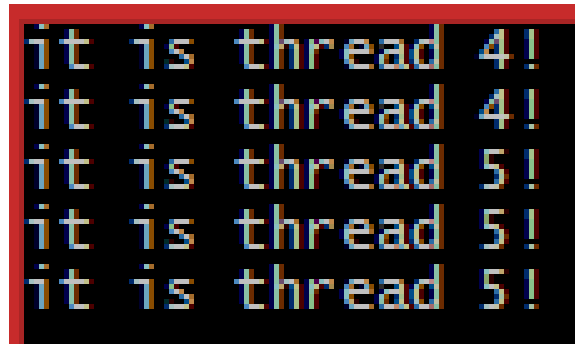
```
#include <processthreadsapi.h>
#define COUNT_OF_THREAD 5

DWORD WINAPI ThreadFunctionThread(LPVOID param)
{
    int* id = (int*)param;
    printf("it is thread %d!\n", *id);
    return 0;
}

int main(int argc, char* argv[])
{
    HANDLE threads[COUNT_OF_THREAD] = { 0 };
    for (int i = 0; i < COUNT_OF_THREAD; i++) {
        threads[i] = CreateThread(NULL, 0, ThreadFunctionThread, &i, 0, 0);
    }

    WaitForMultipleObjects(COUNT_OF_THREAD, threads, TRUE, INFINITE);
    for (int i = 0; i < COUNT_OF_THREAD; i++) {
        CloseHandle(threads[i]);
    }
}
```

Результат работы программы:



```
it is thread 4!
it is thread 4!
it is thread 5!
it is thread 5!
it is thread 5!
```

# ПРОБЛЕМА СИНХРОНИЗАЦИИ

При переходе от последовательных решений к параллельным возникает проблема синхронизации.

Эта проблема может выражаться следующим образом:

- Обеспечение согласованных действий параллельно работающих модулей при выполнении этапов алгоритма
- Целостности используемых данных
- Извещение модулями друг друга о произошедших событиях

# ПРОБЛЕМА СИНХРОНИЗАЦИИ

- Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций – всё определяется взаимными скоростями потоков и моментами скоростями потоков и моментами передачи ЦП от одного потока к другому
- Ситуация, когда два или более потоков, обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называется состязанием или гонками (race conditions)

# КРИТИЧЕСКАЯ СЕКЦИЯ

- Критическая секция (КС, critical section) – это часть программы, результат выполнения которой может непредсказуемо меняться, если в ходе ее выполнения состояние ресурсов, используемых в этой части программы, изменяется другими потоками
- Критическая секция всегда определяется по отношению к определенным критическим ресурсам (например, критическим данным), при несогласованном доступе к которым могут возникнуть нежелательные эффекты

# КРИТИЧЕСКАЯ СЕКЦИЯ

- Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток. Все остальные потоки должны блокироваться на входе в критическую секцию. Подобное требование обычно называется взаимным исключением (mutual exclusion)
- Когда один поток покидает критическую секцию, один из ожидающих потоков может в нее войти

# КРИТИЧЕСКАЯ СЕКЦИЯ WINAPI

```
CRITICAL_SECTION section = { 0 };

/* инициализация критической секции */
InitializeCriticalSection(&section);

/* начало критической секции */
EnterCriticalSection(&section);

/* выход из критической секции */
LeaveCriticalSection(&section);

/* удаление критической секции */
DeleteCriticalSection(&section);
```

# ПРИМЕР

```
#define COUNT_OF_THREAD 5
CRITICAL_SECTION section = { 0 };

int main(int argc, char* argv[])
{
    HANDLE threads[COUNT_OF_THREAD] = { 0 };
    InitializeCriticalSection(&section);
    for (int i = 0; i < COUNT_OF_THREAD; i++) {
        threads[i] = CreateThread(NULL, 0, ThreadFunctionThread, 0, 0, 0);
    }

    WaitForMultipleObjects(COUNT_OF_THREAD, threads, TRUE, INFINITE);
    for (int i = 0; i < COUNT_OF_THREAD; i++) {
        CloseHandle(threads[i]);
    }
    DeleteCriticalSection(&section);
}
```



```

DWORD WINAPI ThreadFunctionThread(LPVOID param)
{
    EnterCriticalSection(&section);
    printf("\nBegin ThreadFunctionThread\n");
    for (int i = 0; i < COUNT_OF_THREAD; i++) {
        if ((rand() % 20) % 2 == 0) {
            Sleep((rand() % 3) * 1000);
        }
    }
    printf("Leave ThreadFunctionThread\n");
    LeaveCriticalSection(&section);
    return 0;
}

```

```

DWORD WINAPI ThreadFunctionThread(LPVOID param)
{
    printf("\nBegin ThreadFunctionThread\n");
    for (int i = 0; i < COUNT_OF_THREAD; i++) {
        if ((rand() % 20) % 2 == 0) {
            Sleep((rand() % 3) * 1000);
        }
    }
    printf("Leave ThreadFunctionThread\n");
    return 0;
}

```

```

Begin ThreadFunctionThread
Leave ThreadFunctionThread

```

```

Begin ThreadFunctionThread
Leave ThreadFunctionThread

```

```

Begin ThreadFunctionThread
Leave ThreadFunctionThread

```

```

Begin ThreadFunctionThread
Leave ThreadFunctionThread

```

```

Begin ThreadFunctionThread
Leave ThreadFunctionThread

```

```

Begin ThreadFunctionThread

```

```

Begin ThreadFunctionThread

```

```

Begin ThreadFunctionThread

```

```

Begin ThreadFunctionThread

```

```

Begin ThreadFunctionThread

```

```

Leave ThreadFunctionThread

```

```

Leave ThreadFunctionThread

```

```

Leave ThreadFunctionThread

```

```

Leave ThreadFunctionThread

```

```

Leave ThreadFunctionThread

```

# ЗАДАЧА ВЗАИМНОГО ИСКЛЮЧЕНИЯ

Необходимо согласовать работу  $n > 1$  параллельных потоков при использовании некоторого критического ресурса таким образом, чтобы удовлетворить следующим требованиям:

- одновременно внутри критической секции должно находиться не более одного
- критические секции не должны иметь приоритета в отношении друг друга
- остановка какого-либо потока вне его критической секции не должна влиять на дальнейшую работу потоков по использованию критического ресурса
- решение о вхождении потоков в их критические секции при одинаковом времени поступления запросов на такое вхождение и равноприоритетности потоков не откладывается на неопределенный срок, а является конечным во времени
- относительные скорости развития потоков неизвестны и произвольны
- любой поток может переходить в любое состояние отличное от активного, вне пределов своей критической секции
- освобождение критического ресурса и выход из критической секции должны быть произведены потоком, использующим критический ресурс за конечное время

# АППАРАТНОЕ РЕШЕНИЕ ЗАДАЧИ

- Запрещение прерываний
- Использование разделяемых переменных:
  - Алгоритм Деккера
  - Алгоритм Петерсона
  - Алгоритм булочной
- Использование специальных команд центрального процессора:
  - test-and-set
  - swap
  - read-conditional-write

# ЗАПРЕЩЕНИЙ ПРЕРЫВАНИЙ

Прерывание – это сигнал процессору, указывающий на событие, которое требует немедленного внимания.

- Прикладные программы, как правило, не могут запрещать прерывания
- На многопроцессорных системах прерывания запрещаются только на текущем процессоре, соответственно, несколько центральных процессоров одновременно могут выполнять критическую секцию
- При запрещении прерываний на длительное время могут возникнуть сложности с функционированием устройств, не получавших должного внимания

```
while (true) {  
    DisableInterrupts();  
    CS(); /* Начало критической секции */  
    EnableInterrupts();  
    NCS(); /* Выход из критической секции */  
}
```

# ИСПОЛЬЗОВАНИЕ РАЗДЕЛЯЕМЫХ ПЕРЕМЕННЫХ.

Можно ли с помощью разделяемой переменной `flag` реализовать код, который будет выполняться в критической секции?

Под `flag` – мы понимаем намерения потока войти в критическую секцию.

```
bool flag = false;
while (true) {
    while (flag) { };
    flag = true;
    CS(); /* Начало критической секции */
    flag = false;
    NCS(); /* Выход из критической секции */
}
```

# АЛГОРИТМ ДЕККЕРА

```
/* Номер потока: 1 или 2. Приведен код потока 1. */
bool flag1 = false; flag2 = false; turn = 1;
/* flagi-намерение i-го потока войти в КС */
/* turn-номер потока, имеющего право войти первым */
while (true) {
    flag1 = true;
l1: if (flag2) {
        if (turn == 1) goto l1;
        else {
            flag1 = false;
            while (turn == 2) {}
        }
    } else {
        CSi();
        turn = 2;
        flag1 = false;
    }
}
```

- Первое известное корректное решение проблемы взаимного исключения.
- Если два процесса попытаются зайти в критическую секцию одновременно, то получится это сделать только у одного процесса
- Алгоритм отдаёт приоритет тому, чья очередь наступила

# АЛГОРИТМ ПЕТЕРСЕНА

```
/* i -номер потока (0 или 1) */
int ready[2] = { 0, 0 }; /* Намерение войти в КС */
int turn= 0 ;           /* Приоритетный поток */
while( true ) {
    ready[i] = 1;
    turn = 1 - i;
    while (ready[1-i] && turn == 1-i) { };
    CSi();
    ready[i]=0;
    NCSi();
}
```

- Более лаконичная реализация способа, предложенного Деккером

# АЛГОРИТМ БУЛОЧНОЙ

В алгоритме булочной (bakery algorithm) организуется очередь из потоков, желающих войти в критическую секцию. Для этого используются следующие разделяемые переменные:

- `int number[n] = { 0, 0, ...}` – *i*-ый элемент массива хранит позицию *i*-ого потока в очереди в критическую секцию (0 означает, что поток не пытается попасть в КС) если два потока имеют одинаковую позицию в очереди, первым в критическую секцию попадает поток, имеющий меньший номер
- `bool choosing[n] = { false, false, ...}` – *i*-ый элемент массива равен true во время вычисления *i*-ым потоком своей позиции в очереди



# АЛГОРИТМ БУЛОЧНОЙ

```
bool choosing[n] = { false };
int number[n] = { 0 };
while (true) {
    choosing[i] = true;
    number[i] = max(number[0], ..., number[n - 1]) + 1;
    choosing[i] = false;
    for (int j = 0; j < n; j++) {
        while (choosing[j]) {};
        while (number[j] != 0 && (number[j], j) < (number[i], i)) {};
        CSi();
        number[i] = 0;
        NCSi();
    }
}
```

# ИСПОЛЬЗОВАНИЕ СПЕЦИАЛЬНЫХ КОМАНД ЦЕНТРАЛЬНОГО ПРОЦЕССОРА

Проверить и присвоить:

```
int Test_and_Set(int *target)
{
    int tmp = *target;
    *target = 1;
    return tmp;
}
```

Решение задачи взаимного исключения:

```
int lock = 0; /* Признак блокировки критических данных */
while (true) {
    while (Test_and_Set(&lock)) {};
    CSi();
    lock = 0;
    NCSi();
}
```

# ИСПОЛЬЗОВАНИЕ СПЕЦИАЛЬНЫХ КОМАНД ЦЕНТРАЛЬНОГО ПРОЦЕССОРА

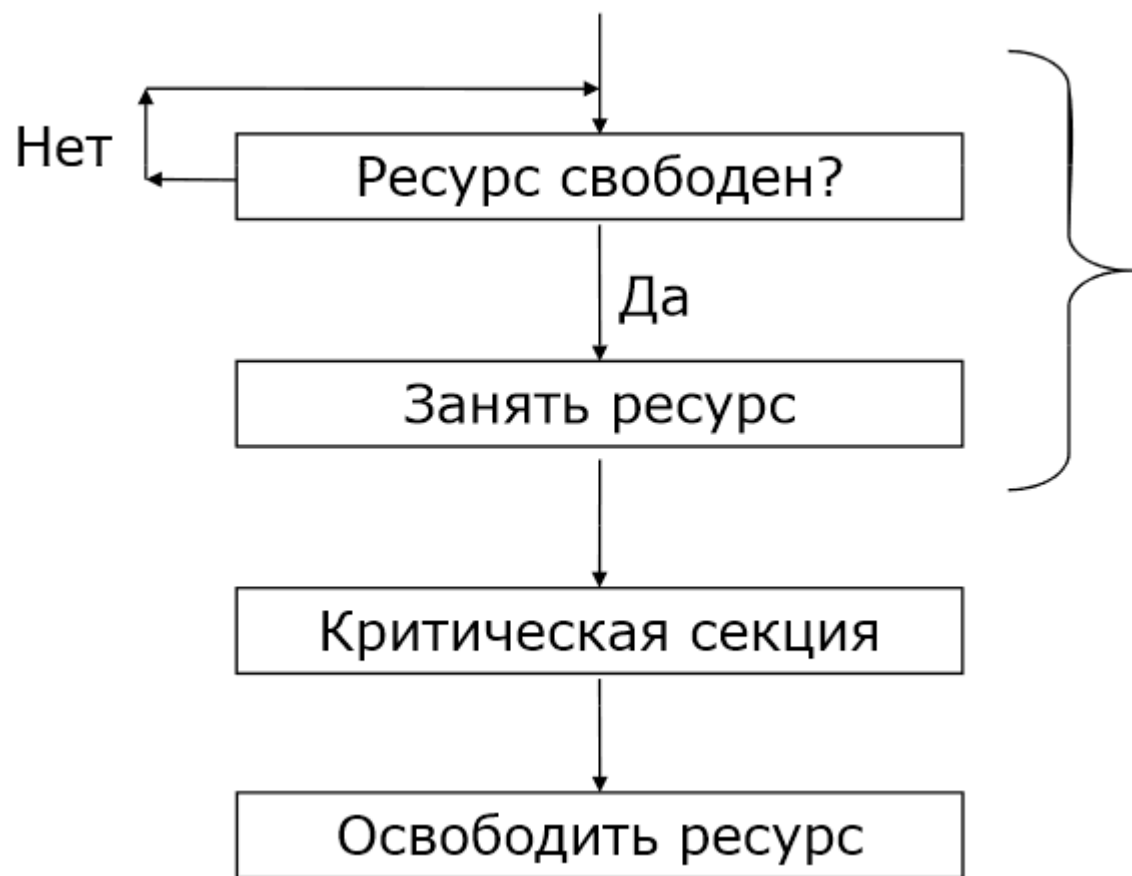
Обменять значения:

```
void Swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Решение задачи взаимного исключения:

```
int lock = 0; /* Признак блокировки критических данных */
int key;
while (true) {
    key = 1;
    do {
        Swap(&lock, &key);
    } while (key);
    CSi();
    lock = 0;
    NCSi();
}
```

# АКТИВНОЕ ОЖИДАНИЕ



- При работе с признаком занятости(блокировки) ресурса, поток использует "активное ожидания", то есть крутится в цикле
- Занять ресурс то есть крутится в цикле, ожидая освобождения признака блокировки

# СЕМАФОР

- Семафоры – примитивы синхронизации более высокого уровня абстракции, чем признаки блокировки, предложены Дийкстрой (Dijkstra) в 1968 г в качестве компонента операционной системы.
- Семафор – это переменная, для которой определены 2 операции: P и V
  - $P(sem)$  (wait/down) – ожидает выполнения  $sem > 0$ , затем уменьшает  $sem$  на 1 и возвращает управление
  - $V(sem)$  (signal/up) – увеличивает  $sem$  на 1 ( )
- Операции P и V выполняются атомарно

# РЕАЛИЗАЦИЯ ДИЙКСТРЫ

С каждым семафором ассоциирована очередь потоков

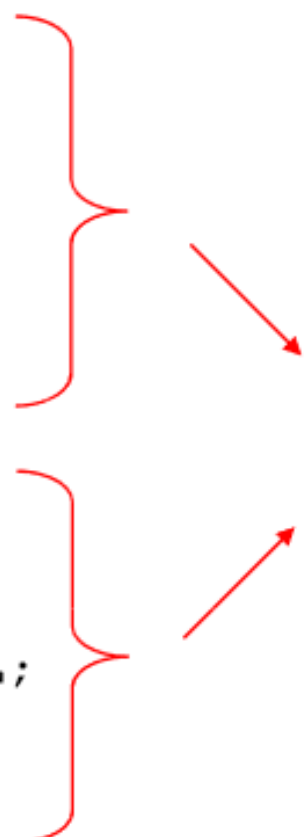
- при вызове потоком  $P(sem)$ :
  - если семафор "свободен" ( $>0$ ), его значение уменьшается на 1, и выполнение потока продолжается
  - если семафор "занят" ( $\leq 0$ ), поток переводится в состояние ожидания и помещается в очередь, соответствующую данному семафору
  - запускается какой-либо другой готовый к выполнению поток
- при вызове потоком  $V(sem)$ :
  - если очередь потоков, ассоциированная с данным семафором, не пуста – один из них разблокируется и помещается в очередь готовых к выполнению
  - поток, вызвавший  $V(sem)$ , продолжает свое выполнение
  - в противном случае (нет потоков, ожидающих освобождения семафора), значение семафора увеличивается

Таким образом, все вызовы изменяют состояние семафора, то есть семафор сохраняет некоторую информацию о прошедших вызовах

```
typedef struct{
    int value;
    list<thread> L;
} semaphore;
```

```
P(S) {
    S.value = S.value - 1;
    if( S.value < 0 ){
        add this thread to S.L;
        block;
    }
```

```
V(S) {
    S.value = S.value + 1;
    if( S.value <= 0 ){
        remove a thread T from S.L;
        wakeup T;
    }
```



P () и V() – критические  
секции, должны  
выполняться атомарно

# ВИДЫ СЕМАФОРОВ

## Двоичный семафор

- `S.value` может принимать значения 0 и 1, инициализируется значением 1
- обеспечивает эксклюзивный доступ к ресурсу (например, при работе в критической секции)
- одновременно может выполняться только один поток

## Счетный семафор

- `S.value` инициализируется значением  $N$  = число доступных единиц ресурса
- представляет ресурсы, состоящие из нескольких однородных элементов
- позволяет потокам исполняться пока есть неиспользуемые элементы



# МЬЮТЕКС

Мьютекс – двоичный семафор, обычно используемый для организации согласованного доступа к неделимому общему ресурсу.

Мьютекс может принимать значения 1 (свободен) и 0 (занят).

Операции над мьютексами:

- `acquire(mutex)` – уменьшить (занять) мьютекс
- `release(mutex)` – увеличить (освободить) мьютекс
- `tryacquire(mutex)` – часто реализуемая не блокирующая операция, выполняющая попытку уменьшить (занять) мьютекс

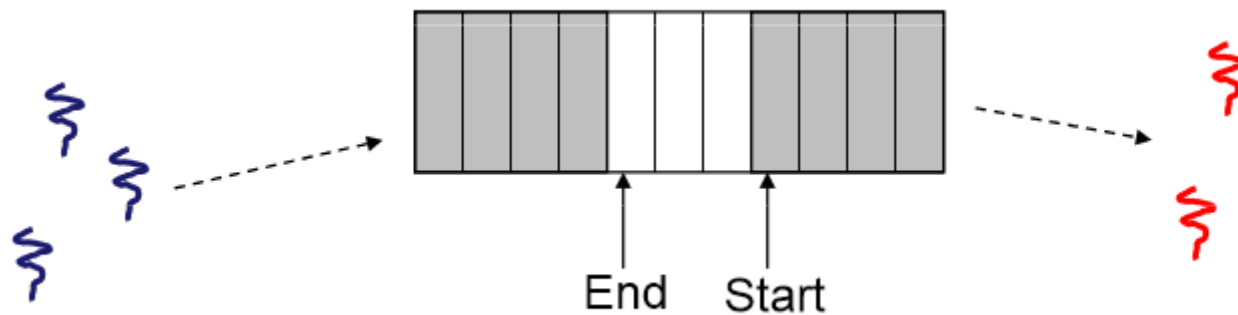
# МЬЮТЕКС

Мьютексы в конкретных реализациях могут иметь дополнительные свойства:

- Запоминание владельца – освободить мьютекс может только поток, захвативший его
- Рекурсивность – поток может многократно захватить мьютекс (вызывать `acquire()`); для освобождения мьютекса поток должен соответствующее число раз вызвать `release()`
- Наследование приоритета – поток, захвативший мьютекс временно наследует максимальный из приоритет потоков, ждущих освобождения данного мьютекса

# ЗАДАЧА «ПРОИЗВОДИТЕЛЬ-ПОТРЕБИТЕЛЬ»

- Имеется циклический буфер из  $N$  элементов
- Потоки-производители добавляют в него записи (по одной за одну операцию)
- Потоки-потребители извлекают записи (также по одной за раз)
- Потоки выполняются параллельно



```
Semaphore mutex = 1; // семафор, управляющий доступом к разделяемым данным  
Semaphore empty = n; // количество пустых записей (после запуска все записи пусты)  
Semaphore full = 0; // количество заполненных записей  
                        // (после запуска нет заполненных записей)
```

Producer:

```
P(empty); // ждем появления свободной записи  
P(mutex); // получаем доступ к указателям (критические данные)  
    <записываем значение в запись>  
V(mutex); // завершили работу с указателями  
V(full);  // сигнализируем о появлении заполненной записи
```

Consumer:

```
P(full); // ждем появления заполненной записи  
P(mutex); // получаем доступ к указателям (критические данные)  
    <извлекаем данные из записи>  
V(mutex); // завершили работу с указателями  
V(empty); // сигнализируем о появлении свободной записи  
    <используем данные из записи>
```

# ЗАДАЧА «ЧИТАТЕЛИ-ПИСАТЕЛИ»

- Имеются критические данные над которыми определены операции чтения и записи
- Потоки-писатели изменяют данные; если поток-писатель работает с данными, они не могут использоваться другими потокам
- Потоки-читатели не изменяют данные; если поток-читатель работает с данными, они могут быть использованы другими потоками-читателями
- Потоки выполняются параллельно



```
Semaphore RC      = 1; // управляет доступом к переменной ReadCount
Semaphore Access  = 1; // управляет допуском к данным писателя или 1-го читателя
int ReadCount     = 0; // количество "активных" (читающих) читателей
```

Writer:

```
P(Access);          // захватываем доступ к критическим данным
    <выполняем операцию записи>
V(Access);          // освобождаем доступ к критическим данным
```

Reader:

```
P(RC);              // получаем эксклюзивный доступ к переменной ReadCount
ReadCount++;        // увеличиваем число активных читателей
if( ReadCount == 1 )
    P(Access);      // если мы первые - захватываем доступ к критическим данным
V(RC);              // освобождаем доступ к переменной ReadCount
    <выполняем операцию чтения>
P(RC);              // получаем эксклюзивный доступ к переменной ReadCount
ReadCount--;        // уменьшаем число активных читателей
if( ReadCount == 0 )
    V(Access);      // если мы последние - освобождаем доступ к критическим данным
V(RC);              // освобождаем доступ к переменной ReadCount
```

## ЗАДАЧА «ЧИТАТЕЛИ-ПИСАТЕЛИ»

Замечания:

- Первый появившийся читатель переходит в состояние ожидания при выполнении  $P(\text{Access})$  в присутствии писателя; все остальные читатели переходят в состояние ожидания при выполнении вызова  $P(RC)$
- Если при завершении работы писателем или читателем уже присутствуют ожидающие писатели и читатели выбор потока, захватывающего семафор, определяется реализацией семафора
- Если потоки-читатели постоянно входят в критическую секцию, возможна задержка ("голодание", starvation) потоков-писателей

# МЬЮТЕКСЫ WINAPI

Атрибуты мьютекса в Win32:

- Идентификатор потока – определяет, какой поток захватил мьютекс
- Счетчик рекурсии – сколько раз была выполнена операция захвата

*/\* Создание Мьютекса \*/*

```
HANDLE CreateMutexW(LPSECURITY_ATTRIBUTES lpMutexAttributes,  
                    BOOL bInitialOwner, LPCWSTR lpName);
```

*/\* Открытие Мьютекса \*/*

```
HANDLE OpenMutexW(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCWSTR lpName);
```

*/\* Удаление Мьютекса \*/*

```
BOOL CloseHandle(HANDLE hObject);
```



# МЬЮТЕКСЫ WINAPI

*/\* Захват Мьютекса \*/*

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);  
DWORD WaitForMultipleObjects(DWORD nCount, HANDLE* lpHandles,  
                             BOOL bWaitAll, DWORD dwMilliseconds);
```

*/\* Освобождение Мьютекса \*/*

```
BOOL ReleaseMutex(HANDLE hMutex);
```

# СЕМАФОРЫ WINAPI

Атрибуты семафора в Win32:

- Максимальное число ресурсов (контролируемых семафором)
- Текущее число ресурсов

*/\* Создание Семафора \*/*

```
HANDLE CreateSemaphoreW(LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
                        LONG lInitialCount, LONG lMaximumCount, LPCWSTR lpName);
```

*/\* Открытие Семафора \*/*

```
HANDLE OpenSemaphoreW(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCWSTR lpName);
```

*/\* Удаление Семафора \*/*

```
BOOL CloseHandle(HANDLE hObject);
```

# СЕМАФОРЫ WINAPI

*/\* Захват Семафора \*/*

**DWORD** WaitForSingleObject(**HANDLE** hHandle, **DWORD** dwMilliseconds);

**DWORD** WaitForMultipleObjects(**DWORD** nCount, **HANDLE\*** lpHandles,  
**BOOL** bWaitAll, **DWORD** dwMilliseconds);

*/\* Освобождение Семафора \*/*

**BOOL** ReleaseSemaphore(**HANDLE** hSemaphore, **LONG** lReleaseCount, **LPLONG** lpPreviousCount);

# СЕМАФОРЫ

## РЕЗЮМЕ

- С помощью семафоров можно решить любую классическую задачу синхронизации, но по сути, семафоры – это разделяемые глобальные переменные (что является признаком плохой структуры программы)
- Семафоры используются как для решения задачи взаимного исключения, так и для координации действий
- Отсутствует связь между семафором и данными, доступом к которым он управляет данными
- Нет контроля использования семафоров со стороны компилятора или операционной системы соответственно нет гарантий что системы, соответственно – нет гарантий, что семафоры будут использованы правильно

# ПОНЯТИЕ ВЗАИМОБЛОКИРОВКИ

- Операционная система управляет многими видами ресурсов
- Если процессам предоставляются исключительные права доступа к ресурсам, то процесс, получивший в свое распоряжение один ресурс и затребовавший другой, может ожидать предоставления второго ресурса в течение неопределенного времени

# ВЗАИМОБЛОКИРОВКА (ТУПИК)

Существуют неразделяемые ресурсы

- одиночные или счетные (то есть имеющиеся более чем в одном экземпляре)
- аппаратные или программные: принтеры, ленточные накопители, центральный процессор, файлы и записи в них, семафоры и т.д.

Работа процесса с ресурсами включает три стадии

- Получить/захватить ресурс
  - если ресурс свободен, он предоставляется процессу
  - если ресурс занят, процесс блокируется
- Использовать ресурс
- Освободить ресурс

Возможна следующая нежелательная ситуация

- Процесс А захватил ресурс 1 и ожидает предоставления ему ресурса 2
- Процесс В захватил ресурс 2 и ожидает предоставления ему ресурса 1

Такая ситуация называется взаимоблокировкой или тупиком (deadlock)

```
Semaphore  Sem1 = 1; /* управляет доступом к ресурсу 1 */  
Semaphore  Sem2 = 1; /* управляет доступом к ресурсу 2 */
```

функция процесса А:

```
{  
    /* инициализация */  
  
1    P (Sem1) ;  
4    P (Sem2) ;  
  
    /* использование  
       обоих ресурсов */  
  
    V (Sem2) ;  
    V (Sem1) ;  
}
```

функция процесса В:

```
{  
    /* инициализация */  
  
2    P (Sem2) ;  
3    P (Sem1) ;  
  
    /* использование  
       обоих ресурсов */  
  
    V (Sem2) ;  
    V (Sem1) ;  
}
```

После шага 4 мы получили взаимоблокировку

# ОПРЕДЕЛЕНИЕ ТУПИКА

Множество процессов находится в тупиковой ситуации:

- если каждый процесс ожидает некоторого события
- это событие может быть вызвано только действиями другого процесса из данного множества

В большинстве случаев ожидаемое событие – освобождение некоторого ресурса.

В дальнейшем рассмотрении мы будем предполагать именно такую ситуацию.



# НЕОБХОДИМЫЕ УСЛОВИЯ ТУПИКА

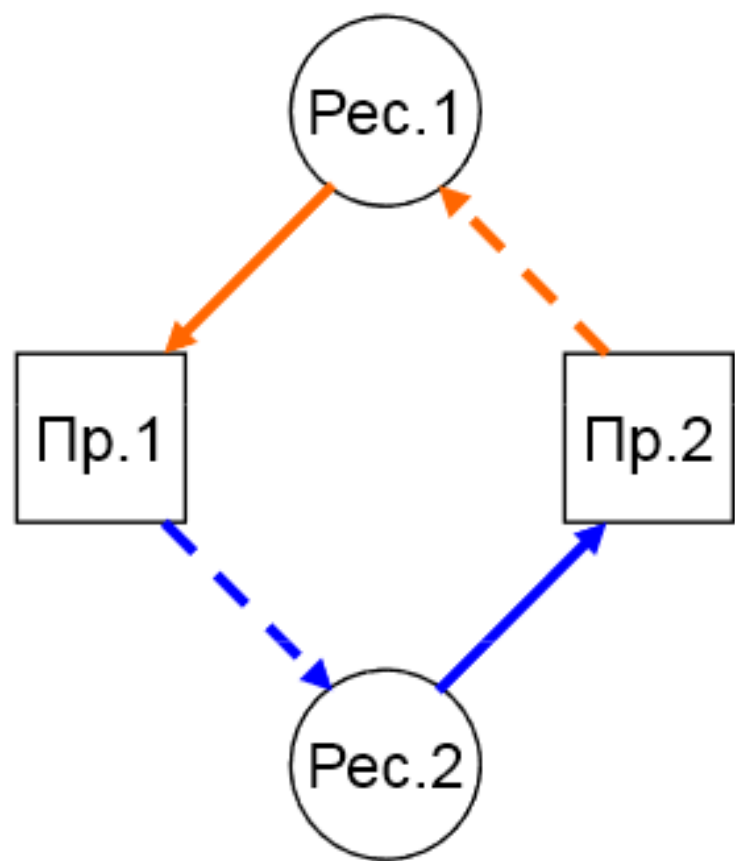
Для возникновения ситуации взаимоблокировки должны выполняться следующие условия:

- Mutual Exclusion (Взаимное исключение) – по крайней мере один из запрашиваемых ресурсов является неделимым (то есть должен захватываться в эксклюзивное использование)
- Hold and wait (Удержание ресурсов при ожидании) – существует процесс, владеющий некоторым ресурсом и ожидающий освобождения другого ресурса
- No preemption (Неперераспределяемость ресурсов) – ресурсы не могут быть отобраны у процесса без его желания
- Circular wait (Циклическое ожидание) – существует такое множество процессов  $\{P_1, P_2, \dots, P_N\}$ , в котором  $P_1$  ждет  $P_2$ ,  $P_2$  ждет  $P_3, \dots$ ,  $P_N$  ждет  $P_1$

# ГРАФ ПРОЦЕСС-РЕСУРС

Направленный граф "процесс-ресурс" включает:

- Множество вершин  $V = P \cup R$ , где  $P = \{P_1, P_2, \dots, P_N\}$  – множество процессов,  $R = \{R_1, R_2, \dots, R_M\}$  – множество ресурсов
- Дуги запросов – направлены от процессов к ресурсам дуга  $P_i \rightarrow R_j$  означает что процесс  $P_i$  запросил ресурс  $R_j$
- Дуги распределения – направлены от ресурсов к процессам дуга  $R_i \rightarrow P_j$  означает, что ресурс  $R_j$  выделен процессу  $P_i$
- Если граф "процесс-ресурс" не имеет циклов, тупиков нет тупиков
- Если граф "процесс-ресурс" имеет цикл, возможно, тупик существует



Ресурс 1 выделен  
процессу 1



Процесс 2 запросил  
ресурс 1



Ресурс 2 выделен  
процессу 2



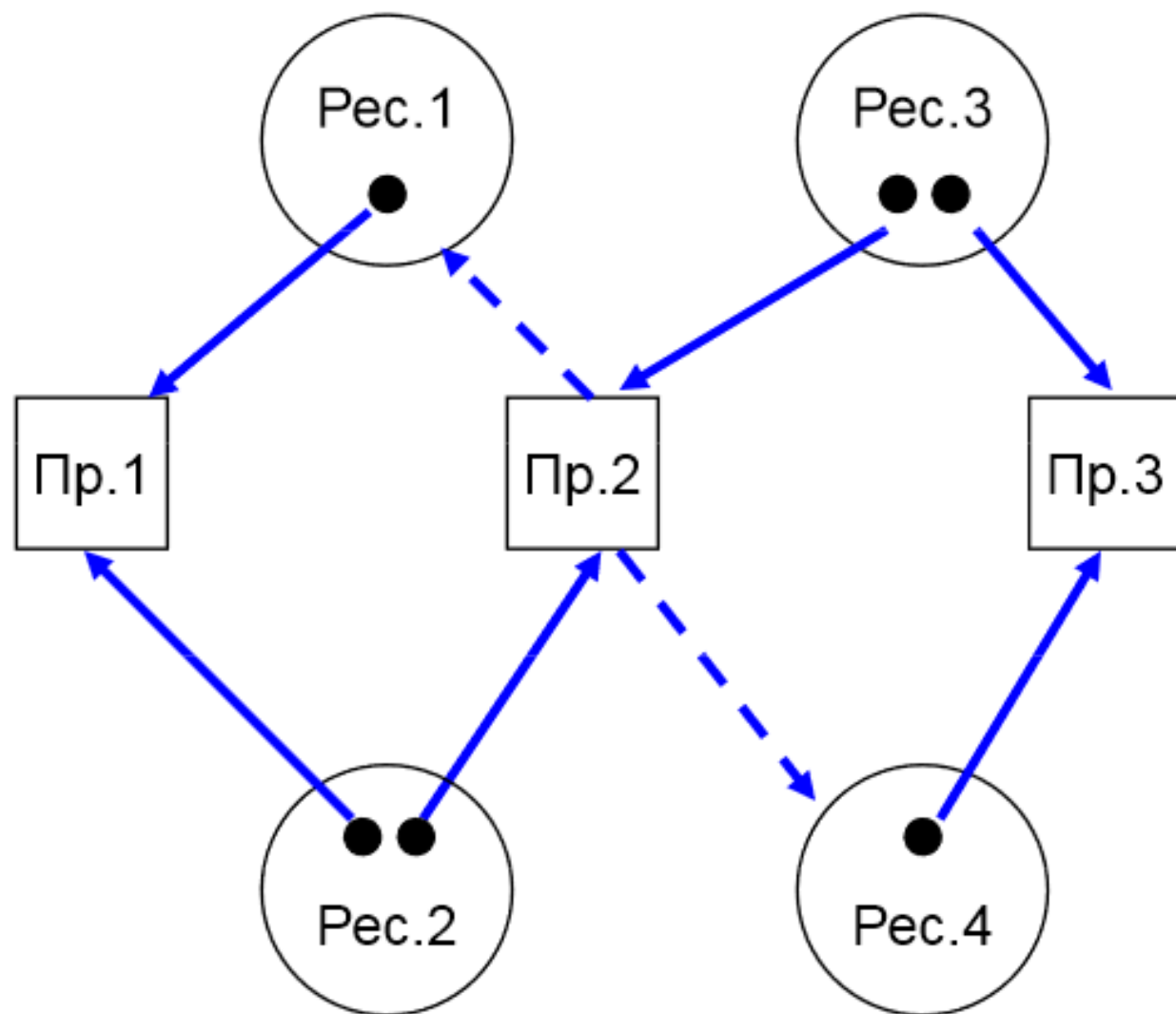
Процесс 1 запросил  
ресурс 2

# РЕДУКЦИЯ (УМЕНЬШЕНИЕ) ГРАФА

- Граф "процесс-ресурс" может быть сокращен за счет процесса, все запросы которого могут быть удовлетворены
- При сокращении все ресурсы выбранного процесса освобождаются – все дуги графа, обозначающие выделения ресурсов этому процессу, удаляются

Доказан ряд утверждений связанных с процессом редукции

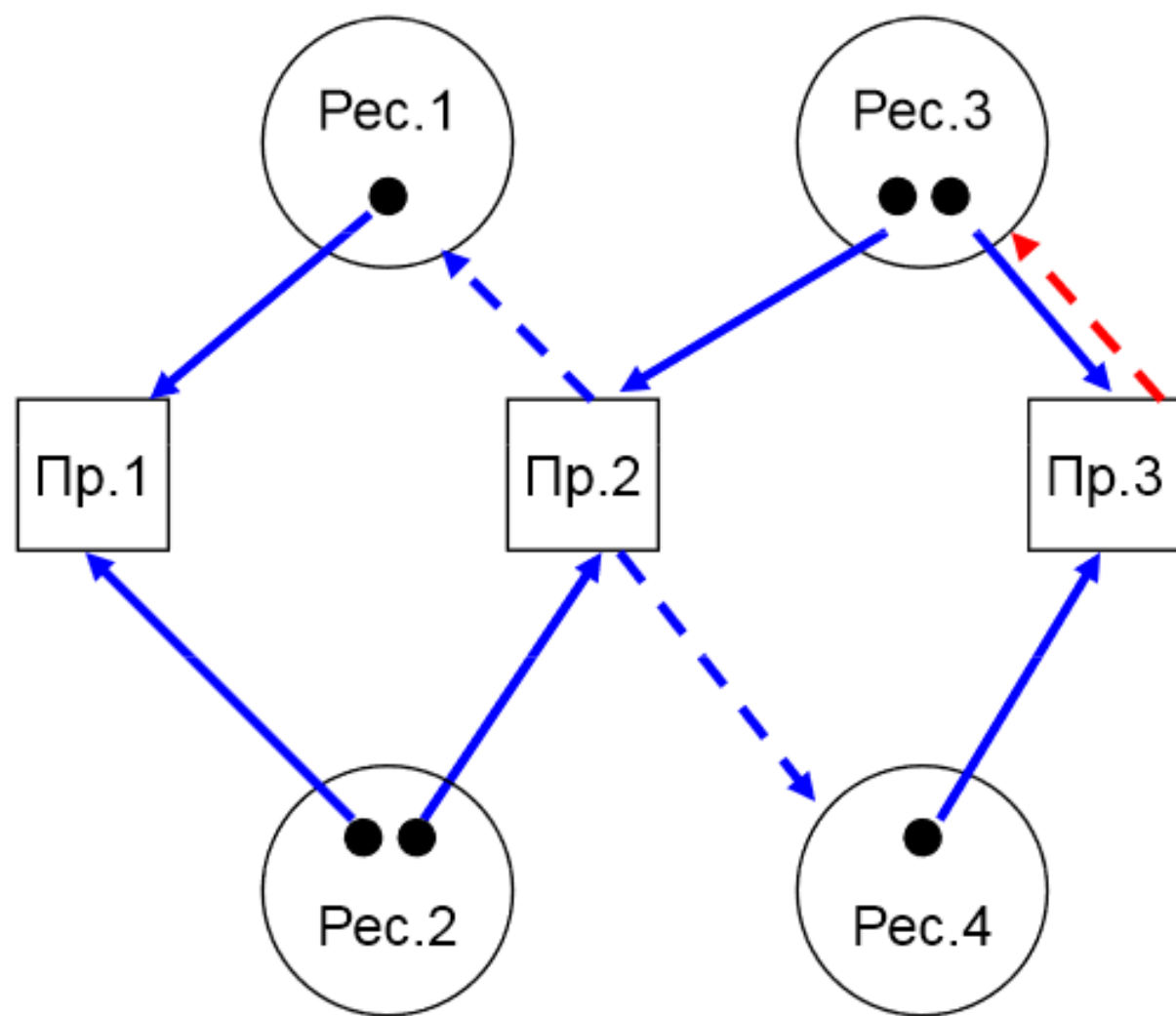
- Если граф полностью редуцируем, взаимоблокировка отсутствует
- Порядок выполнения редукции не имеет значения



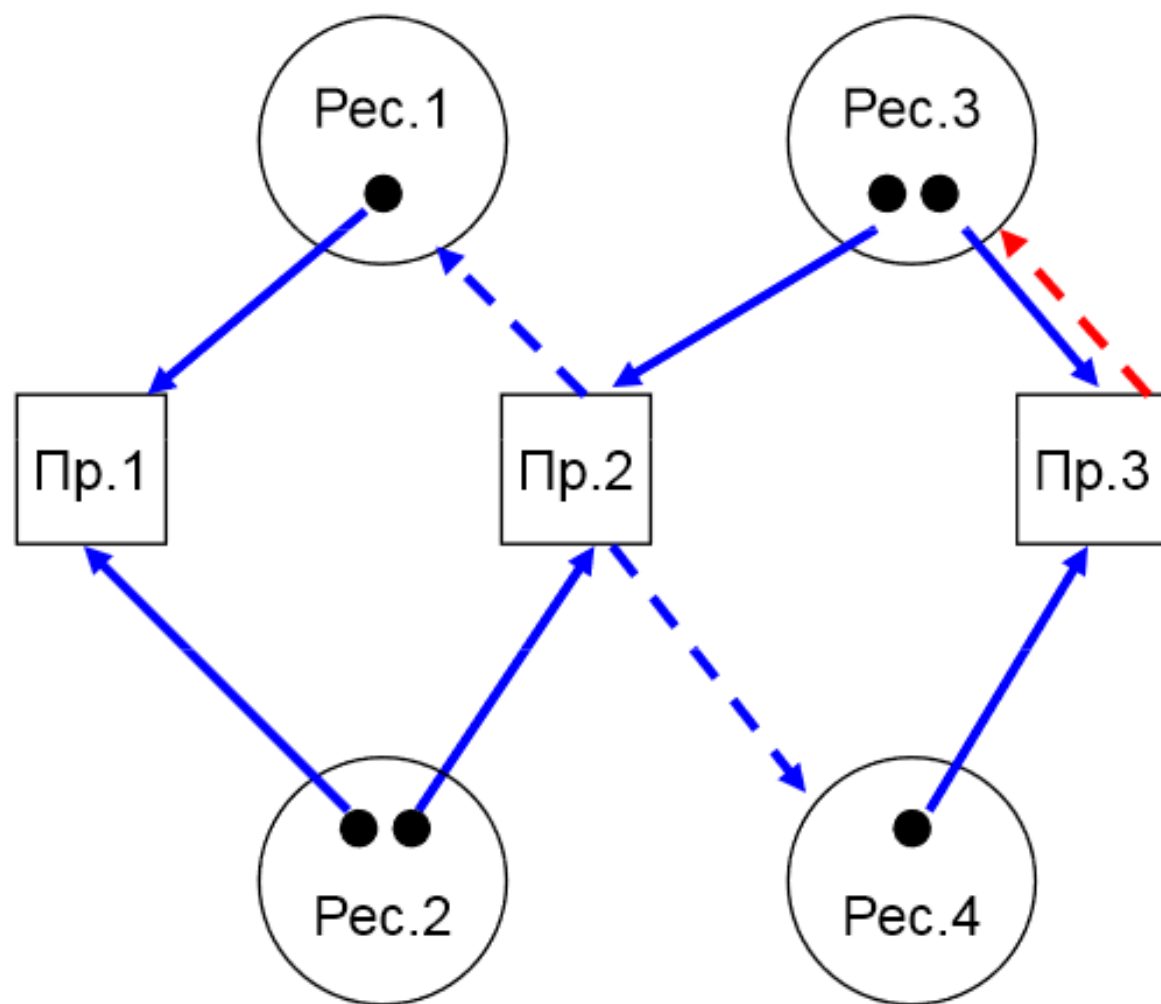
→ Ресурс  
выделен  
процессу

- - -> Процесс  
запросил  
ресурс

Какие запросы  
могут привести к  
возникновению  
взаимоблокировки?



Есть  
взаимоблокировка



Есть  
взаимоблокировка

# РЕШЕНИЕ ПРОБЛЕМЫ ТУПИКА

- Игнорирование проблемы ("Алгоритм страуса") считаем что – считаем, что тупики никогда не возникают, либо возникают достаточно редко и не приносят значительного ущерба
  - “За производительность и удобство работы пользователей стоит заплатить такую цену, как нечастые сбои в работе”
- Предупреждение тупиков
  - Предотвращение тупиков (prevention) – достигается устранением одного из 4 условий существования тупика (не допускает возникновения тупиков)
  - Избегание тупиков (avoidance) – аккуратное распределение ресурсов на основании имеющейся информации об их планируемом использовании позволяет избежать возникновения тупиков
- Устранение тупиков – позволяем тупику возникнуть, затем обнаруживаем и устраняем его. Включает две стадии: обнаружение тупика и восстановление после возникновения тупика (detection & recovery)



# ПРЕДОТВРАЩЕНИЕ ТУПИКОВ. УСТРАНЕНИЕ ВЗАИМОИСКЛЮЧЕНИЯ.

- Устранения условия "Mutual exclusion" (взаимное исключение) можно достигнуть построив над ресурсом абстракцию, позволяющую использовать ресурс нескольким процессам ресурс нескольким процессам одновременно
- Пример: абстракция "очередь печати", построенная над ресурсом "принтер"
- К сожалению, это далеко не всегда возможно и требует дополнительных ресурсов

# ПРЕДОТВРАЩЕНИЕ ТУПИКОВ. УСТРАНЕНИЕ НЕПЕРЕРАСПРЕДЕЛЯЕМОСТИ

- Для устранения условия "No preemption" (Неперераспределяемость ресурсов) требуется обеспечить выполнение следующих операций:
  - запоминание контекста работы процесса с ресурсом
  - передача ресурса другому процессу
  - возврат ресурса процессу и восстановление контекста работы процесса с ресурсом
- Для каких-то ресурсов это возможно (например, для ЦП), для других – нет (например, для принтера)
- Момент передачи ресурсов (2 подхода):
  - Если процесс затребовал ресурсы, часть из которых недоступна, перераспределяем принадлежащие ему ресурсы
  - Перераспределяем ресурсы процессов в состоянии ожидания для удовлетворения поступившего запроса от процесса в состоянии выполнения

# ПРЕДОТВРАЩЕНИЕ ТУПИКОВ. УСТРАНЕНИЕ УСЛОВИЯ HOLD&WAIT

Исключение данного условия – задача прикладных программистов. Существует несколько подходов:

- Можно запрашивать все необходимые ресурсы до начала выполнения
  - Но обычно процессы не знают, какие ресурсы им понадобятся
  - Возможно голодание при ожидании множества популярных ресурсов
  - Ресурсы используются неэффективно (возможно, часть ресурсов нужна на очень короткое время)
- При возникновении потребности в дополнительных ресурсах – освобождаем все уже имеющиеся, затем запрашиваем те, что необходимы в настоящий момент
  - Также возможно голодание и имеет место неэффективное использование ресурсов

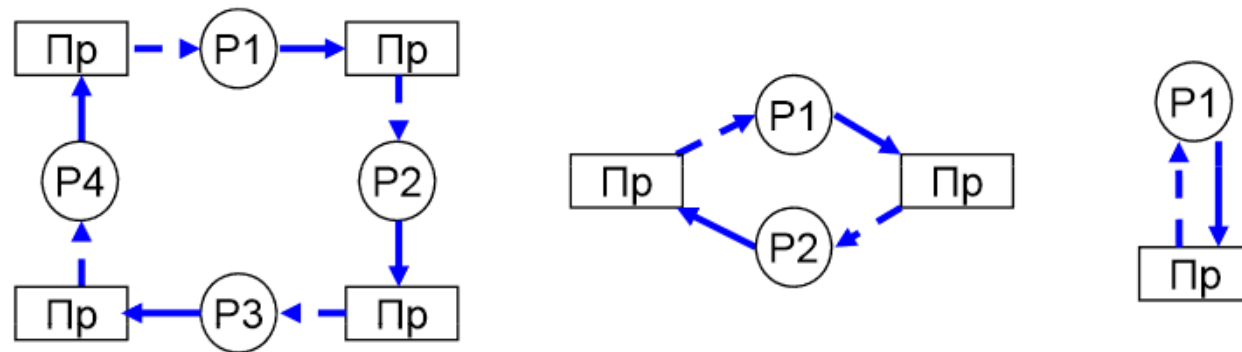
Как правило, в ОС имеются вызовы, позволяющие прикладной программе реализовать алгоритм, не оставляющий за собой владение уже имеющимися ресурсами при запросе дополнительных

# ПРЕДОТВРАЩЕНИЕ ТУПИКОВ. УСТРАНЕНИЕ УСЛОВИЯ CIRCULAR WAIT...

- Можно позволить процессам владеть только одним ресурсом в каждый момент времени
- Можно ввести для ресурсов нумерацию и обязать процессы запрашивать ресурсы строго в порядке возрастания их номеров

# ПРЕДОТВРАЩЕНИЕ ТУПИКОВ. УСТРАНЕНИЕ УСЛОВИЯ CIRCULAR WAIT...

- Ресурсам присваиваются номера ( $P_1, P_2, \dots$ )
- Позволяется запрашивать ресурсы строго в порядке возрастания (или убывания) их номеров
- Идея: в цикле всегда есть процесс, у которого номер имеющегося ресурса больше номера запрошенного (исключение составляет случай, когда процесс запрашивает еще одну единицу уже имеющегося у него ресурса)



Недостатки подхода:

- Нумерация не всегда возможна
- Неэффективное использование ресурсов

# ИЗБЕГАНИЕ ТУПИКОВ. АЛГОРИТМ БАНКИРА.

- Если у нас имеется информация о будущем, можем ли мы гарантировать, что тупик не возникнет? Считаем, что максимальные требования всех процессов к ресурсам известны до начала выполнения процессов

Стратегия избегания тупиков, Алгоритм Банкира:

- Перед выделением ресурса проверяем, является ли состояние, в которое мы перейдем после выделения, безопасным
- Если новое состояние безопасно – выделяем ресурс
- Если новое состояние небезопасно – ресурс не выделяем, блокируем процесс, выполнивший запрос

# ИЗБЕГАНИЕ ТУПИКОВ. БЕЗОПАСНОЕ СОСТОЯНИЕ.

- Состояние называется безопасным, если для него имеется последовательность процессов  $\{P_1, P_2, \dots, P_n\}$  такая, что для каждого  $P_i$  ресурсы, которые затребовал  $P_i$ , могут быть предоставлены за счет имеющихся незанятых ресурсов и ресурсов, выделенных всем процессам  $P_j$ , где  $j < i$
- Состояние безопасно, поскольку ОС может гарантированно избежать тупика посредством блокирования любых новых запросов, пока не выполнится безопасная последовательность
- Данная стратегия основана на идее избегания возникновения циклического ожидания

# ИЗБЕГАНИЕ ТУПИКОВ. БЕЗОПАСНОЕ СОСТОЯНИЕ.

- Предположим имеются 12 устройств хранения

Процесс	МАХ потребность	Владеет	Может запросить
p0	10	5	5
p1	4	2	2
p2	9	2	7

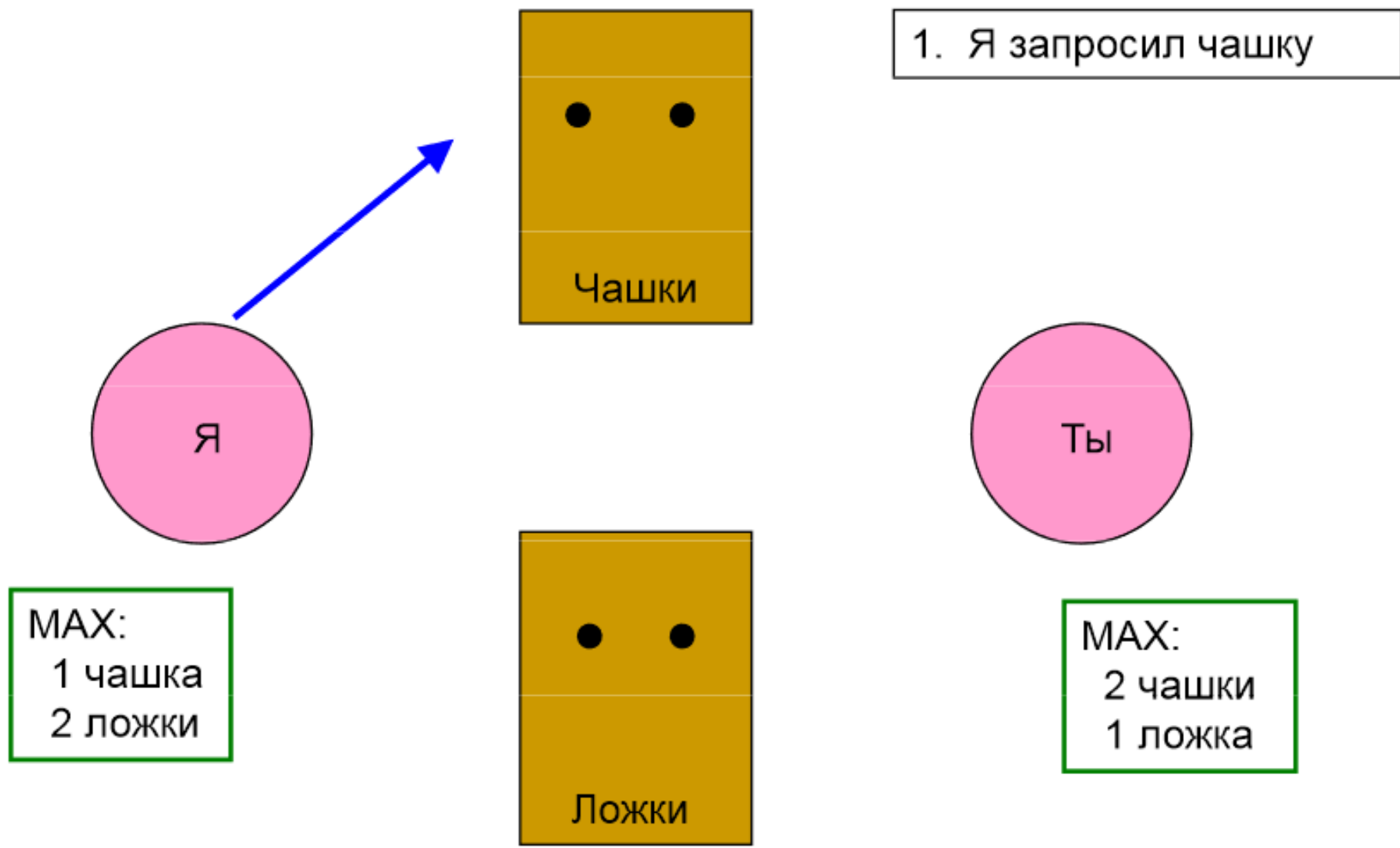
3 устройства свободны

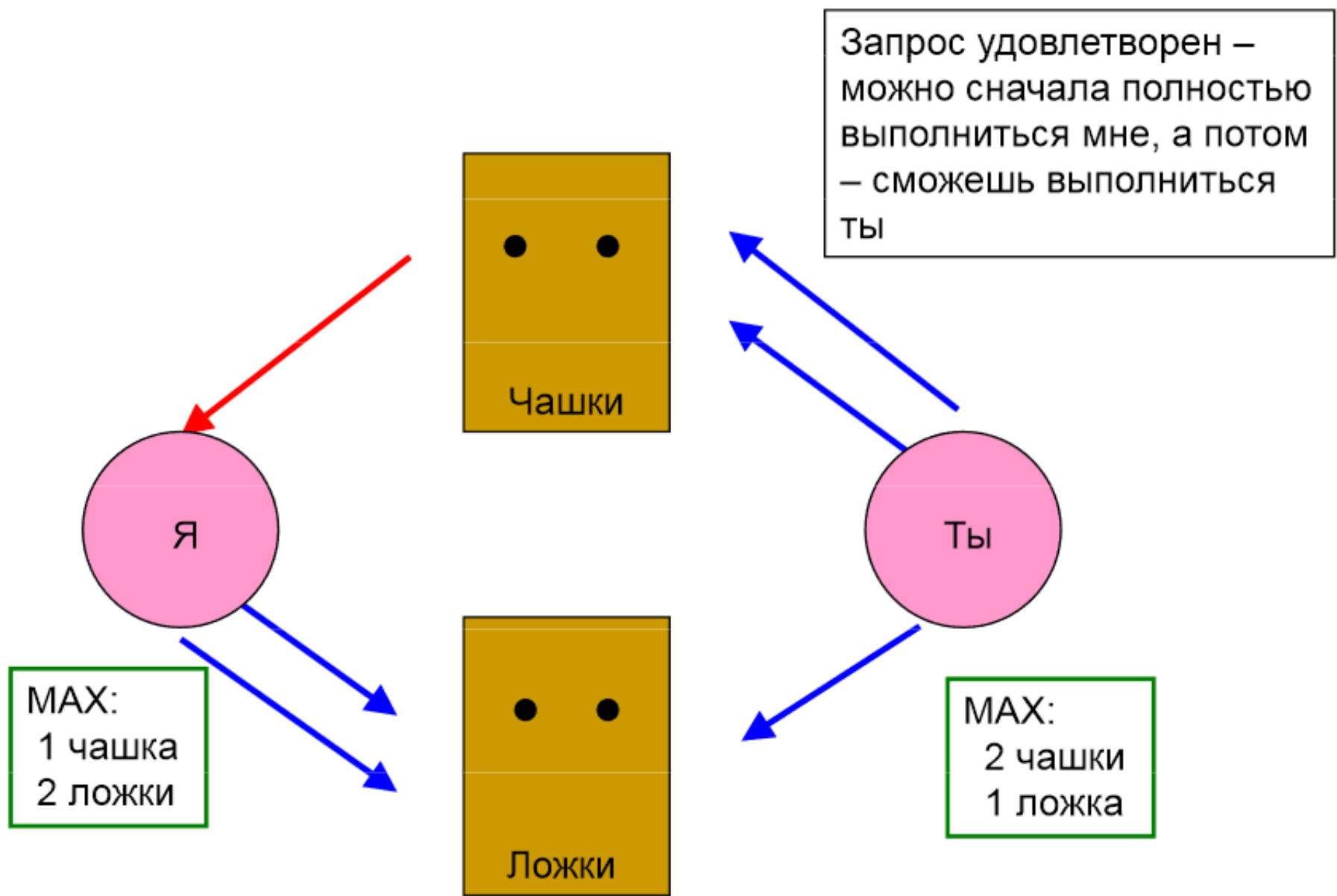
- Текущее состояние безопасно, поскольку существует безопасная последовательность: <p1,p0,p2>
- P<sub>1</sub> может завершиться, используя только свободные ресурсы
- P<sub>0</sub> может завершиться, используя только свободные ресурсы + ресурсы, которые сейчас выделены процессу P<sub>1</sub>
- P<sub>2</sub> может завершиться, используя свободные ресурсы + ресурсы, которые сейчас выделены процессам P<sub>1</sub> и P<sub>0</sub>
- Если P<sub>2</sub> запросит еще 1 устройство, удовлетворение запроса будет отложено, поскольку оно переведет систему в небезопасное состояние

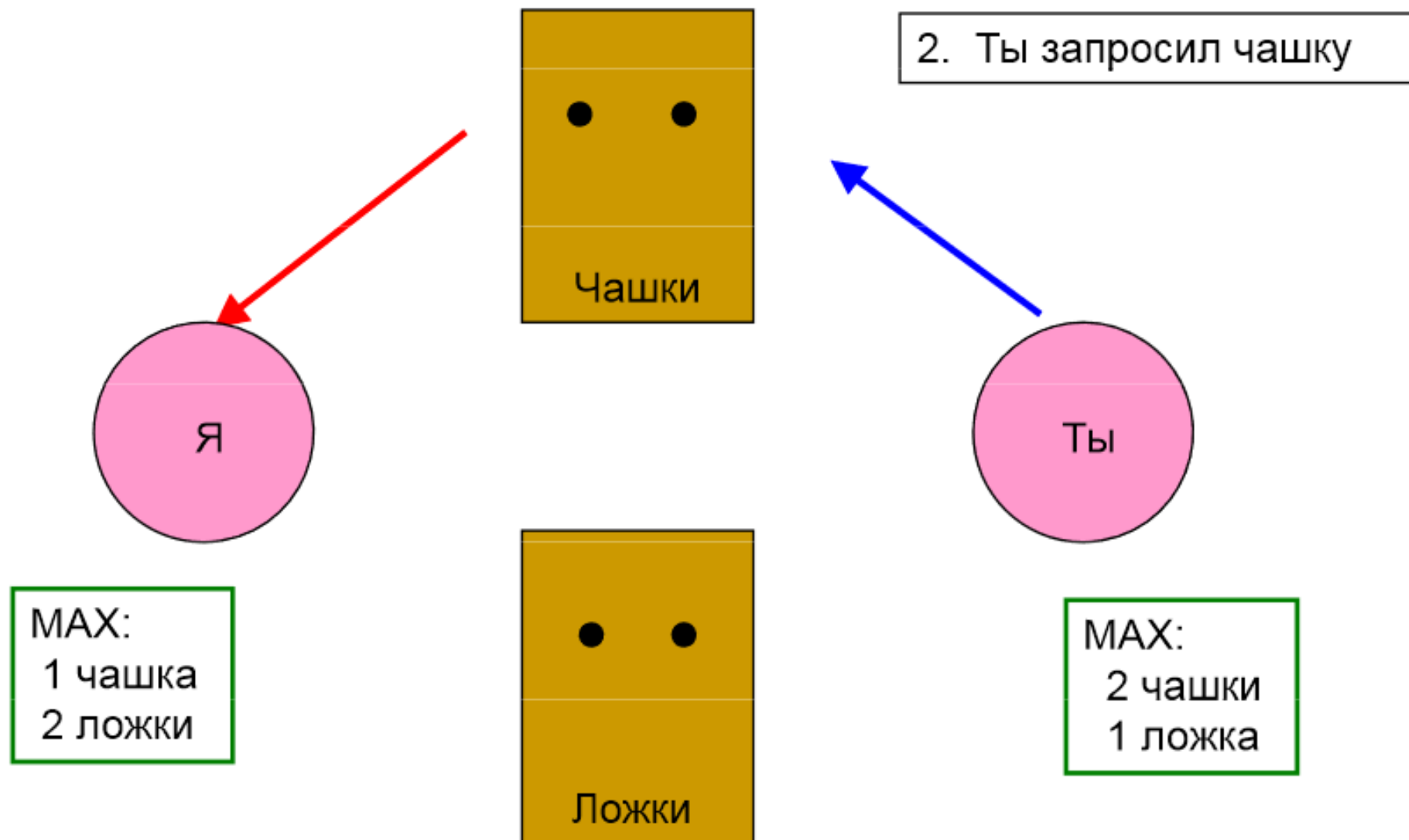


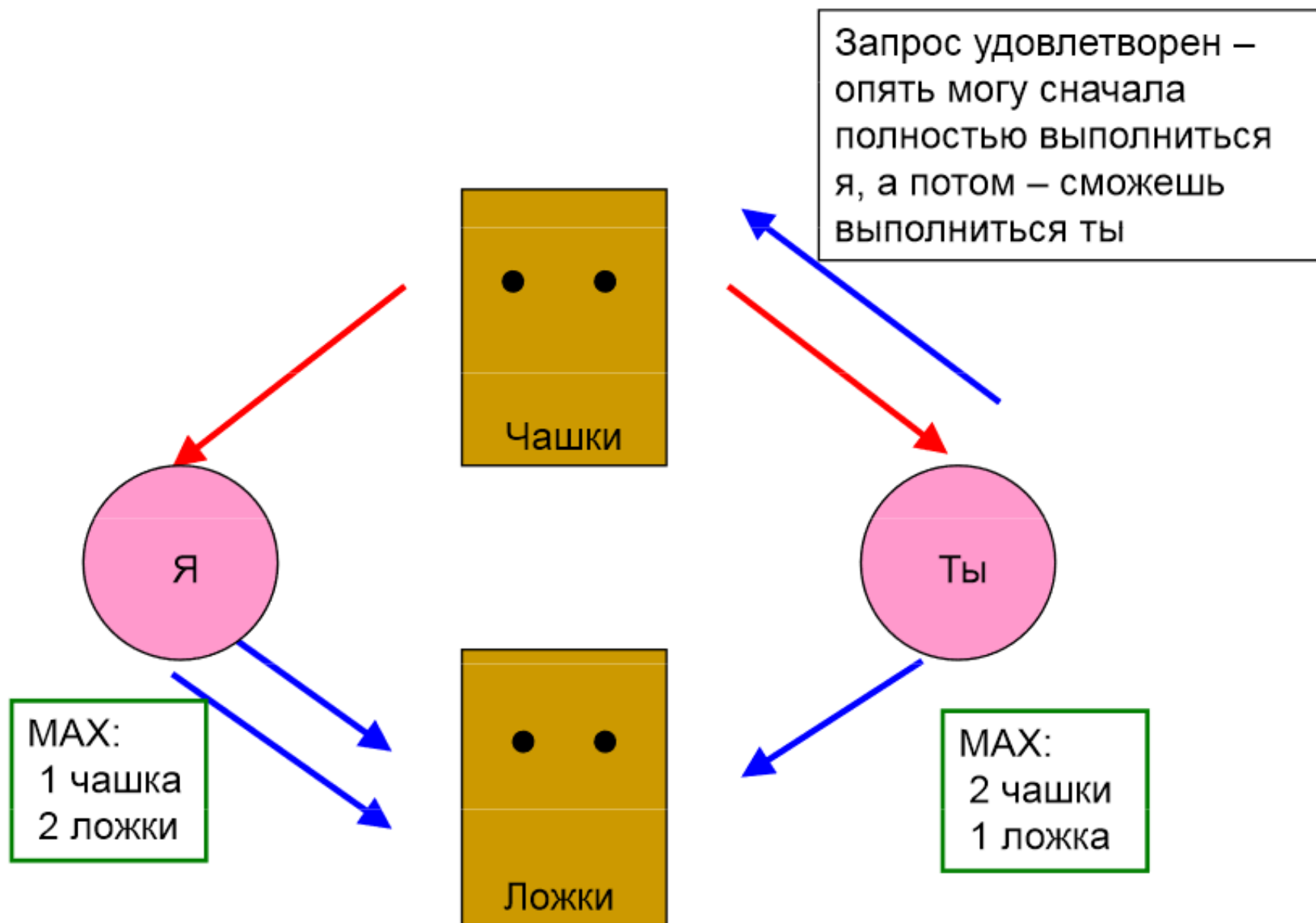
# ИЗБЕГАНИЕ ТУПИКОВ. ГРАФ "ПРОЦЕСС-РЕСУРС".

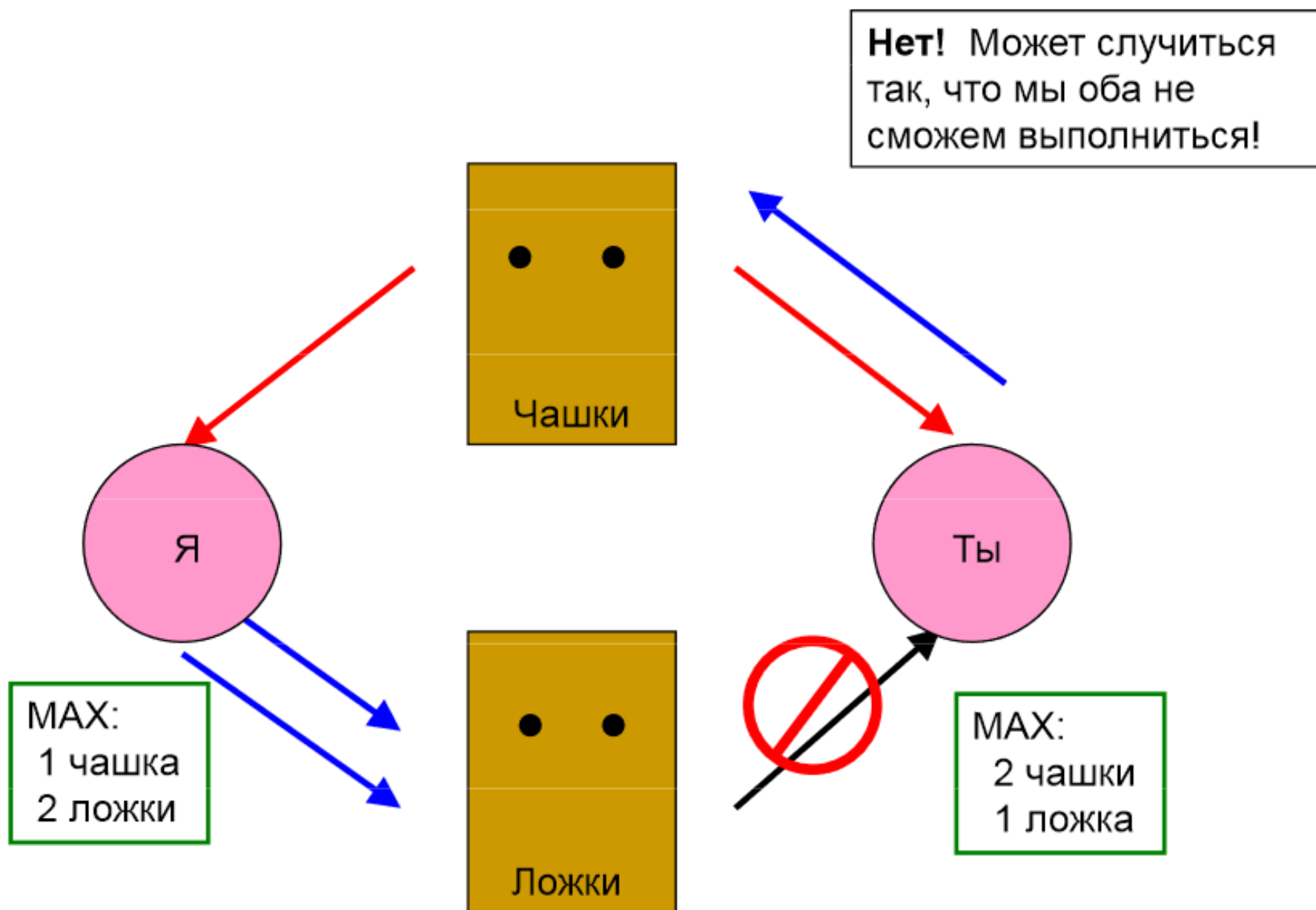
- Алгоритм банкира можно проиллюстрировать, используя граф "процесс-ресурс"
- При выполнении запроса:
  - представляем, что мы его удовлетворили
  - представляем, что все остальные возможные запросы удовлетворены
  - проверяем, может ли граф "процесс-ресурс" быть полностью редуцирован?
    - если да – выделяем запрошенный ресурс
    - если нет – блокируем процесс, выполнивший запрос

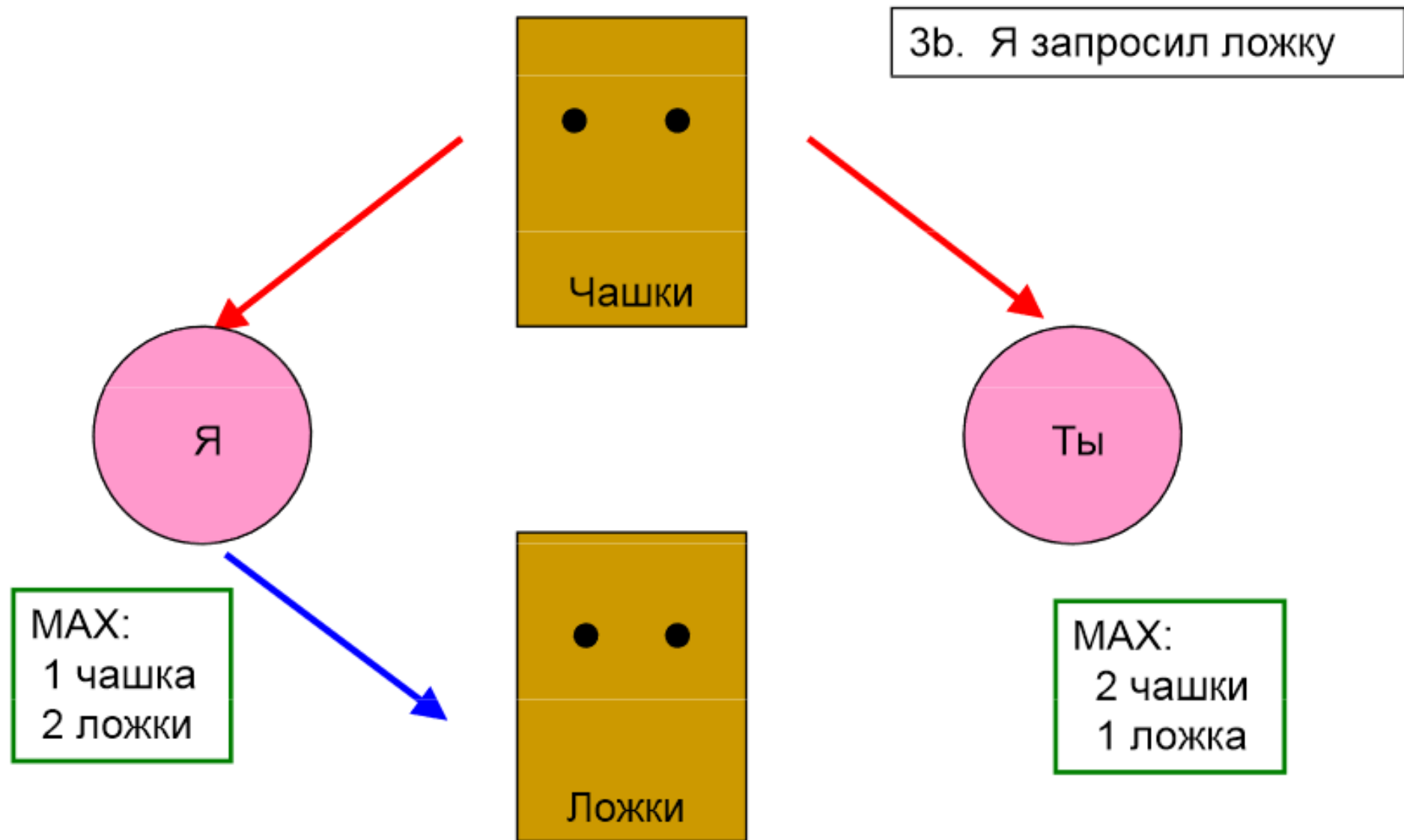


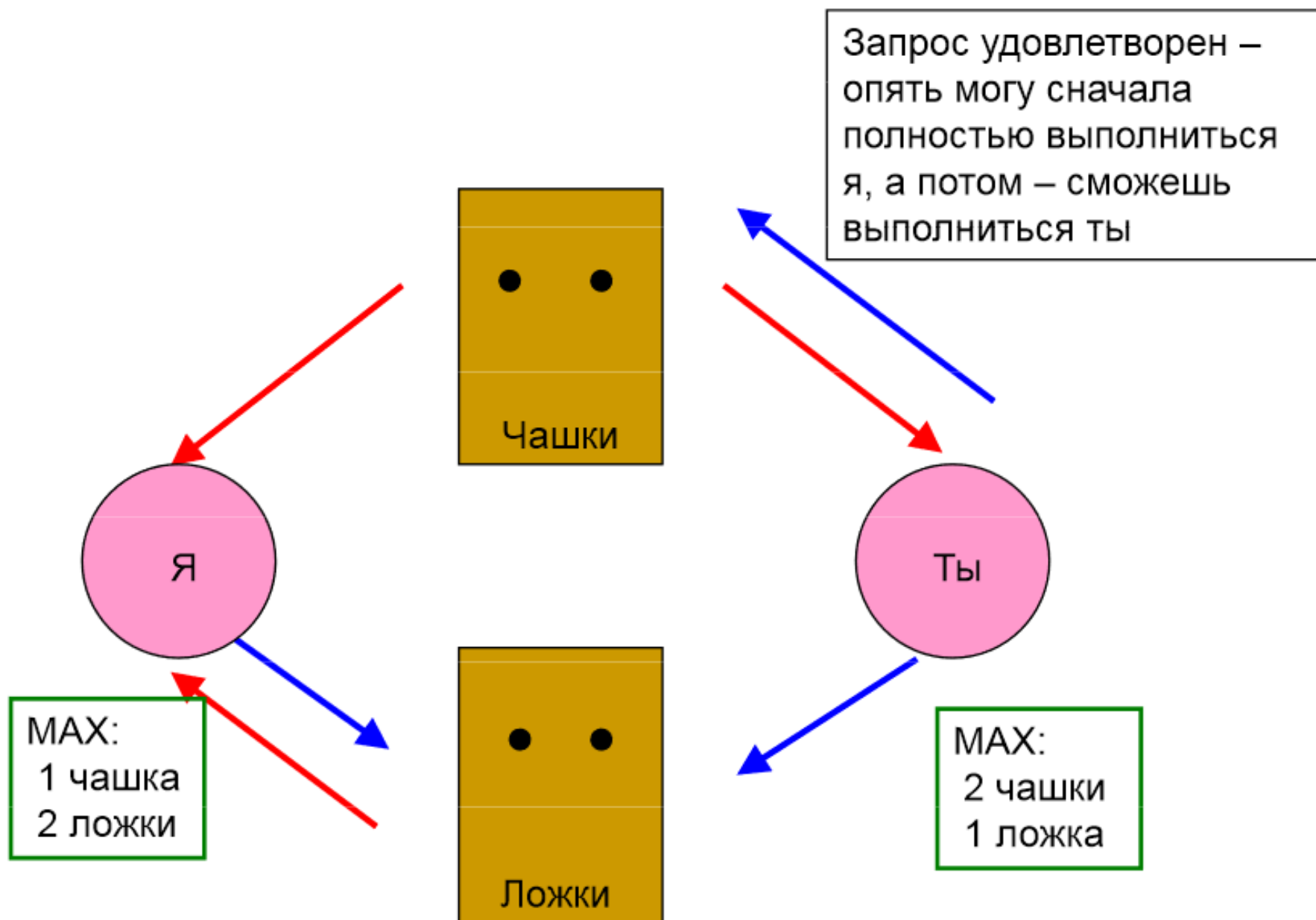














A,B,C - ресурсы

	Выделено			Max			Доступно		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Безопасно ли текущее состояние?

A,B,C - ресурсы

	Выделено			Max			Доступно		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Можно ли удовлетворить запрос P0 на (1,2,2)?

# ОБНАРУЖЕНИЕ ТУПИКА. ВОССТАНОВЛЕНИЕ ПОСЛЕ ТУПИКА.

Если ни один из перечисленных подходов не используется, может возникнуть тупик. В таком случае необходимо:

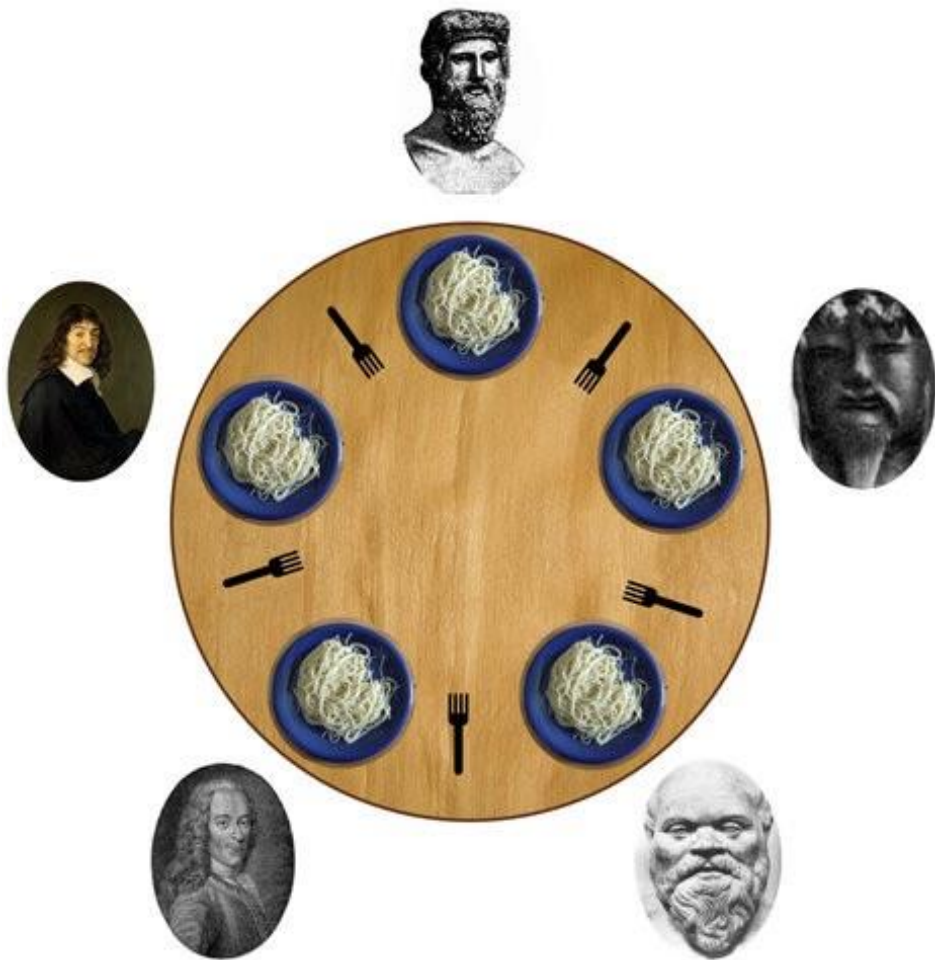
- Обнаружить возникновение тупика. Для этого нужно:
  - отслеживать выделение ресурсов (какой процесс каким ресурсом владеет)
  - отслеживать поступающие запросы (какой процесс какой ресурс ожидает)
- Иметь решения для восстановления из тупика

Очень накладно поддерживать и обнаружение, и восстановление

# ВОССТАНОВЛЕНИЕ ПОСЛЕ ТУПИКА

- Уничтожение одного/всех процессов, участвующих в тупике
  - Можно продолжать уничтожать процессы, пока тупик не распадется
  - Все вычисления уничтоженных процессов придется повторить
  - Грубо, но эффективно
- Перераспределение ресурсов между процессами вплоть до разрушения тупика
  - Ресурсы отбираются у владельцев и отдаются другим процессам
- Откат выбранного процесса к некоторой контрольной точке или к началу (partial or total rollback)

# ОБЕДАЮЩИЕ ФИЛОСОФЫ



Пять безмолвных философов сидят вокруг стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов.

Каждый философ может:

- Размышлять
- Есть

Условия:

- философ может есть только тогда, когда держит в руках две вилки – взятую справа и слева
- Взятие каждой вилки и возвращение её на стол являются отдельными действиями, которые должны выполняться одно за другим

Задача: разработать такой алгоритм, при котором ни один из философов не будет голодать.

# ВЗАИМОБЛОКИРОВКА. РЕЗЮМЕ.

- Взаимоблокировка (тупик) – проблема, возникающая при совместном использовании неразделяемых ресурсов
- Существующие средства борьбы с взаимоблокировками
- "Страусовый" алгоритм
- Предупреждение взаимоблокировок
- Избегание взаимоблокировок
- Обнаружение взаимоблокировок и восстановление после них