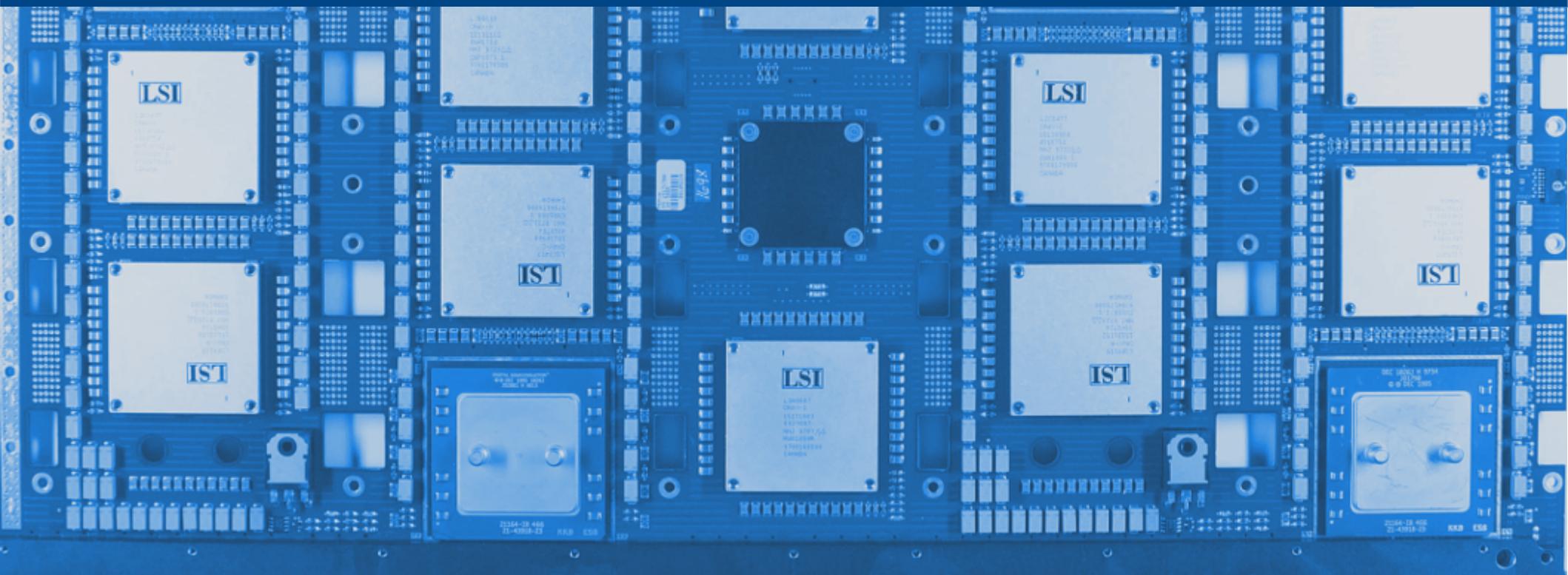


Параллельное программирование



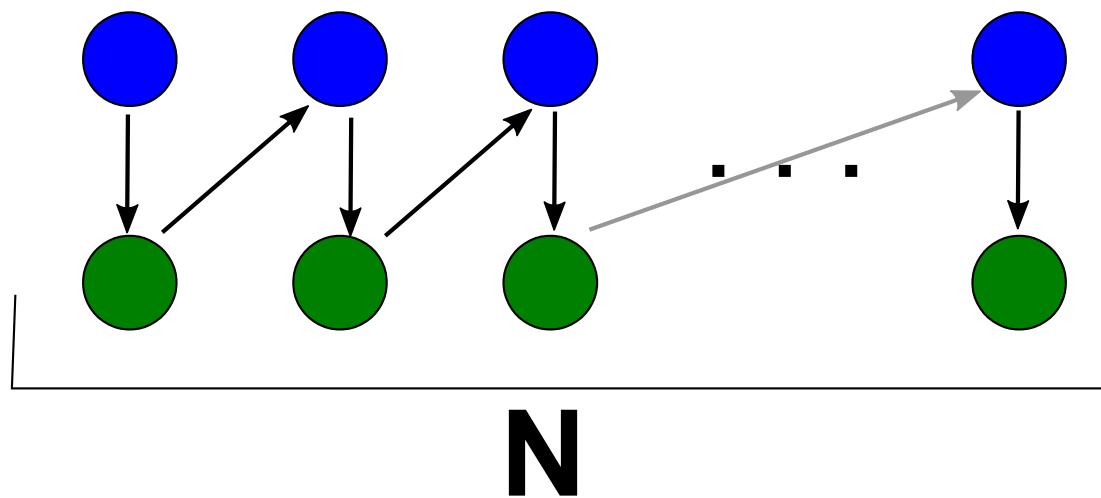
Яковлев Виктор Вадимович

(c) 2014-2017

Графовые модели программ

```
for (i=0; i<N; ++i) {  
    A = A + 1;  
    B = A;  
}
```

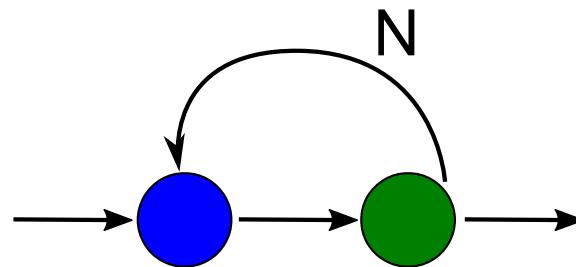
Операционная история



Графовые модели программ

```
for (i=0; i<N; ++i) {  
    A = A + 1;  
    B = A;  
}
```

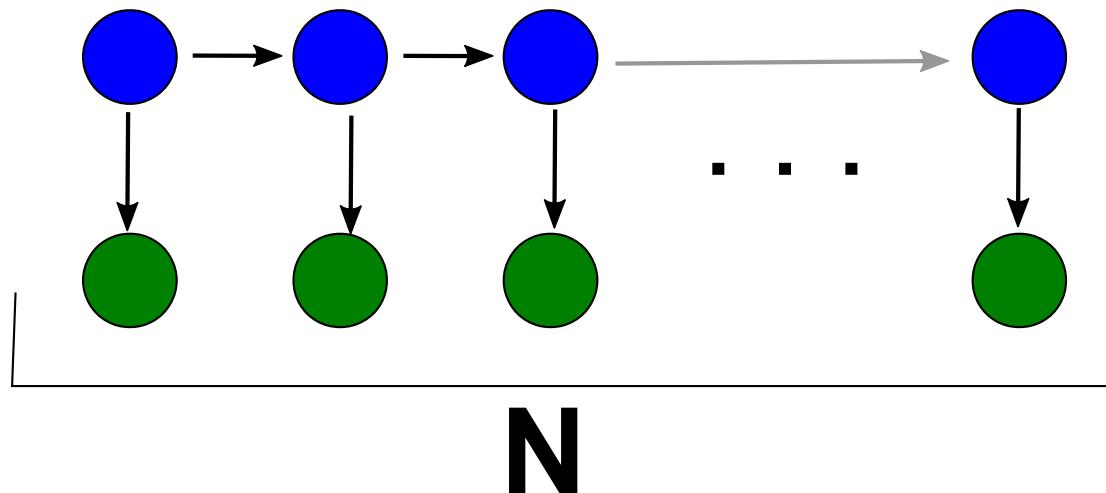
Граф управления



Графовые модели программ

```
for (i=0; i<N; ++i) {  
    A = A + 1;  
    B = A;  
}
```

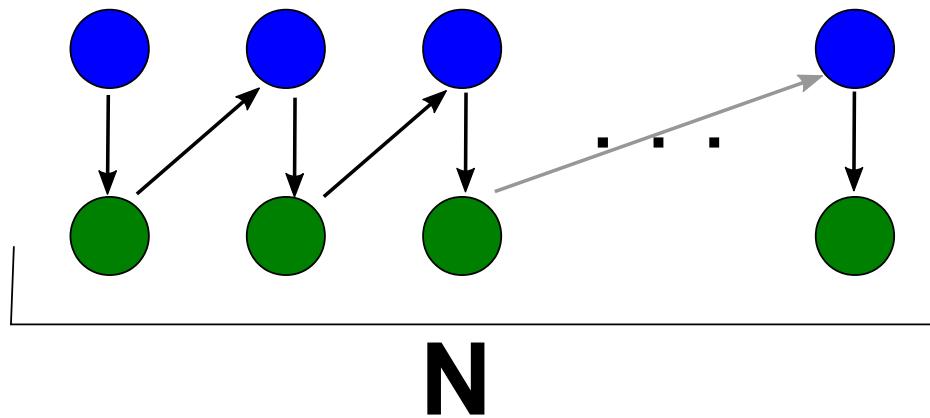
История информации



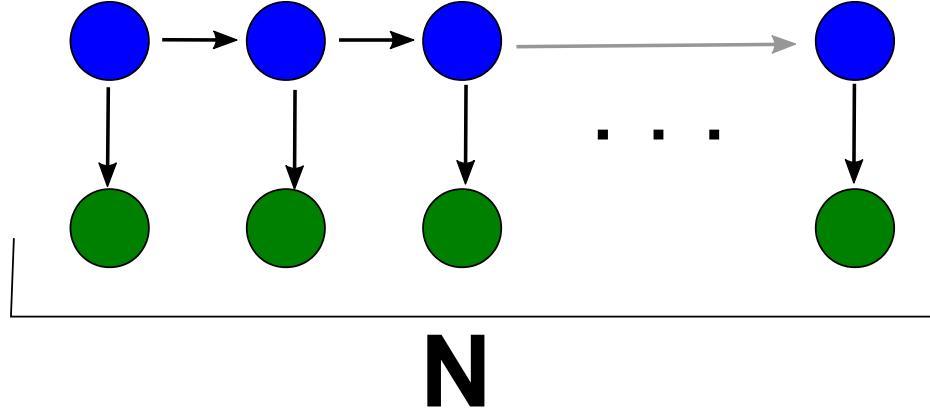
Графовые модели программ

операционные графы

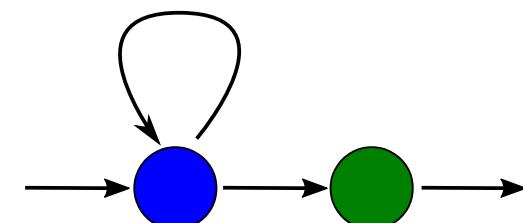
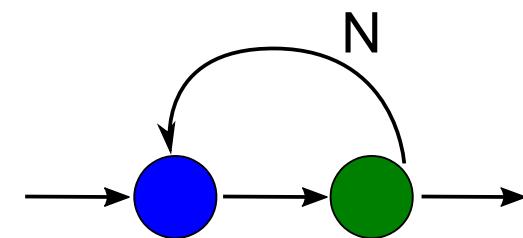
истории



информационные графы

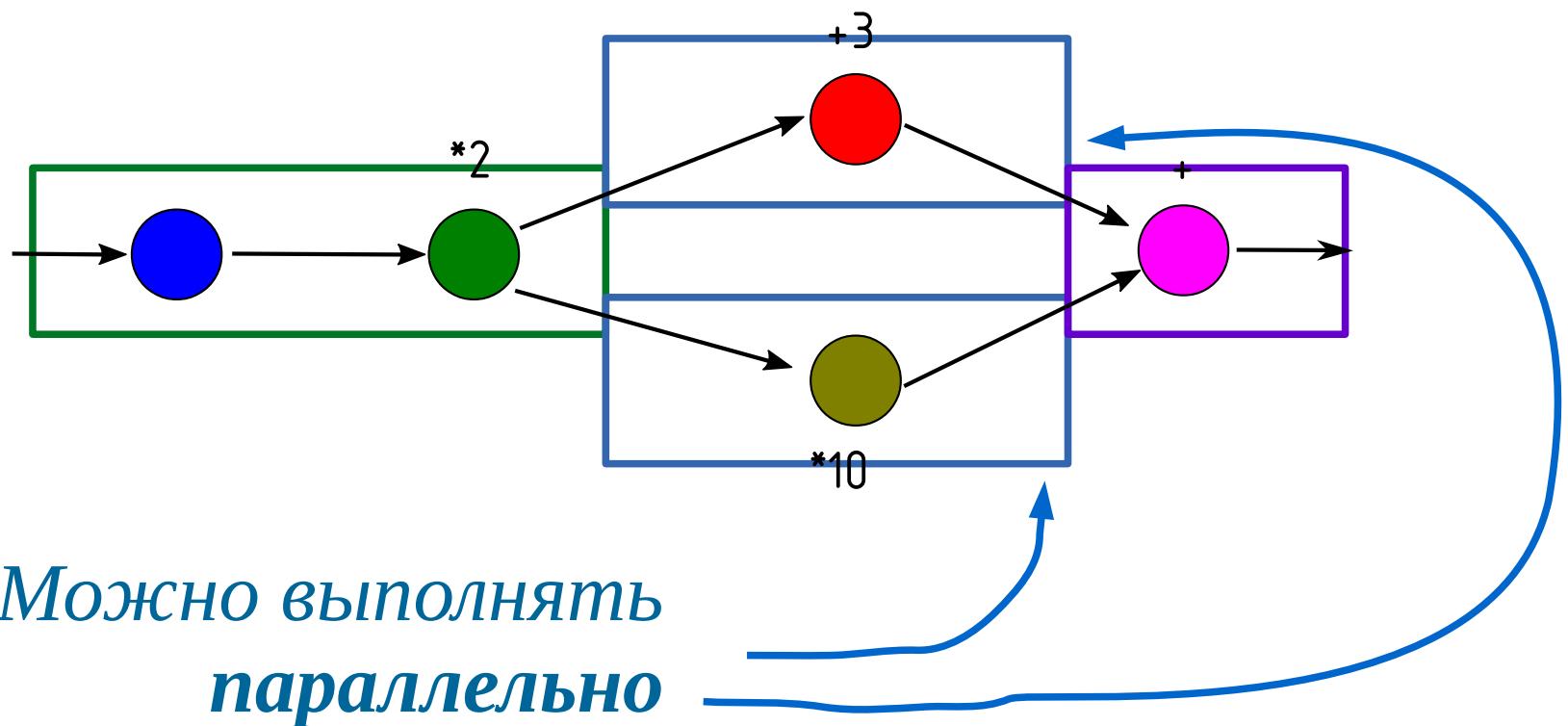


компактные представления



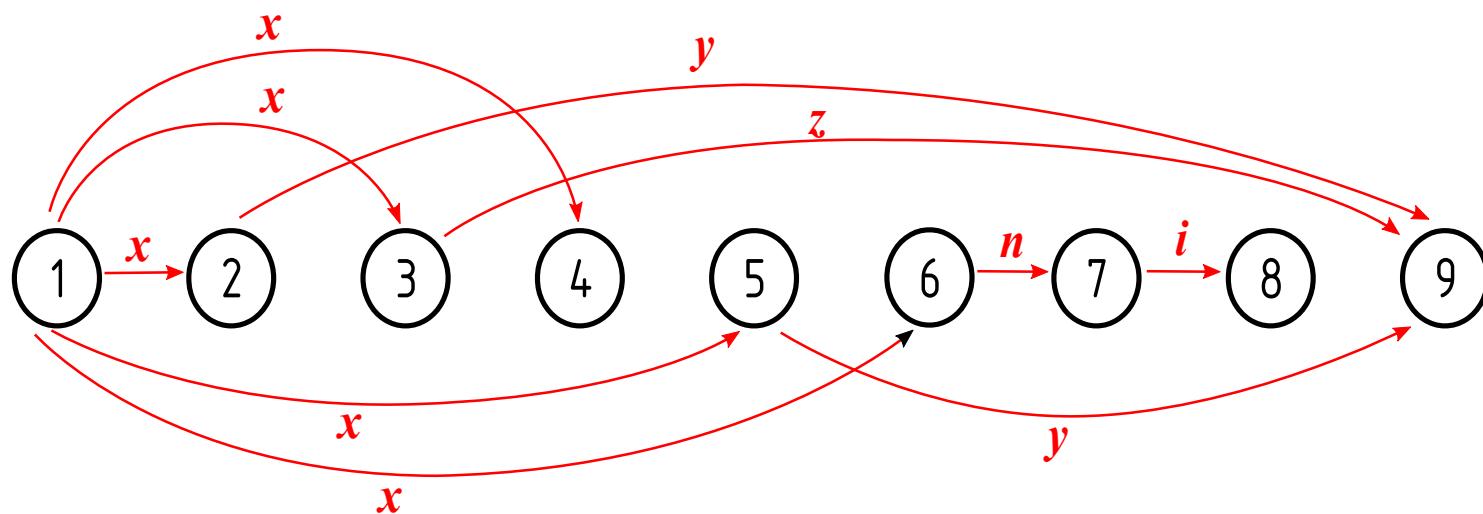
Информационный граф программы

```
A = 1;  
B = A * 2;  
C = B + 3;  
D = B * 10;  
E = C + D;
```



Операционно-информационный граф программы

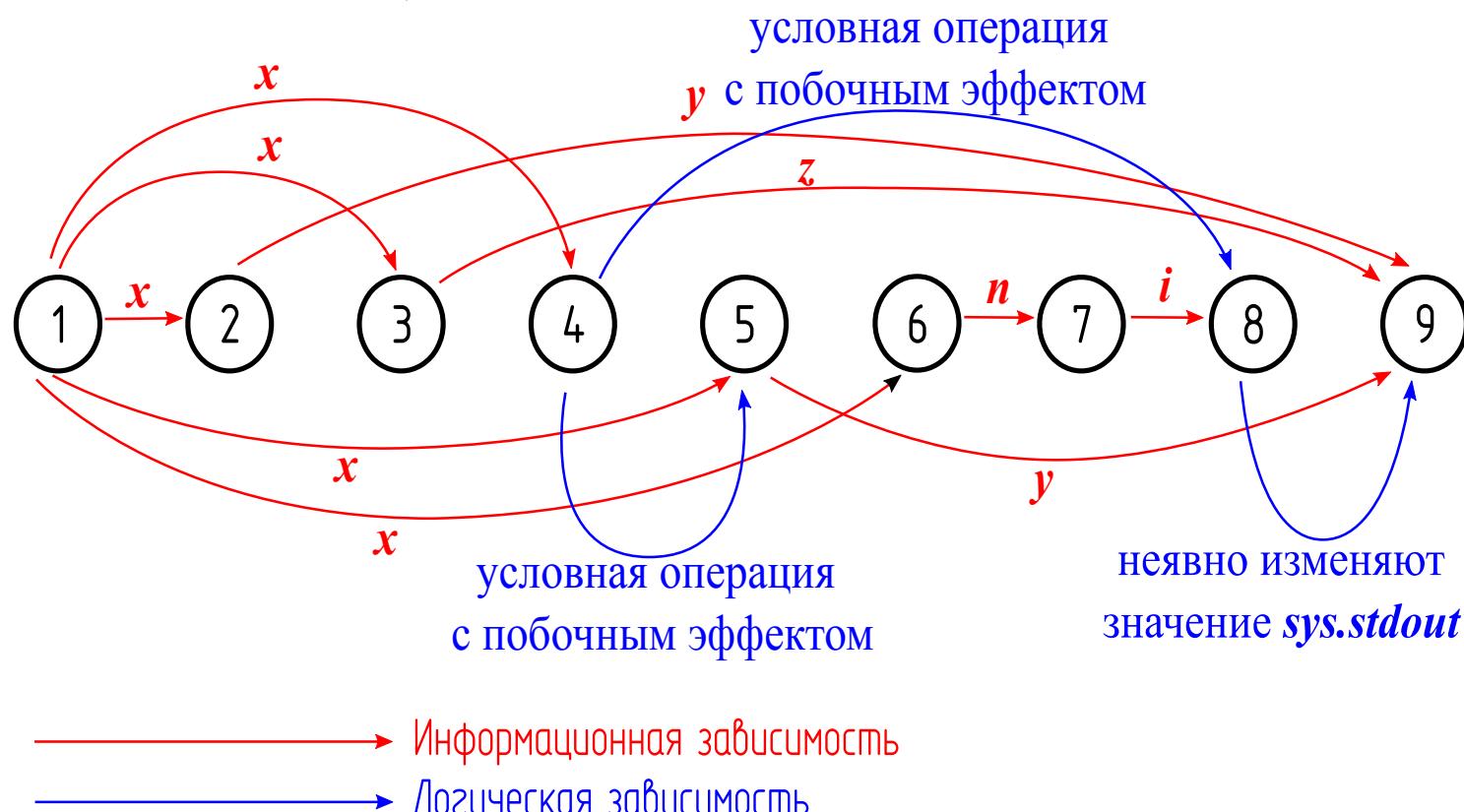
```
1:  input(x)
2:  y = x
3:  z = x + 1
4:  if x < 0:
5:      y = -x
_:
6:      n = x * 2
7:      for i in range(0, n):
8:          print(i)
9:  print(y, z)
```



→ Информационная зависимость

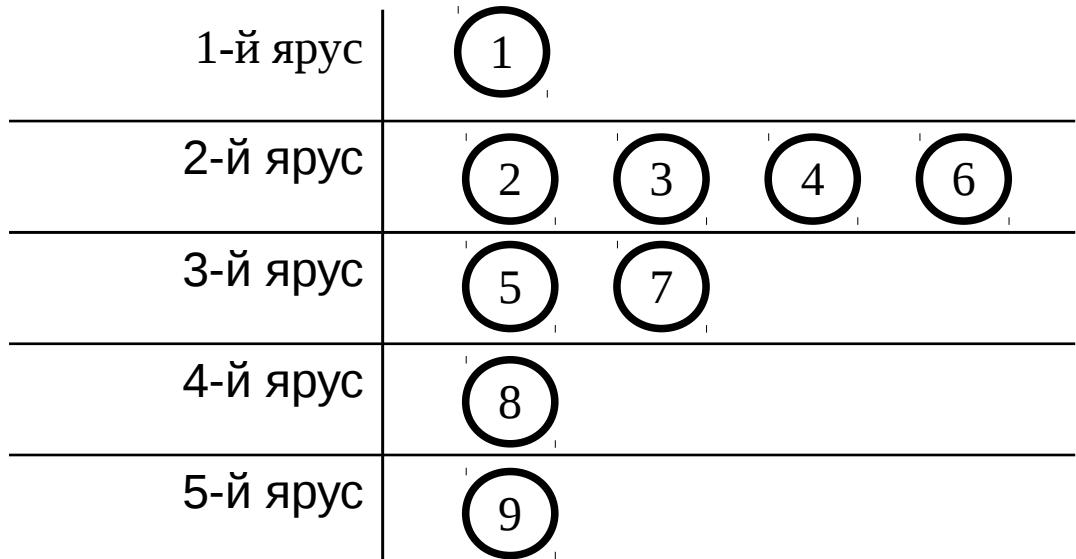
Операционно-информационный граф программы

```
1:  input(x)
2:  y = x
3:  z = x + 1
4:  if x < 0:
5:      y = -x
_:
6:  n = x * 2
7:  for i in range(0, n):
8:      print(i)
9:  print(y, z)
```

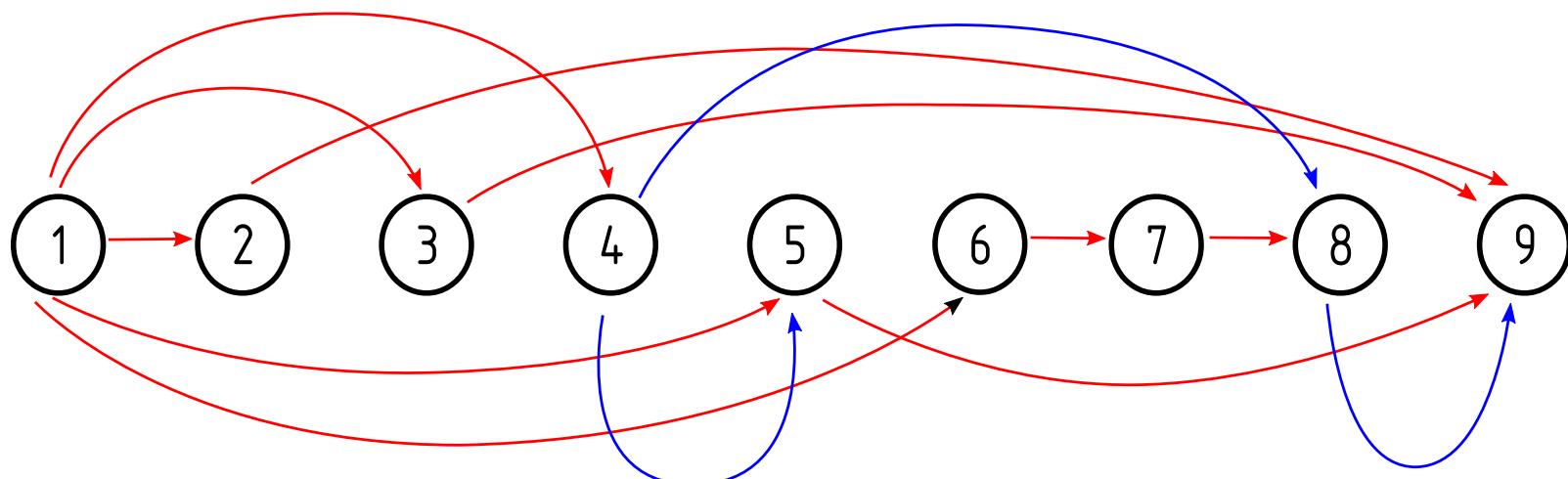


Ярусно-параллельная форма программы

```
1:  input(x)
2:  y = x
3:  z = x + 1
4:  if x < 0:
5:      y = -x
6:  else:
7:      n = x * 2
8:      for i in range(0, n):
9:          print(i)
9:  print(y, z)
```

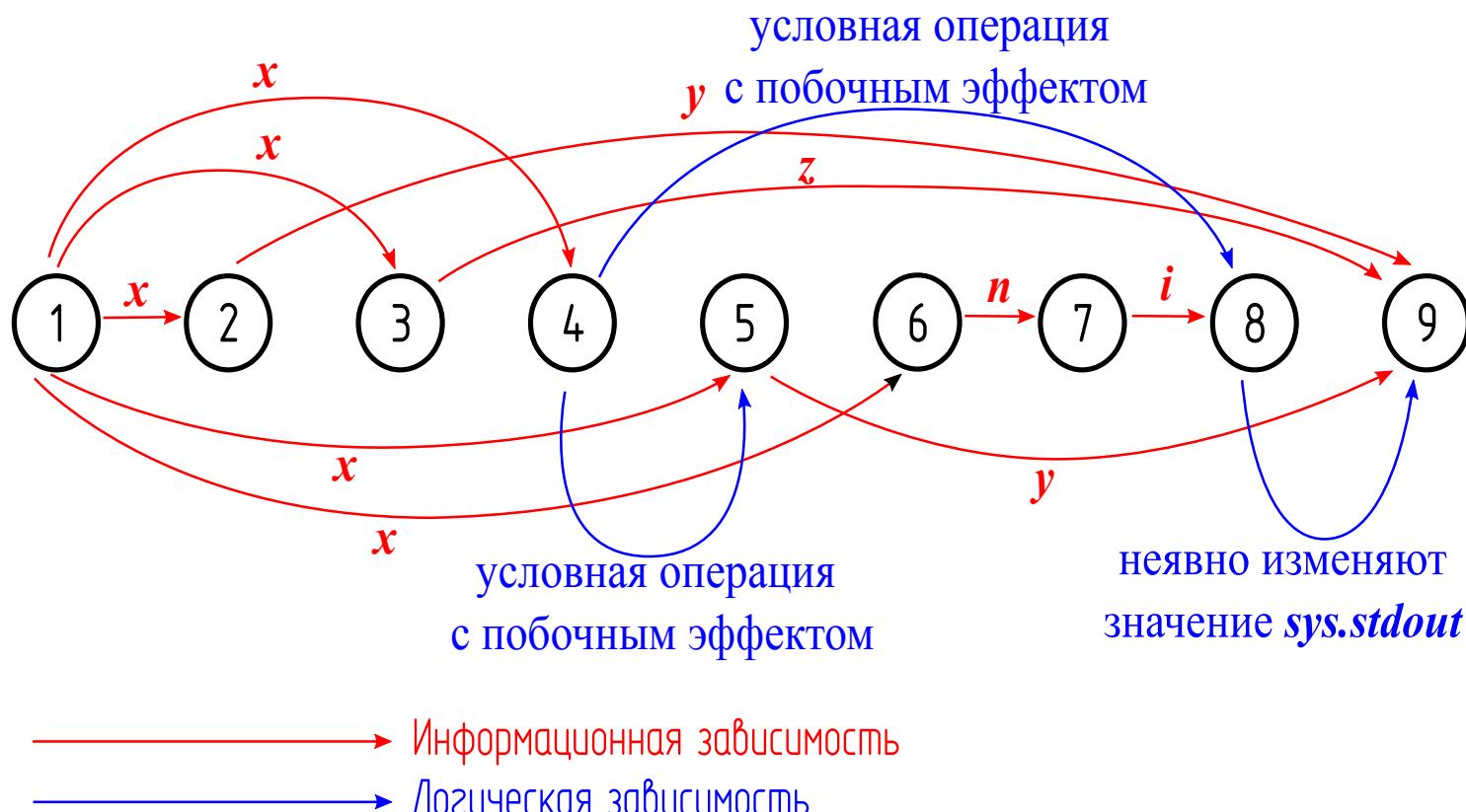


1. Первый ярус содержит все узлы, которые не зависят от других узлов
2. $k+1$ ярус содержит все узлы, которые зависят от узлов графа, принадлежащих k -му, $k-1$ -му, $k-2$ -му ... 1-му ярусу.



Логическая зависимость при распараллеливании

- Изменения значений переменных
- Вызов функций с побочными эффектами



Логическая зависимость при распараллеливании

- Изменения значений переменных

- Запретить изменения переменных

and

- Гарантировать неизменяемость используемых структур данных

- Вызов функций с побочными эффектами

- Запретить функции с побочными эффектами (**чистый функциональный** язык Haskell)

or

- Свести к минимуму использование функций с побочными эффектами (**не чистые функциональные** языки программирования)

and

- Иметь возможность выявления функций с побочными эффектами на этапе компиляции/интерпретации

Неизменяемость: `const` и `restrict` в Си99

```
void func(const int * arr)
{
    /*
    const запрещает изменять
    значения arr, но
    не гарантирует, что это
    значения не поменяются
    извне
    */
}
```

```
caller() {
    int * arr = ...;
    pthread_create(&func, &arr);
    ...
    pthread_create(&func, &arr);
}
```

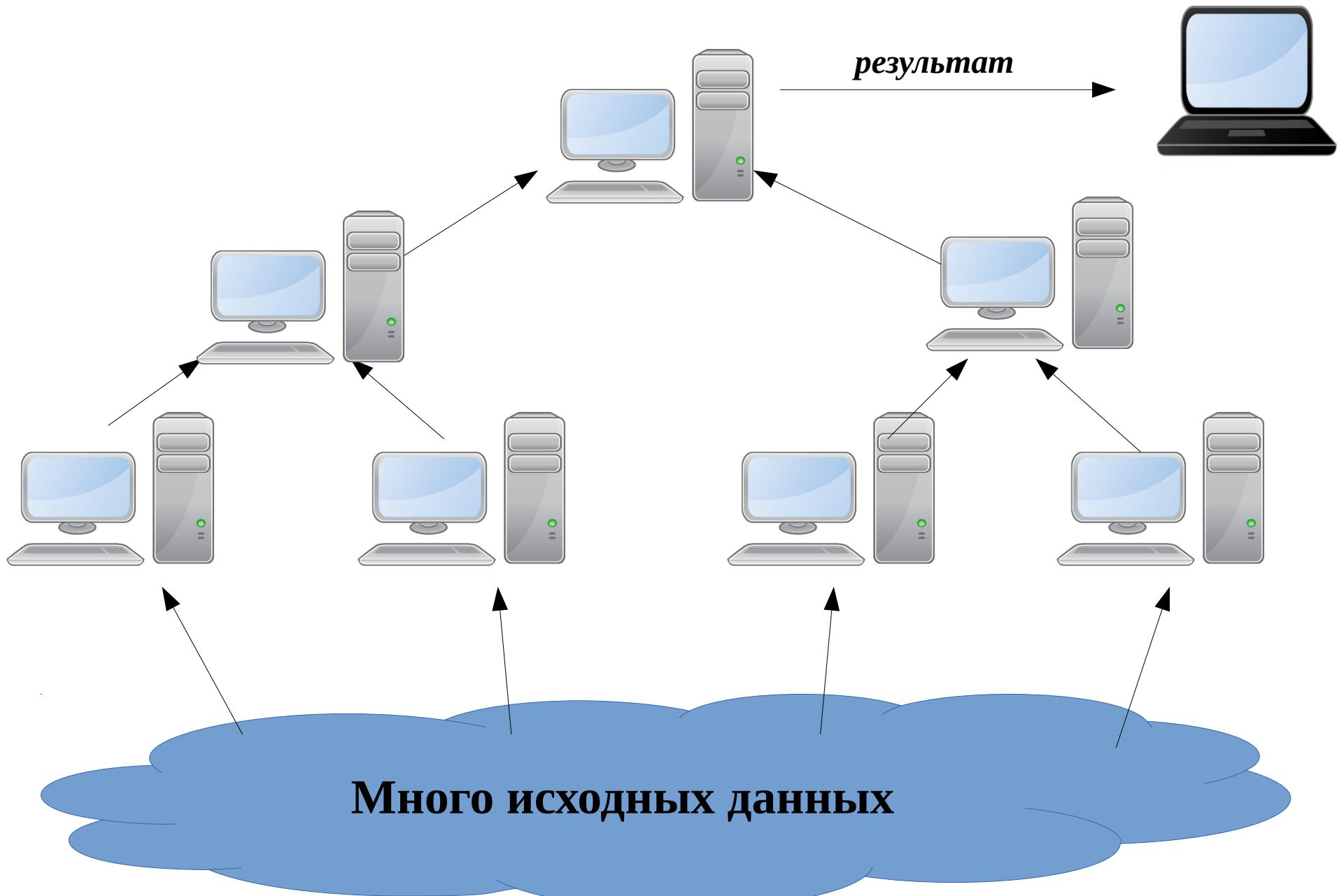
```
void func(const int *
restrict arr) {
    /*
    const запрещает изменять
    значения arr, а restrict
    гарантирует, что это
    значения не поменяются
    извне
    */
}
```

C++:

- __restrict__ – для g++, clang++
- __declspec(restrict) – для Microsoft Visual C++

*Ответственность за гарантию неизменяемости
данных лежит на разработчике!*

Парадигма MapReduce



Парадигма MapReduce

MAP: Отображение $\overline{\text{source}} \rightarrow \overline{\text{result}}$

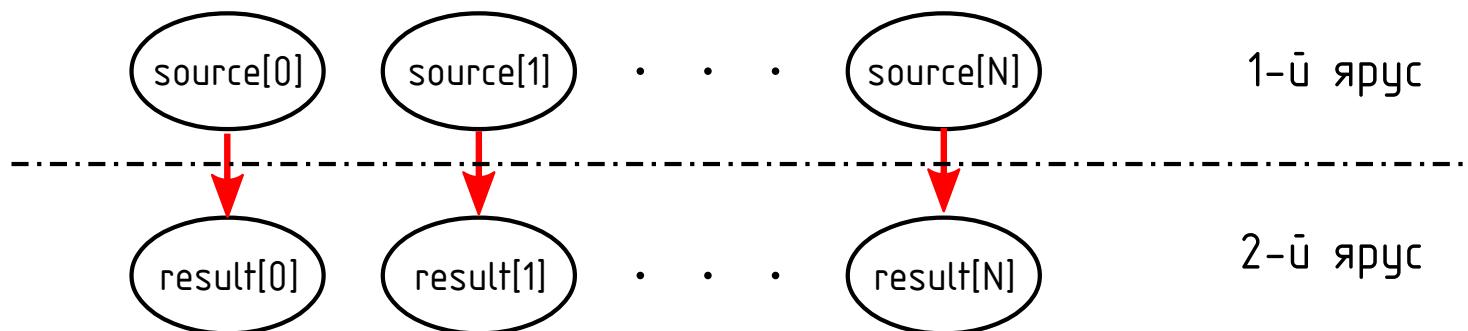
```
std::vector<T> map(std::function<T(T)> unary_func, const std::vector<T> & source)
{
    std::vector<T> result(source.size());
    for (size_t i=0; i<source.size(); ++i) {
        result[i] = unary_func(source[i]);
    }
    return result;
}
```

REDUCE (FOLD): Отображение $\overline{\text{source}} \rightarrow \text{result}$

```
T reduce(std::function<T(T,T)> binary_func, const T & init_val, const std::vector<T> & source)
{
    T result = init_val;
    for (size_t i=0; i<source.size(); ++i) {
        result = binary_func(result, source[i]);
    }
    return result;
}
```

Парадигма MapReduce

```
std::vector<T> map(std::function<T(T)> unary_func, const std::vector<T> & source)
{
    std::vector<T> result(source.size());
    for (size_t i=0; i<source.size(); ++i) {
        result[i] = unary_func(source[i]);
    }
    return result;
}
```

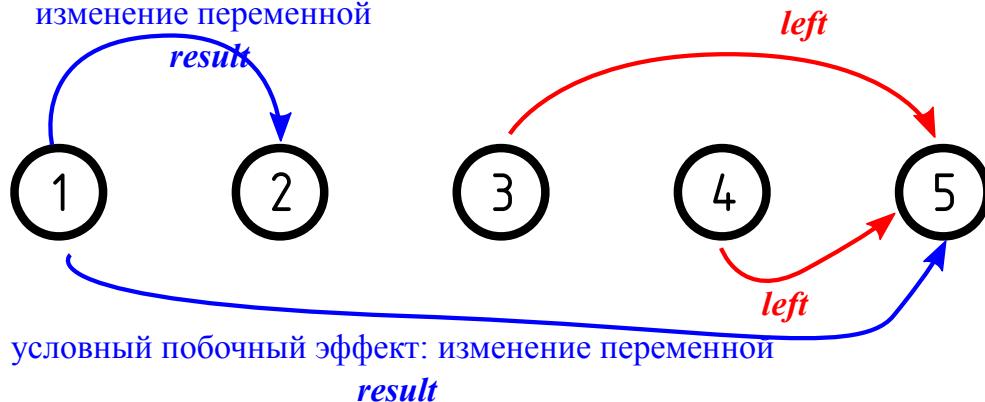


Парадигма MapReduce

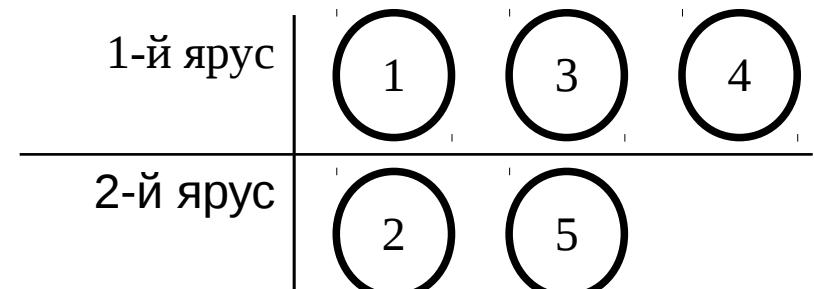
```
T reduce(std::function<T(T,T)> binary_func,
         const T & init_val, const std::vector<T> & source, size_t from, size_t to)
{
    T result;
    if (from == to-1) /* (1) */
        result = binary_func(init_val, source[from]); /* (2) */
    else {
        T left = reduce(binary_func, init_val, source, from, (to-from)/2); /* (3) */
        T right = reduce(binary_func, init_val, source, (to-from)/2, to); /* (4) */
        result = binary_func(left, right); /* (5) */
    }
    return result;
}
```

условный побочный эффект:

изменение переменной

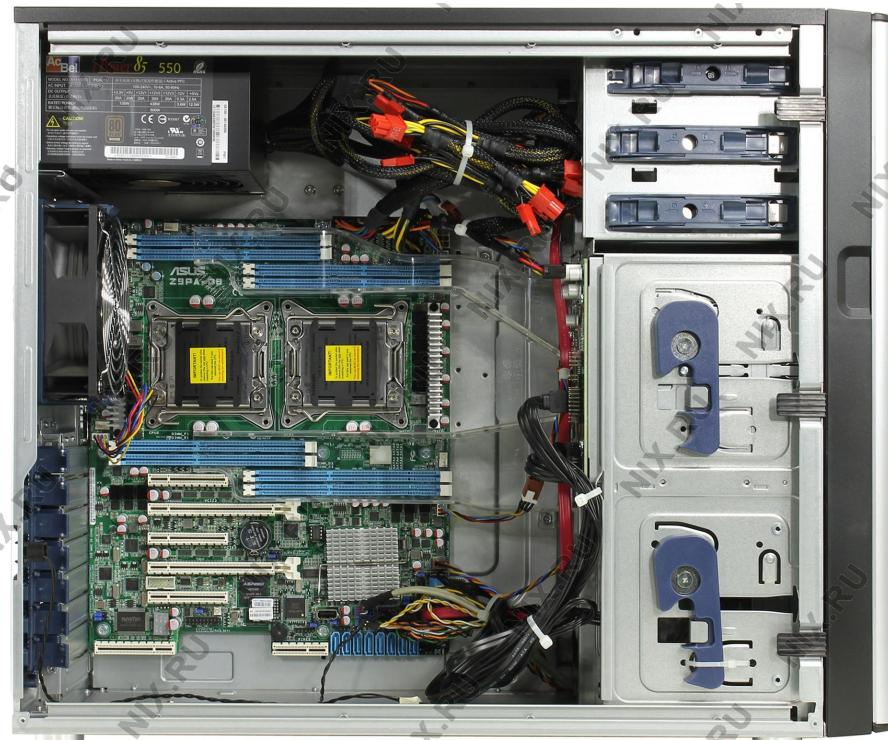


условный побочный эффект: изменение переменной
result



Параллельные вычислительные машины

Системы с общей памятью



Системы с распределенной памятью



- **Кооперативная многозадачность**
(Windows 2.x, 3.x, Mac OS до 9.x, виртуальные машины, встраиваемые системы)
- **Вытесняющая многозадачность**

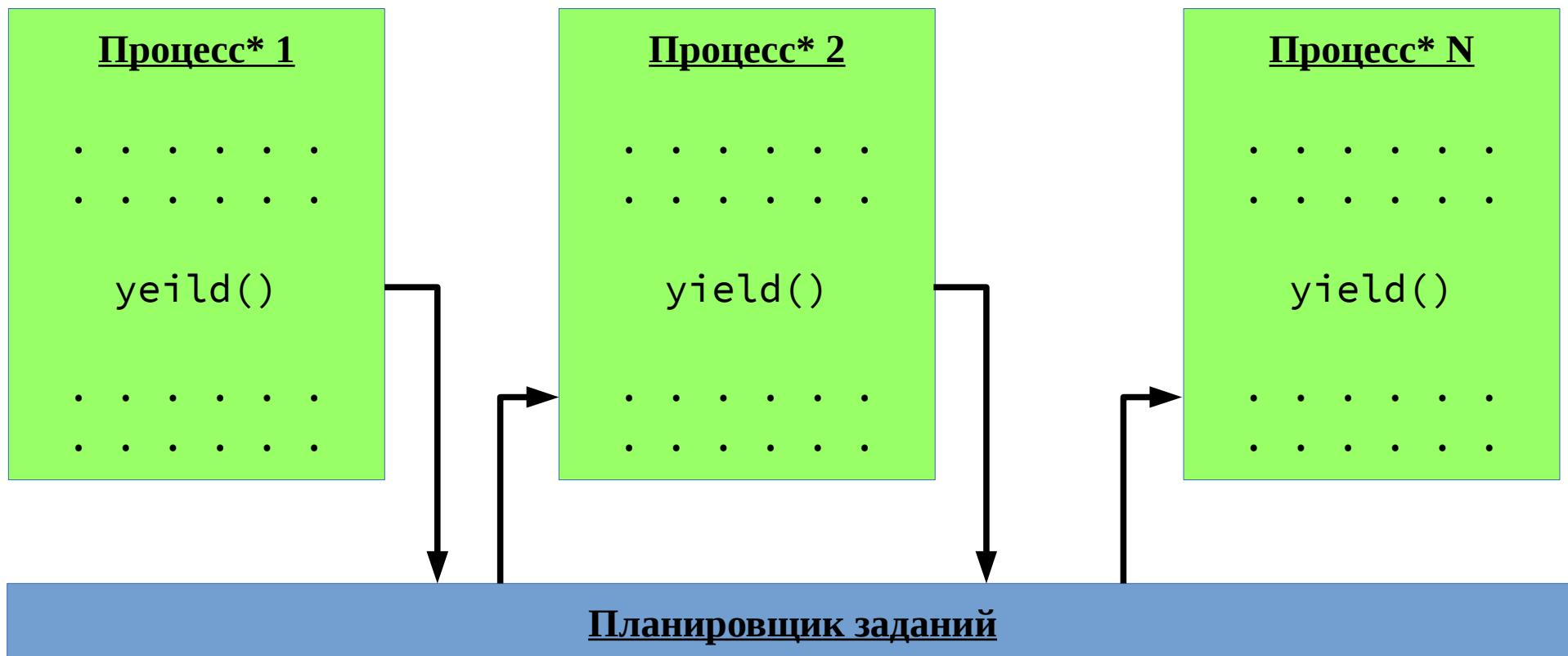
Кооперативная многозадачность

Достоинства

- Не требуется таймер для прерываний
- Минимальное время простоя

Недостатки

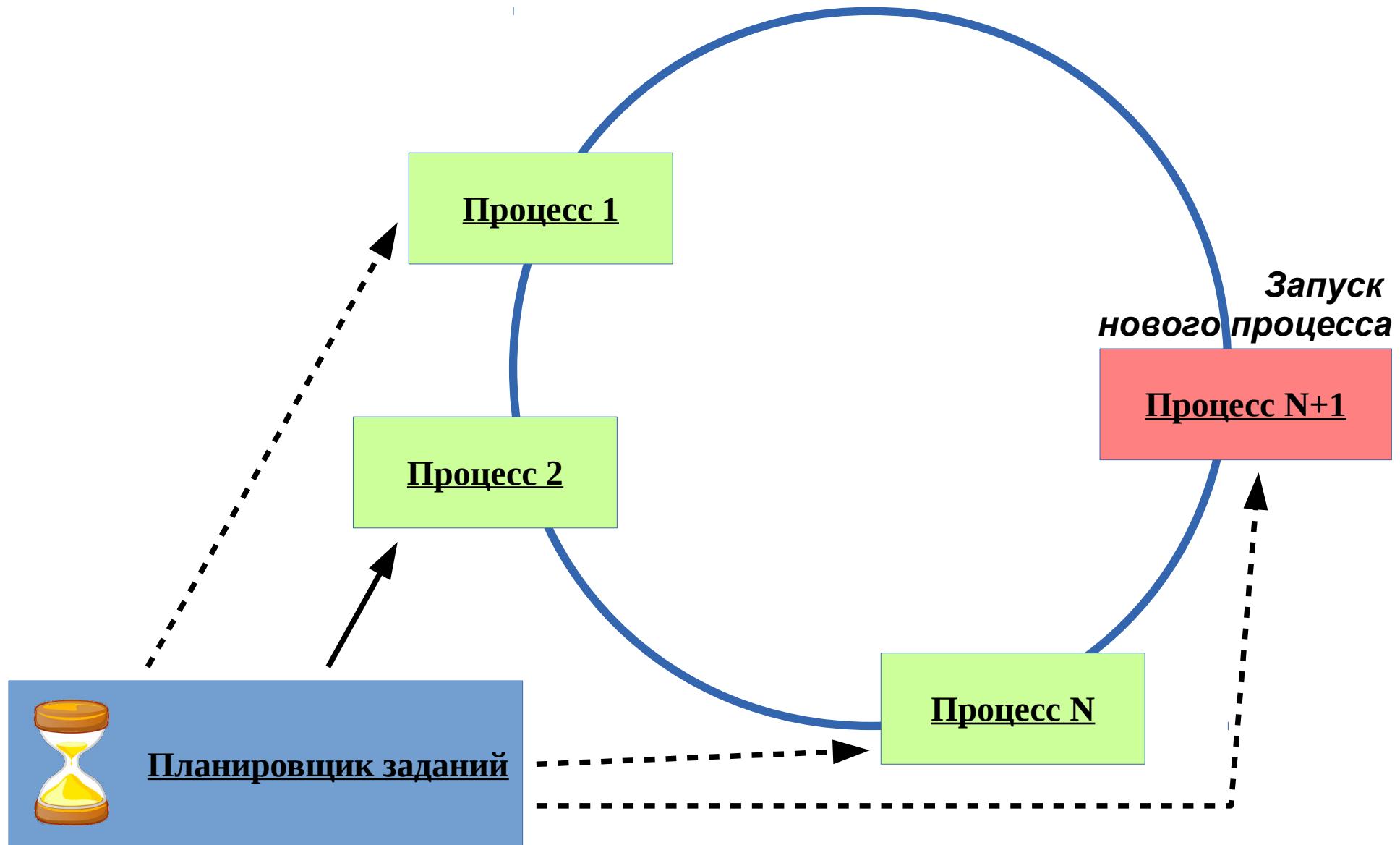
- При зависании одного процесса повисает вся система



* В данном контексте процесс и поток (нить) не различаются

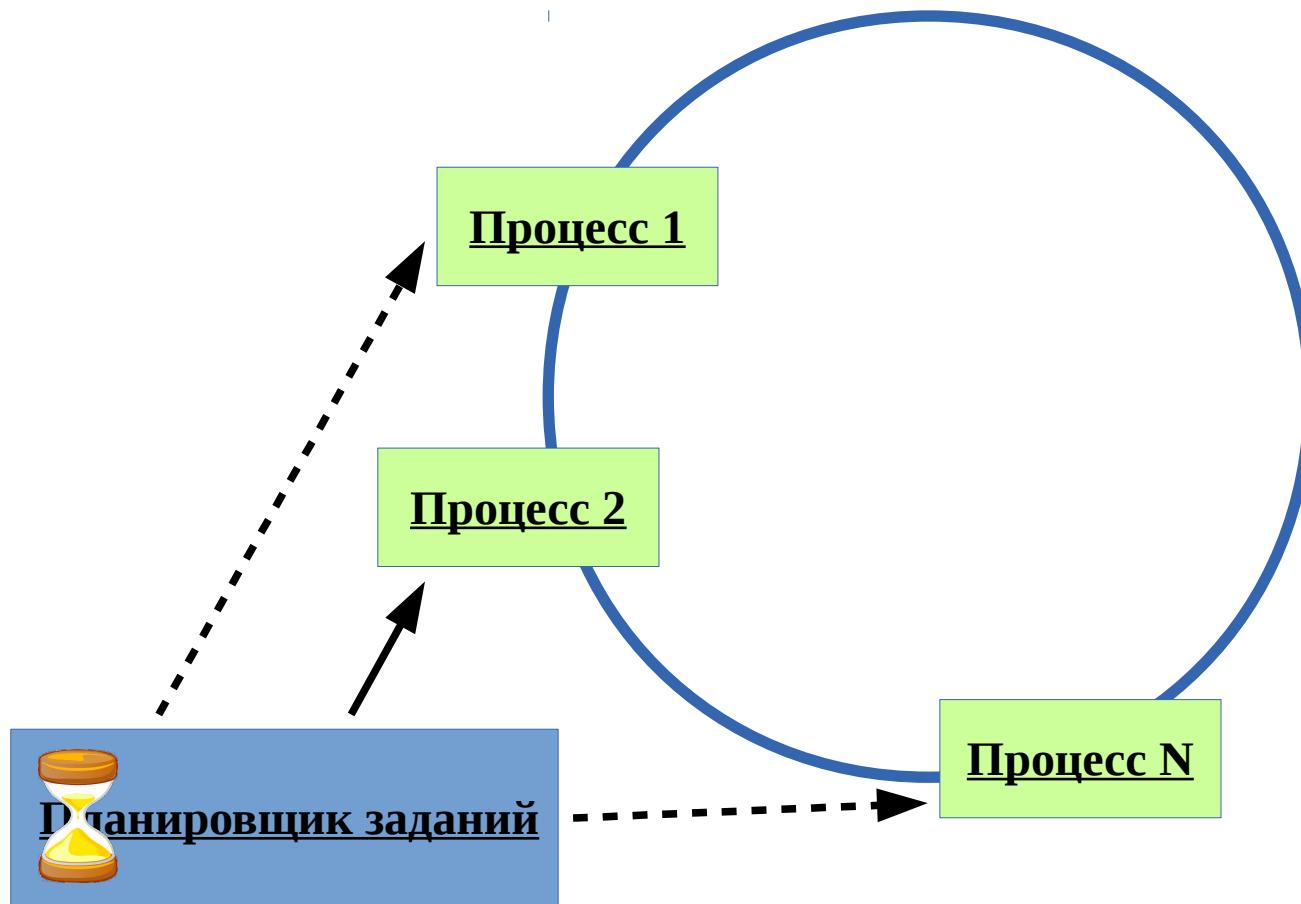
- **Кооперативная многозадачность**
(Windows 2.x, 3.x, MacOS до 9.x, виртуальные машины, встраиваемые системы)
- **Вытесняющая многозадачность**
 - **Round-Robin**
(Windows 95/98/ME, старые UNIX-системы)
 - **Multilevel Feedback Queue**
(Windows NT, Linux до 2.6, *BSD, Mac OS X)
 - **Вариации механизма Multilevel Feedback Queue:**
 - **O(1)**
(Linux 2.6.0 – 2.6.23)
 - **Completely Fair Scheduler**
(современные версии Linux)

Вытесняющая многозадачность: Round Robin

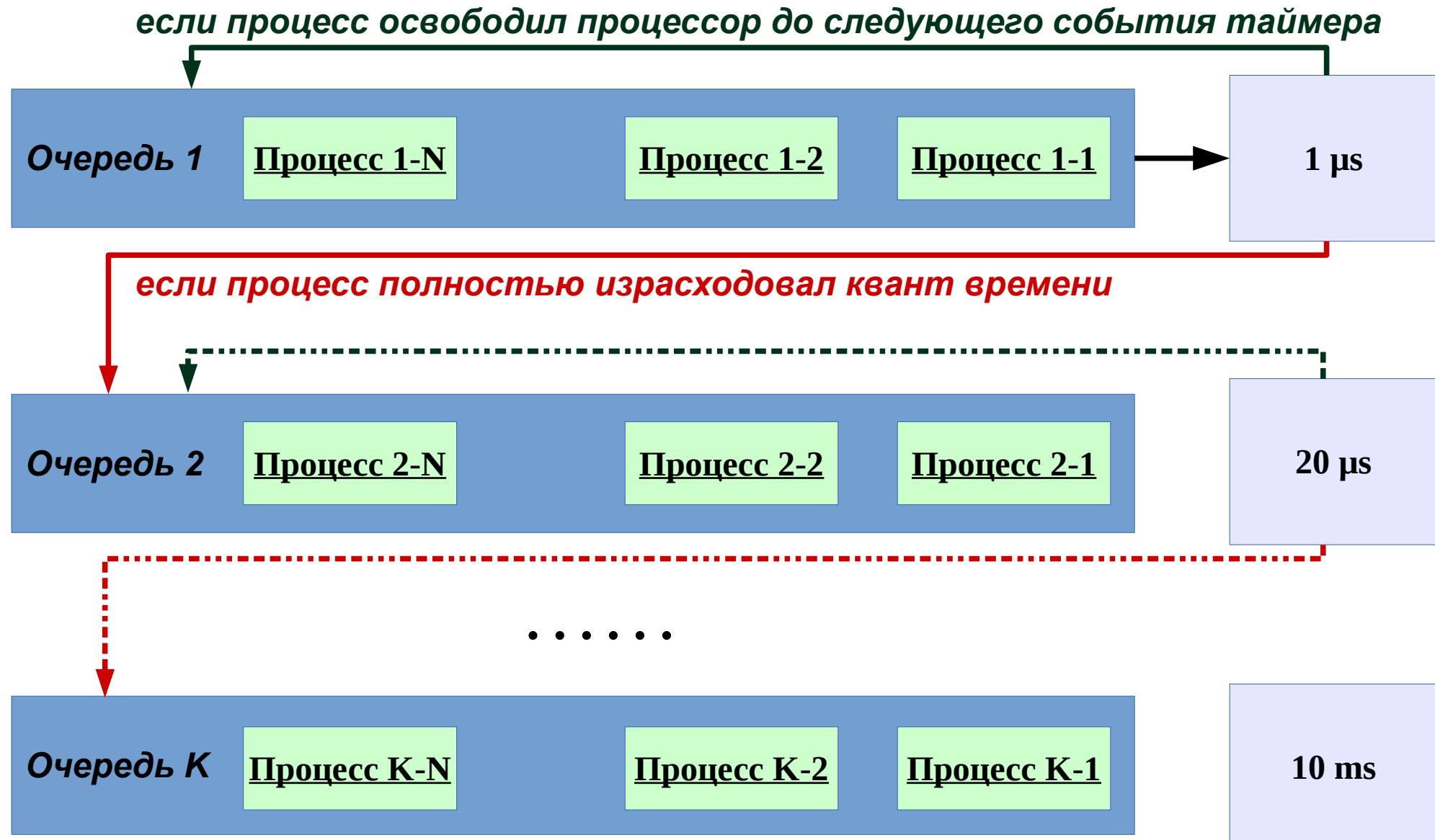


Вытесняющая многозадачность: Round Robin

- Процессы могут иметь приоритет
- В алгоритме Round Robin приоритет процесса – это целое число, которое определяет, сколько раз нужно пропустить выбор процесса планировщиком



Вытесняющая многозадачность: Multilevel Queue



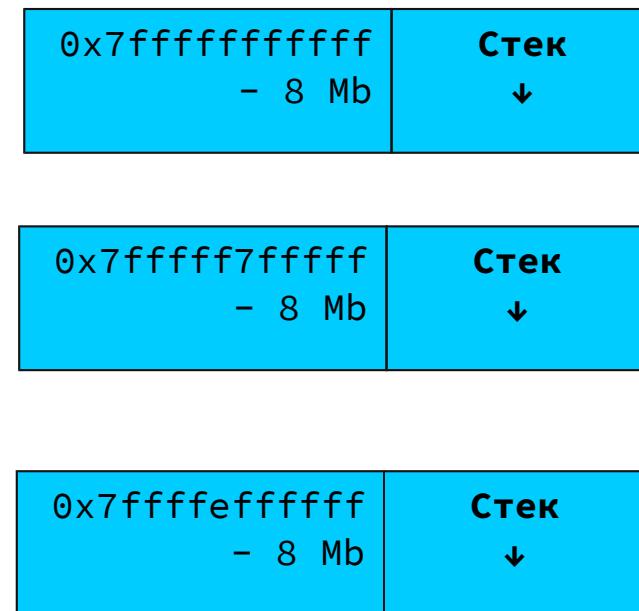
- Для каждой из очередей можно применять свой собственный алгоритм планирования задач
- Приоритет – коротким задачам

Процессы и нити (потоки)

Память процесса:



Память потока:



- **Процесс** – это изолированное адресное пространство + отдельная сущность планировщика задачий
- **Нить (системная)** – это отдельная сущность планировщика задачий

POSIX

```
#include <pthread.h>
void* worker(void* arg) {
    int* res = malloc(sizeof(int));
    *res = 2*(*(int*)arg); return res;
}
main() {
    pthread_t thr;
    /* Запуск функции worker
       в отдельном потоке */
    int arg = 123;
    pthread_create(&thr, NULL,
                  &worker, &arg);
    . . . . /* не блокируемый код */
    /* Ждать до завершения потока */
    int* res;
    pthread_join(&thr, &res);
    printf("Worker result: %d\n", *res);
    free(res);
}
```

C++ 11

```
#include <thread>
void worker(int in, int &out) {
    out = 2 * in;
}
int main() {
    /* Запуск функции worker
       в отдельном потоке */
    int arg = 123;
    int res;
    std::thread thr(worker, arg,
                    std::ref(res));
    . . . . /* не блокируемый код */
    /* Ждать до завершения потока */
    thr.join();
    std::cout << "Worker result: "
              << res << "\n";
}
```

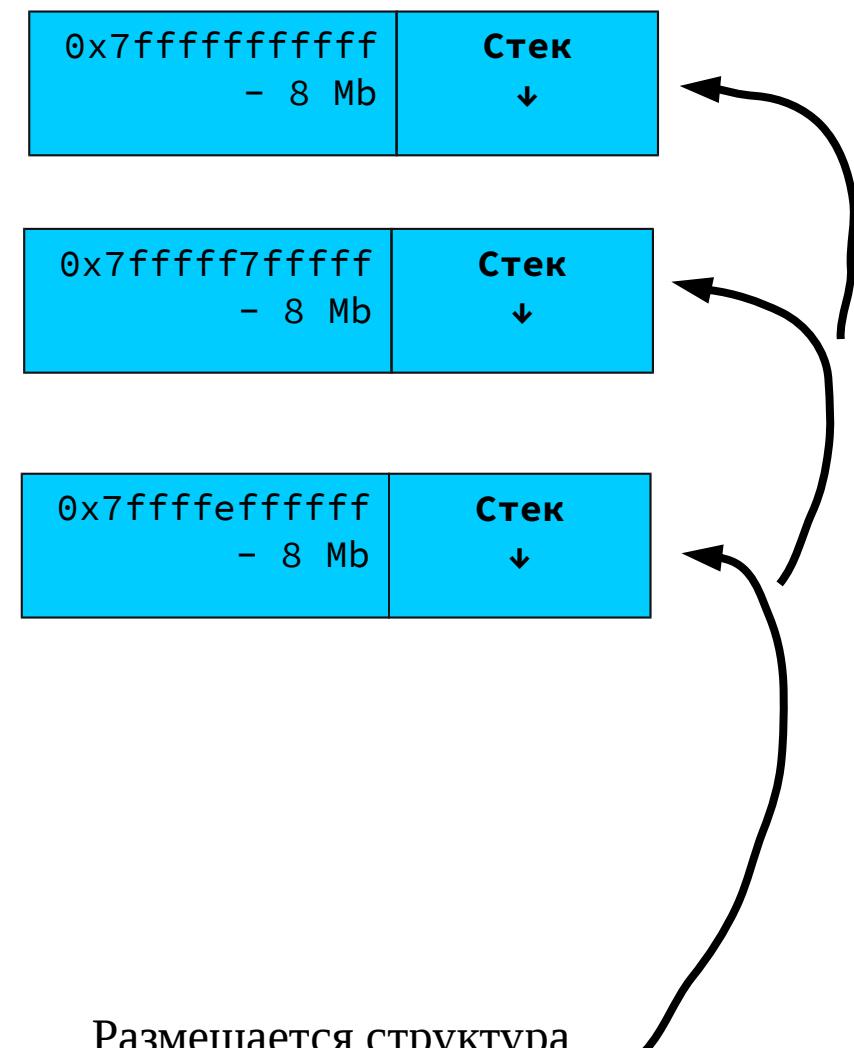
Реализации многопоточности в разных ЯП: Python (модуль `threading`)

Память процесса:



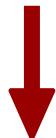
Размещается структура
PyInterpreterState

Память потока:



Размещается структура
PyThreadState

- Запуск потока - это `pthread_create` + инициализация структуры `PyThreadState`
- Каждый поток имеет свой стек вызовов
- Используется один экземпляр интерпретатора `PyInterpreterState` на все потоки



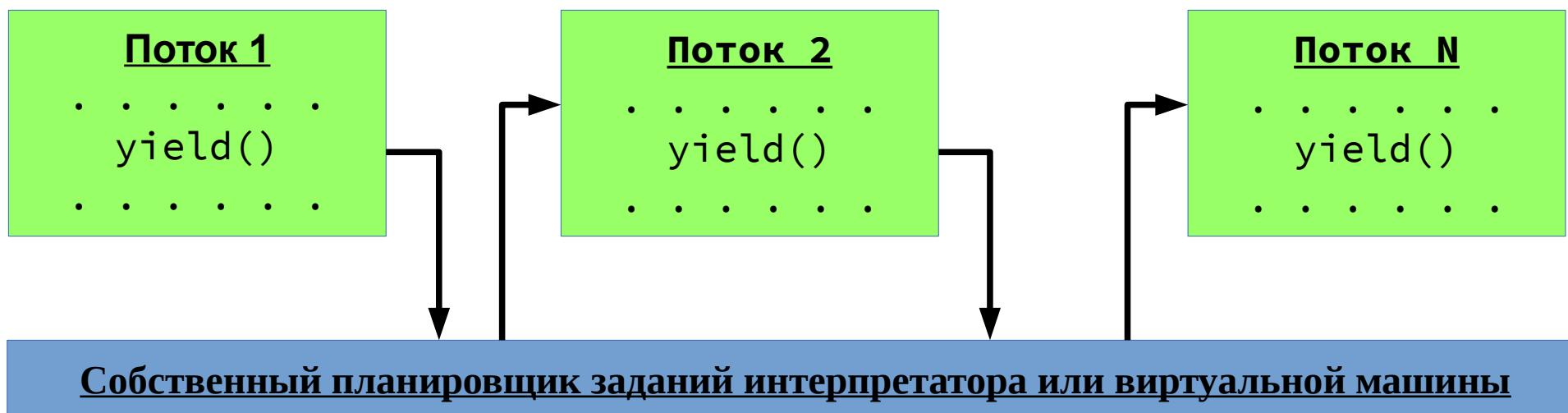
Global Interpreter Lock (GIL)

- Одновременно возможно выполнение только одного Python-потока, даже на многоядерном процессоре
- Исключение - вызов некоторых функций из Python/C-модулей

```
Py_BEGIN_ALLOW_THREADS // освобождение GIL
.... // какой-то Си-код, не использующий Py API
Py_END_ALLOW_THREADS // захват GIL
```

“Зеленые” нити – кооперативная многозадачность

- Виртуальная машина гарантирует, что каждая нить будет периодически приостанавливать свою работу
- Каждой нити выделяется ровно столько памяти, сколько ей требуется



- Каждому потоку Java соответствует нить ОС
- Кроме основного потока в JVM (1.8) выполняются:
 - Потоки Garbage Collector'a
 - **Finalizer**
 - **Reference Handler**
 - **Signal Dispatcher** - обработчик сигналов ОС
 - **Compiler** - компилятор JIT
 - **Periodic Task** - обработчик TimerEvent
- С потоком связаны два стека - Java Stack и Native Stack
- Green Threads - через стороннюю библиотеку, которая модифицирует байт-код
<https://github.com/puniverse/quasar>

- Компилятор вставляет `yield`-инструкции
- Реализуется модель $M:N$, где
 M – количество системных процессов,
 N – количество нитей

Семафоры и мьютексы (на примере Qt API)

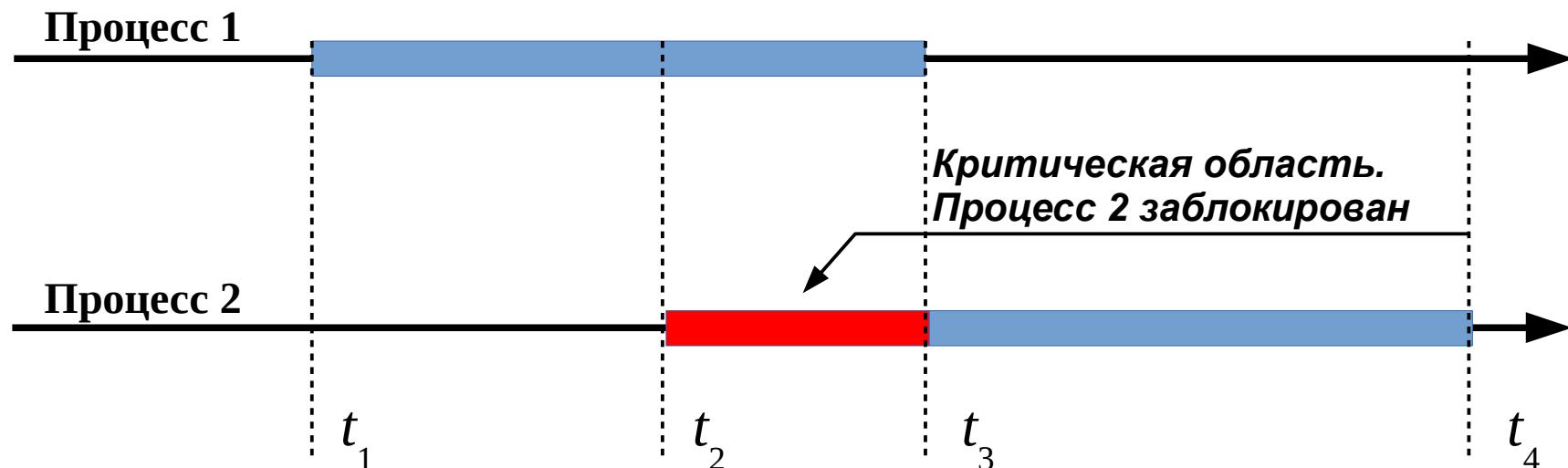
Семафор

```
QSemaphore sem(5); // N = 5  
  
sem.acquire(3); // N = 2; OK  
  
sem.release(3); // N = 5  
  
sem.acquire(4); // N = 1; OK  
  
sem.acquire(1); // N = 0; WAIT!
```

Мьютекс

```
QMutex mx; // state = unlocked  
  
mx.lock(); // locked once; OK  
  
mx.unlock(); // unlocked  
  
mx.lock(); // locked once; OK  
  
mx.lock(); // locked twice; WAIT!
```

Синхронизация процессов



S: Семафор;

```
Процесс 1 {  
    . . . .  
    . . . .  
    P1(S); // t1  
    . . . .  
    V2(S); // t3  
    . . . .  
}
```

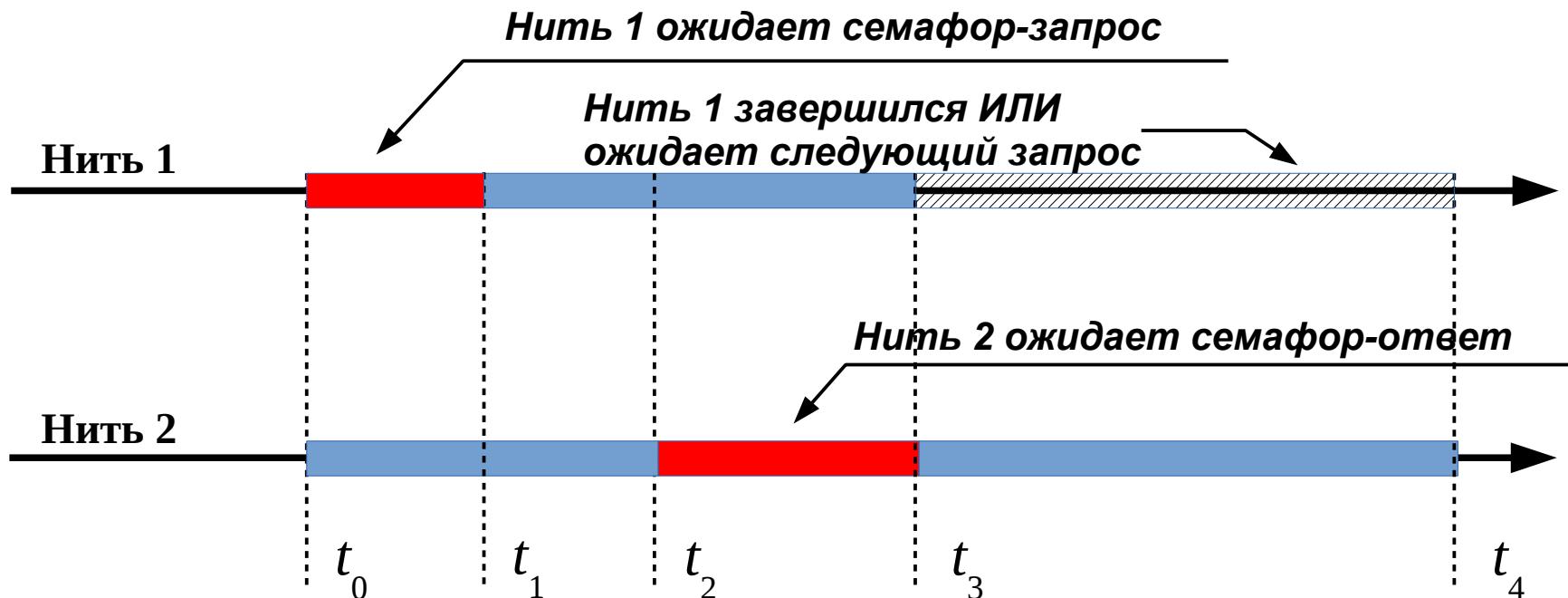
S: Семафор;

```
Процесс 2 {  
    . . . .  
    . . . .  
    P(S); // t2  
    . . . . // t3  
    V(S); // t4  
    . . . .  
}
```

1) Proberen (дат.) – пробовать; пытаться; испытывать; подвергать испытанию; делать опыты; отведывать

2) Verhogen (дат.) – возвышать; повышать; поднимать; увеличивать; усиливать; переводить в высший класс; увеличить

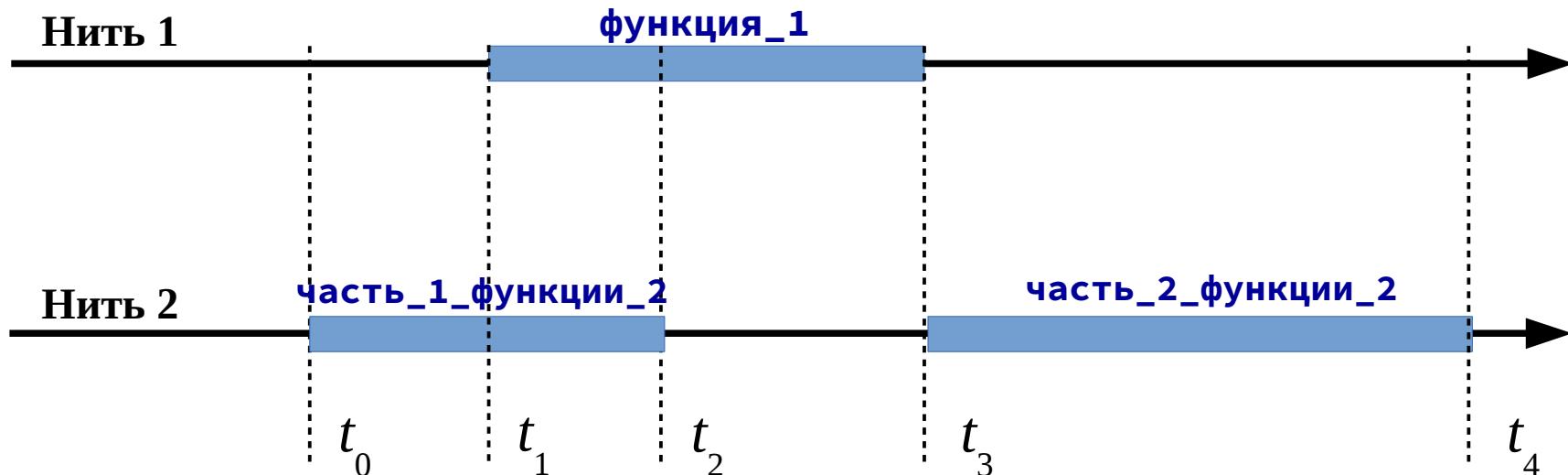
Взаимодействие с семафорами



```
функция_1() {
    P(семафор_запроса);           // t0
    request = данные_запроса;      // t1
    ...
    response = . . .;
    данные_ответа = response;
    V(семафор_ответа);           // t3
    /* maybe */ функция_1();
}
```

```
функция_2() {
    request = . . .                // t0
    данные_запроса = request;
    V(семафор_запроса);           // t1
    ...
    P(семафор_ответа);           // t2
    response = данные_ответа;     // t3
    ...
}
```

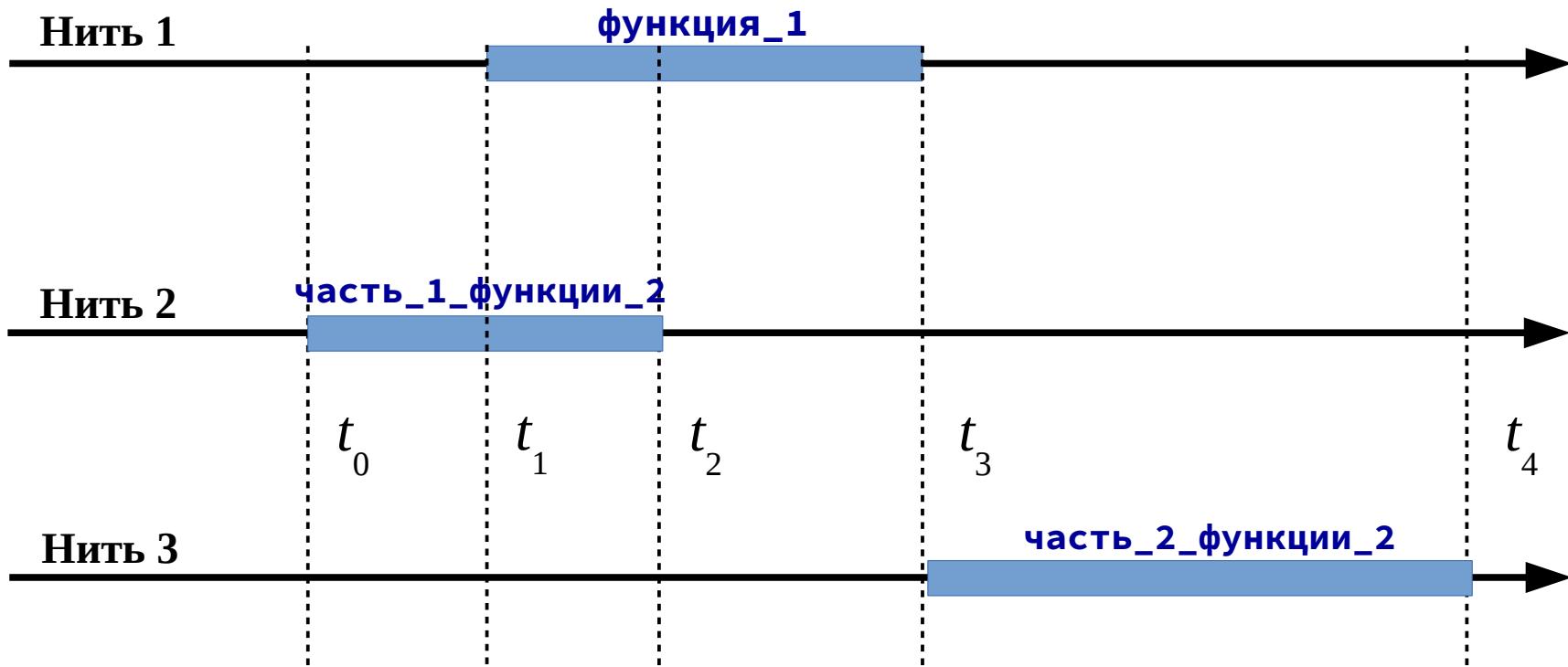
Асинхронное взаимодействие через сообщения



```
событие_запроса = функция_1() {  
    request = данные_запроса; // t1  
    . . . // t2  
    response = . . .;  
    данные_ответа = response;  
  
    данные_ответа -> событие_ответа // t3  
}
```

```
часть_1_функции_2() {  
    request = . . . // t0  
    данные_запроса = request;  
    данные_запроса -> событие_запроса // t1  
    . . .  
    еще_какие_то_действия; // t1...t2  
}  
  
событие_ответа = часть_2_функции_2() {  
    response = данные_ответа; // t3  
    . . .  
}
```

Асинхронное взаимодействие через сообщения



Асинхронный обмен сообщениями позволяет масштабировать программу на большее количество потоков выполнения

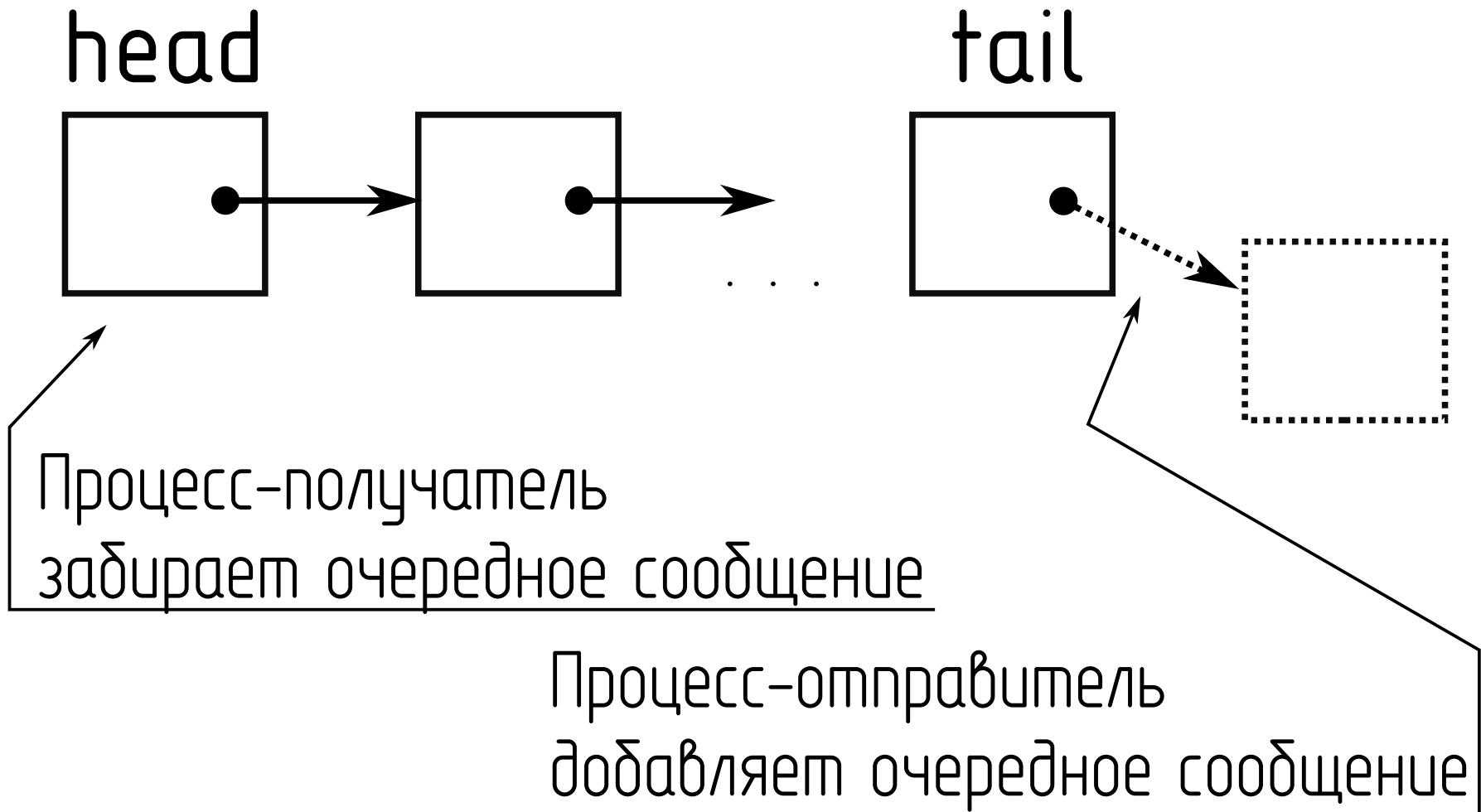
Общая память

- отсутствие накладных расходов на механизм передачи сообщений
- возможность использования не параллельных программ с минимальными изменениями

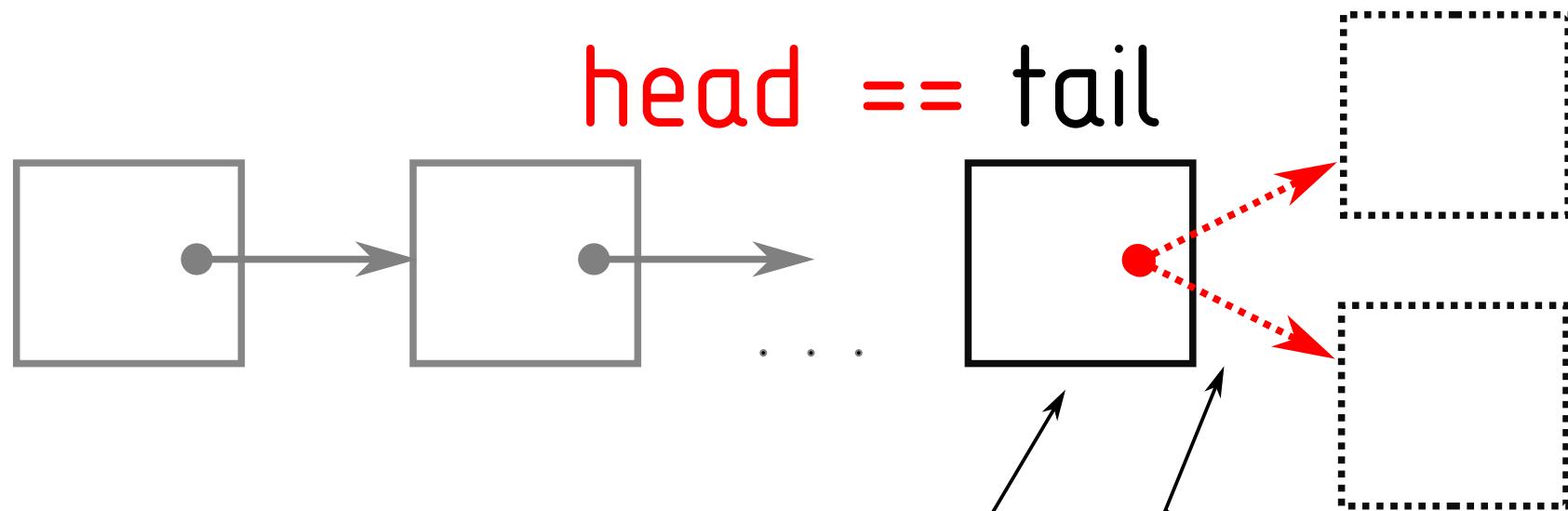
Обмен сообщениями

- маловероятные проблемы с синхронизацией
- неограниченные возможности масштабирования

Очередь сообщений: FIFO



Очередь сообщений: FIFO



Процесс-получатель забирает очередное сообщение

Два процесса-отправителя добавляют очередные сообщения

Варианты решения проблемы:

- Блокировки отдельных узлов
 - Неблокирующие реализации FIFO

- **Объекты синхронизации (связаны с состоянием процесса/нити):**
 - мьютексы
 - семафоры
 - условные переменные
- **Операции на уровне процессора (непрерывная попытка изменения до успеха):**
 - Compare-And-Swap (x86, x86_64)
 - Load-Link / Conditional-Store (ARM)

Атомарные операции

CAS (Compare-And-Swap):

- Для Intel x86 реализуется специальной командой

cmpxchg

[Примерно] эквивалентная реализация на C++:

```
bool compare_and_swap(int* addr, int* old_val, int new_val) {
    bool result = false;
    if (*addr == *old_val) {
        *addr = new_val;
        result = true;
    }
    else {
        *old_val = *addr;
    }
    return result;
}
```

Атомарные операции

CAS (Compare-And-Swap):

- Для Intel x86 реализуется специальной командой

cmpxchg

Пример [x86 (>=i486)]:

```
bool compare_and_swap(int* addr, int* old_val, int new_val) {  
    movl    (old_val), %eax  
    movl    new_val, %ebx  
    cmpxchgl %ebx, (addr)  
    jz     .success  
    movl    %eax, (old_val)  
    movl    $0, %eax  
    jmp     .done  
success: movl    $1, %eax  
done:  
}
```

Практические реализации

- <http://libcds.sourceforge.net>
- boost::lockfree

Теоретические исследования

- M.Michael, M.Scott ***Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.*** Proc. 15th Annual ACM Symp. on Principles of Distributed Computing (PODC). pp. 267–275.
- P. Tsigas, and Y. Zhang ***A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems.*** Proc. of the 13th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA'01), pp. 134-143, July 2001.
- C.Evequoz ***Non-Blocking Concurrent FIFO Queues With Single Word Synchronization Primitives.*** 13th Ada-Europe International Conference on Reliable Software Technologies, Venice, Italy, June 16-20, 2008. pp. 59-72

Дополнительная литература

- Воеводин В.В., Воеводин Вл.В. *Параллельные вычисления*. Спб.:БХВ, 2002. 602 С.
- Энтони Уильямс. *Параллельное программирование на C++ в действии. Практика разработки многопоточных программ*. М.:ДМК-Пресс, 2012. 672 С.
- Таненбаум Э., Бос Х. *Современные операционные системы. 4-е изд.* Спб.:Питер, 2015. 1120 С.
- Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. — Pragmatic Bookshelf, 2007. — 536 р.