



Пролог

3 вечных вопроса

Что делать?

~~Кто виноват?~~

Пролог

3 вечных вопроса

Что делать?

**~~Кто виноват?~~ Почему так
получилось?**

Как сделать?

4x10G NIC (Network Interface Card)



4x10G NIC

Инструменты:

- Netmap — opensource
- pf_ring — proprietary
- DPDK (Data Plane Dev Kit) — opensource

Key features:

- Работа в userspace, минуя kernel
- DMA, no copy

4x10G NIC

Пакет = 512 байт = 2^{**9} байт = 2^{**17} бит

Поток = $40 \times 2^{**30}$ бит

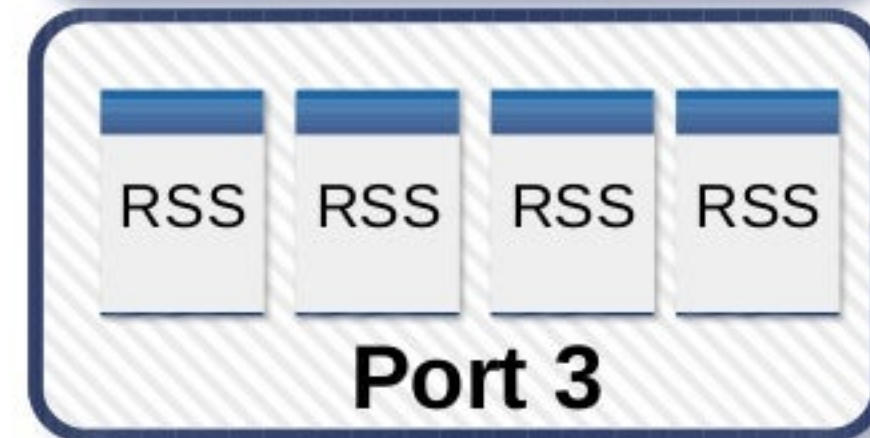
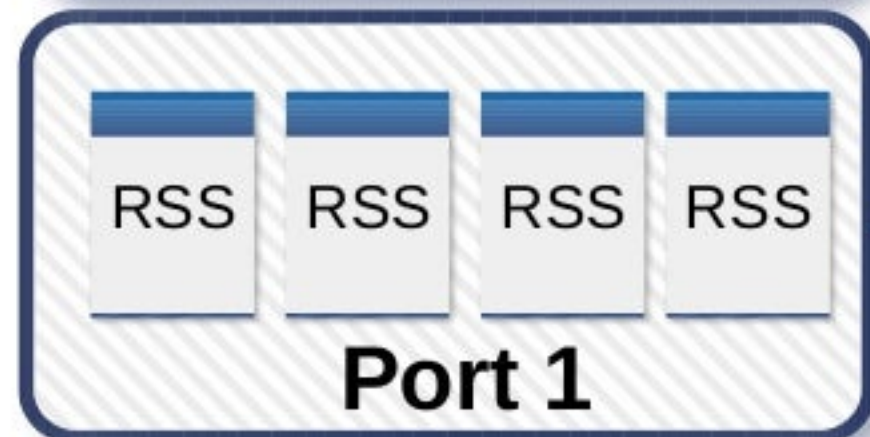
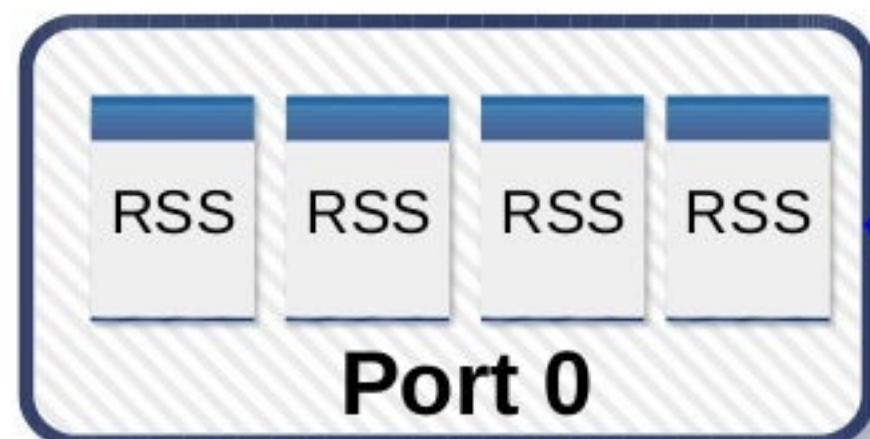
Итого: $40 \times 2^{**30} / 2^{**17} = 40 \times 2^{**13}$ пакетов/сек

Процессор = 4 ГГц $\sim 4 \times 2^{**30}$

На пакет: $4 \times 2^{**30} / 40 \times 2^{**13} = 0.1 \times 2^{**17}$

$\sim 2^{**14} = 16K$ тактов на пакет

NIC



Hash

Application

Thread 0

Thread 1

Thread 2

Thread 3

Thread 4

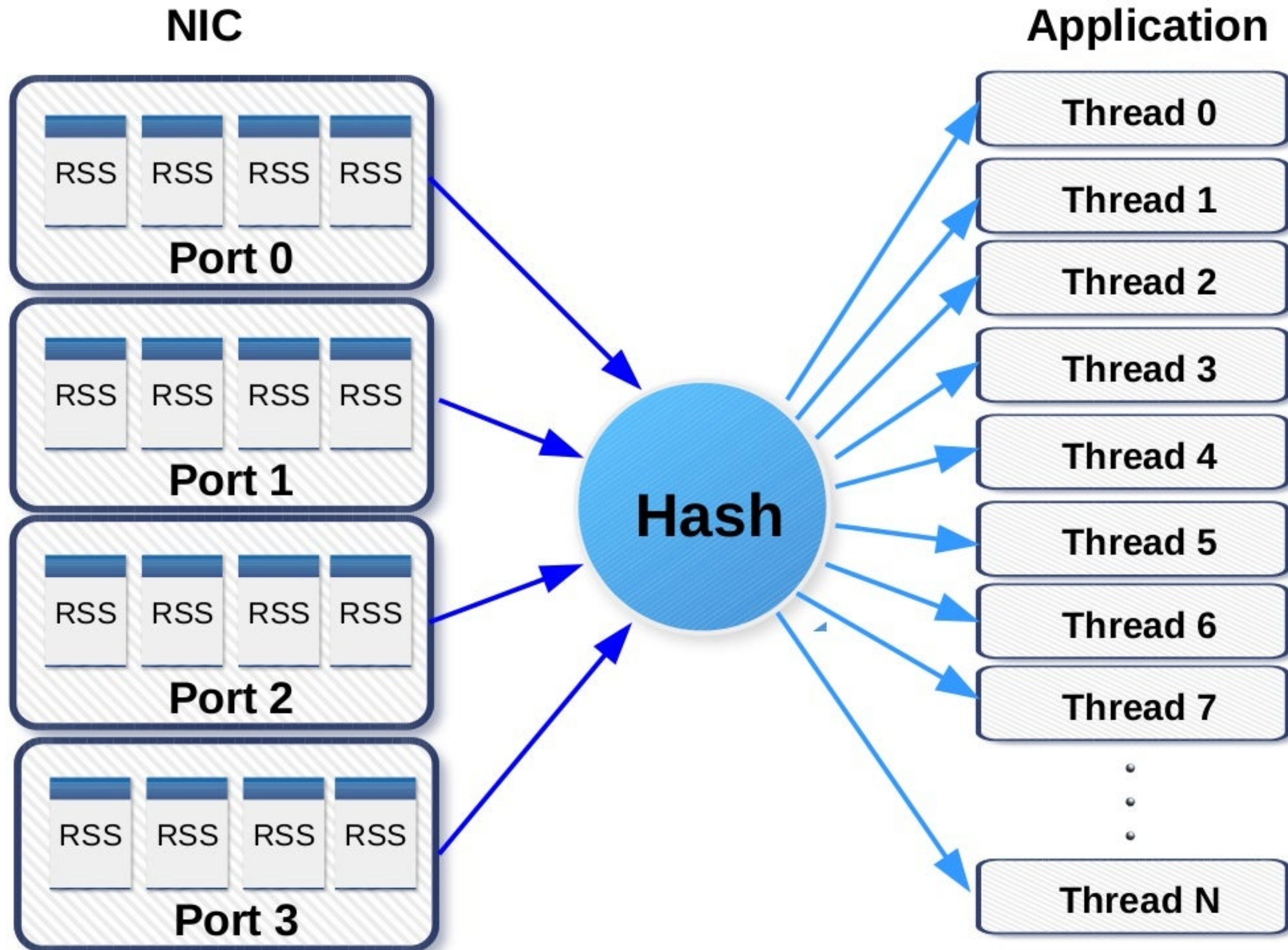
Thread 5

Thread 6

Thread 7

⋮

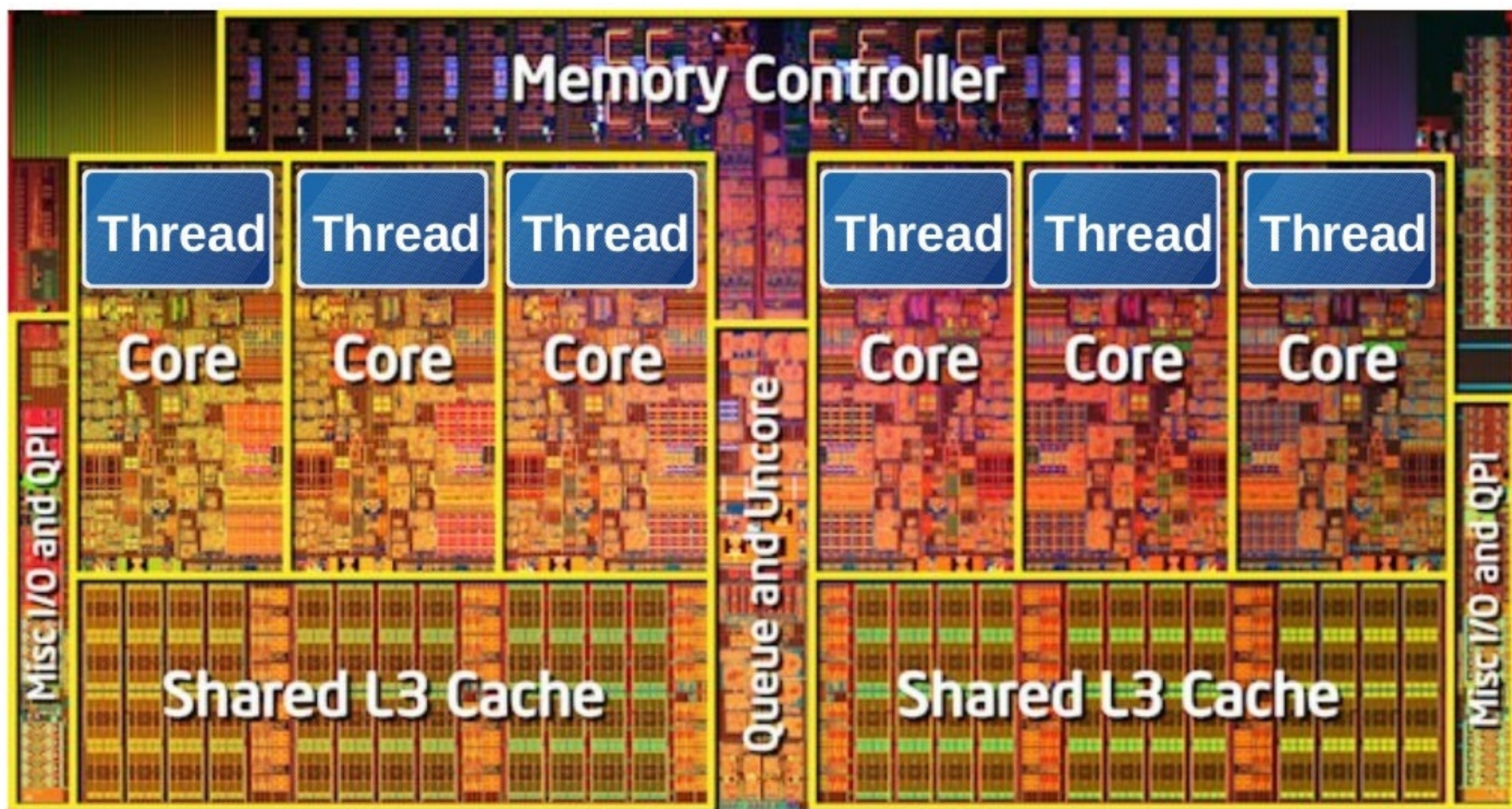
Thread N



Порочные практики

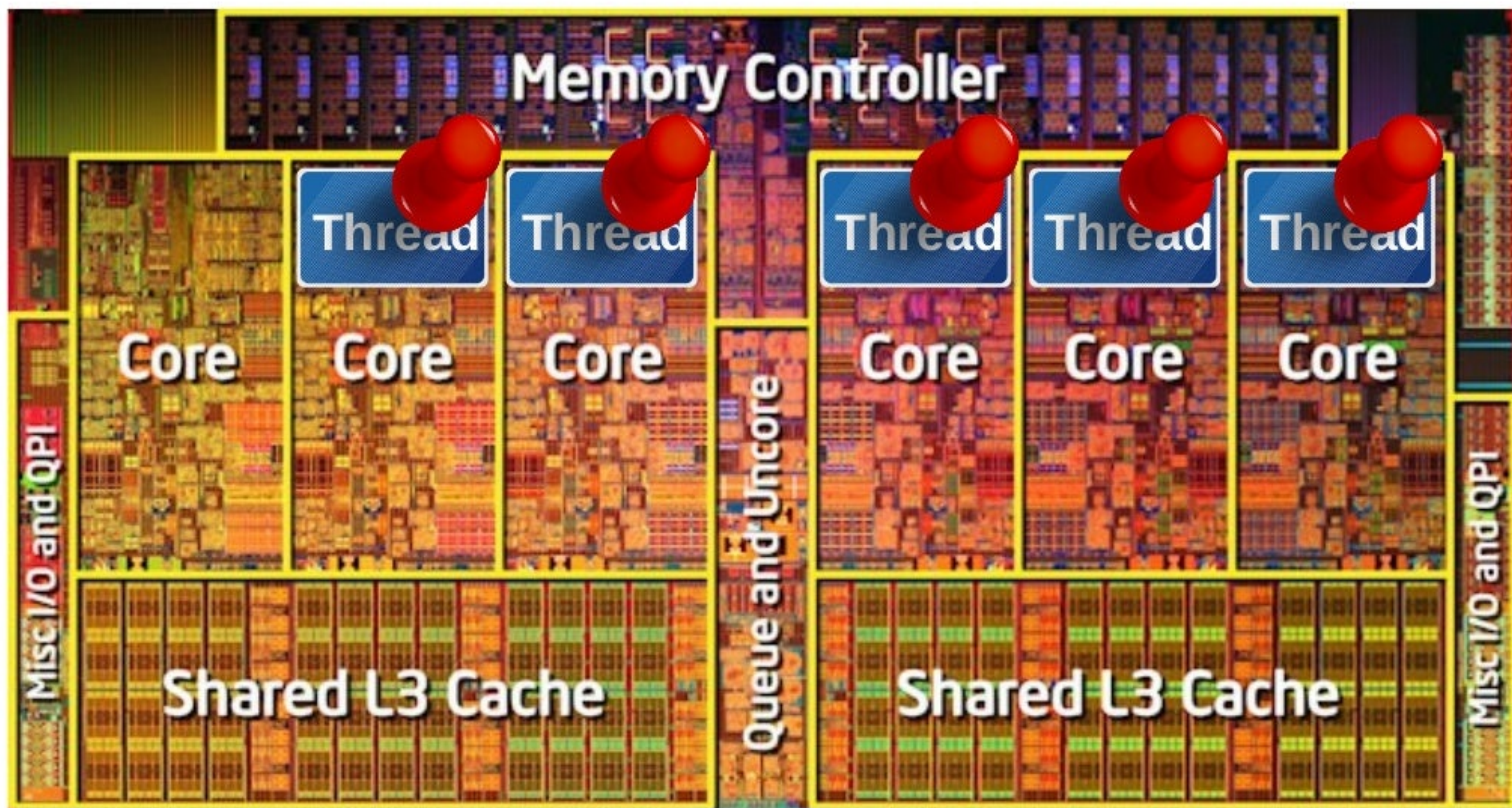
- ✗ Безумное количество потоков — потоков много больше, чем ядер
- ✗ Создание потоков «на лету», под выполнение конкретного действия

Распараллеливание



Число потоков \leq число ядер

Распараллеливание



Core 0 — свободен (для OS)
Hard pinning threads to cores

Worker thread

Характеристики:

- No memory allocation
- Hard pinned to processor core
- No memcry (по возможности)



Worker thread

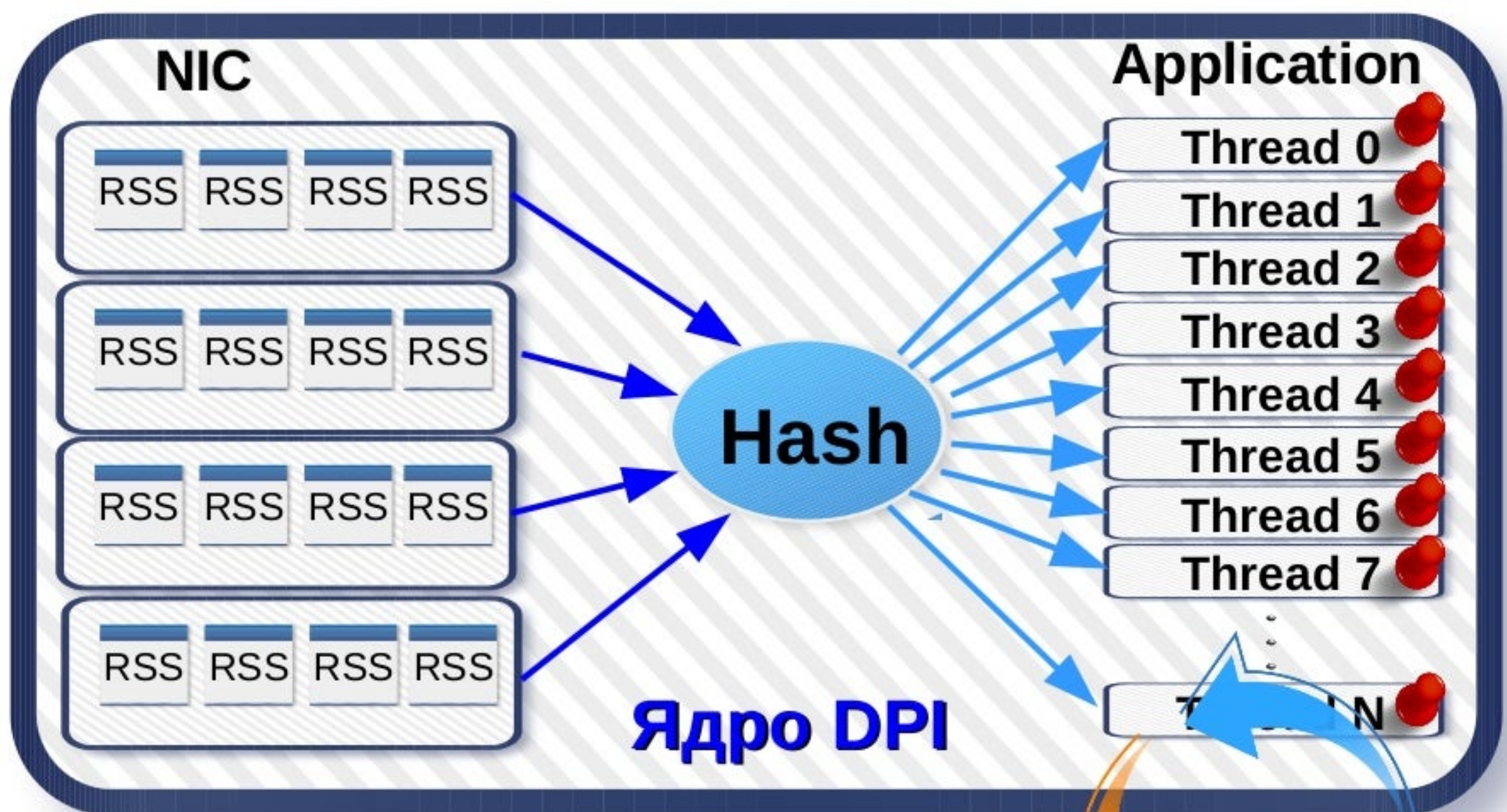


Характеристики:

- No memory allocation (= no std containers)
- Hard pinned to processor core
- No memcry (по возможности)

Проблемы:

- Длительные/отложенные действия (авторизация абонента)
- Периодические задачи (тайм-ауты)
- Мониторинг внутреннего состояния
- Взаимодействие со сторонними системами (DHCP, Radius, ...)



Apartment model
Служебные потоки:
Мониторинг, отложенные действия, взаимодействие
со сторонними системами, ...

Apartment model - концепция


- Поток (thread) — низкоуровневая абстракция, не связанная с предметной областью
- Понятия «управление потоками» не должно быть в принципе
- ИТС (inter-thread communication) — глубоко спрятано. Вызываем member function, где и как она выполняется — определяет владелец

Apartment model - концепция

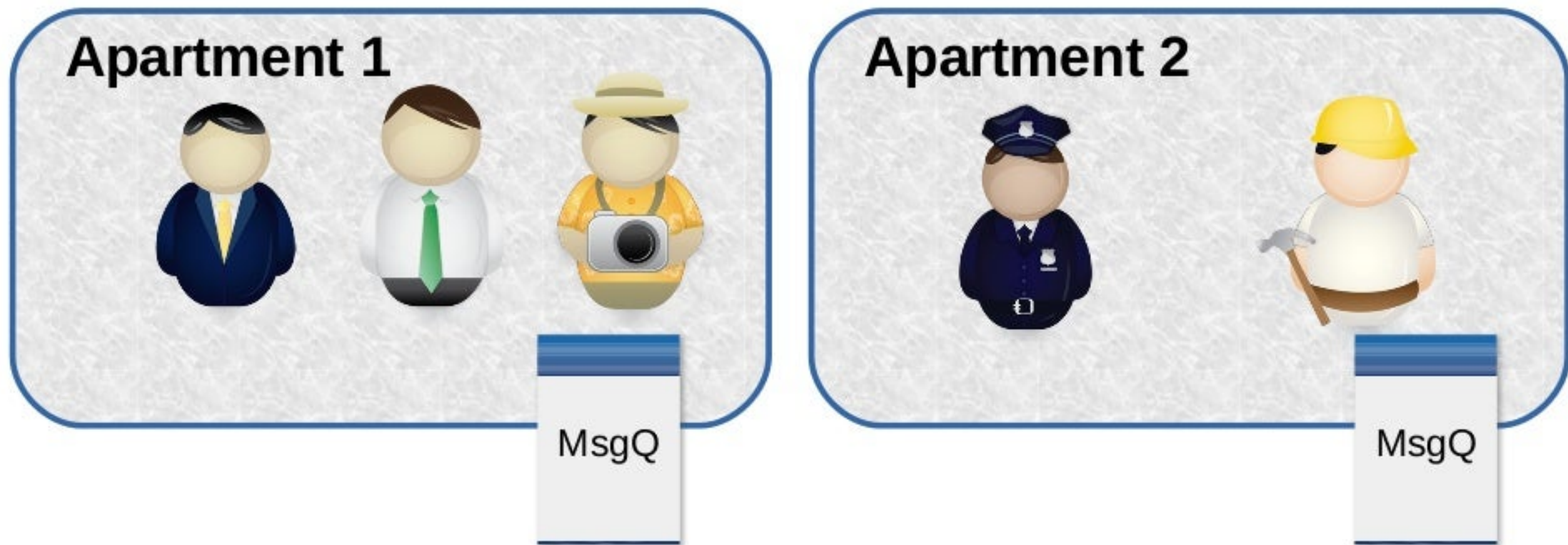
Apartment == thread

*Apartment — замкнутая система
Взаимодействие — только через
MPSC-очередь сообщений*

Msg queue

A vertical queue structure with 10 horizontal slots. The top slot is labeled 'Msg queue'. The queue is outlined in red. A speech bubble from the text box above points to the queue.

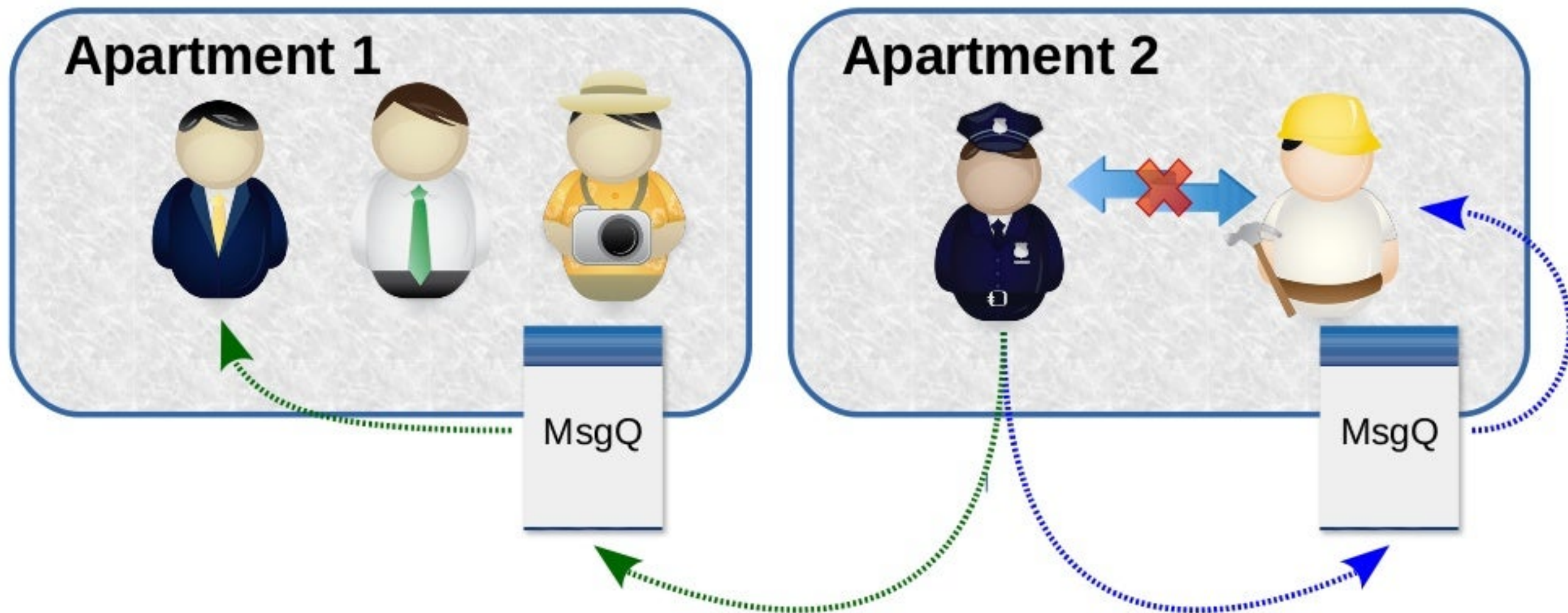
Apartment model - концепция



Апартаменты заселяются компонентами 

Заселение - на этапе старта, положение компонента задается в conf-файле

Apartment model - концепция



Каждый компонент — **single threaded**

Общение между компонентами — **async call**

Компоненты не знают о расположении друг друга

Apartment model

```
class Component {
public:
    Component( Apartment& ap ): apartment_( ap ) {}

    void foo( int arg1, void* arg2 ) {
        if ( current_apartment() != apartment_ )
            apartment_.invoke( this, foo, arg1, arg2 );
        else {
            // нахожусь в своем апартаменте
            // выполняем свою работу
        }
    }

    // вызов – всегда асинхронный
    void bar() {
        apartment_.invoke( this, do_bar );
    }

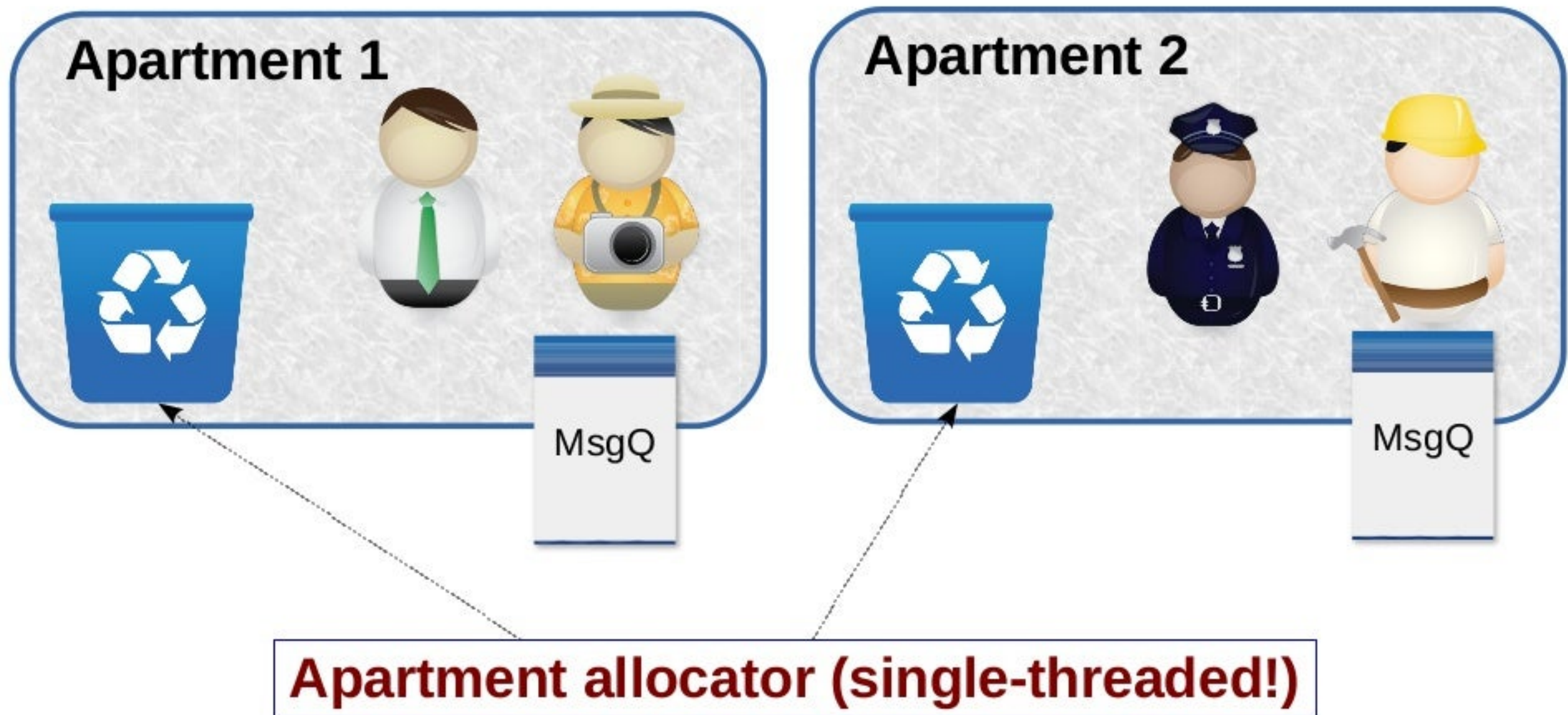
private:
    void do_bar() { ... }

private:
    Apartment& apartment_; // в какой апартамент заселен
};
```

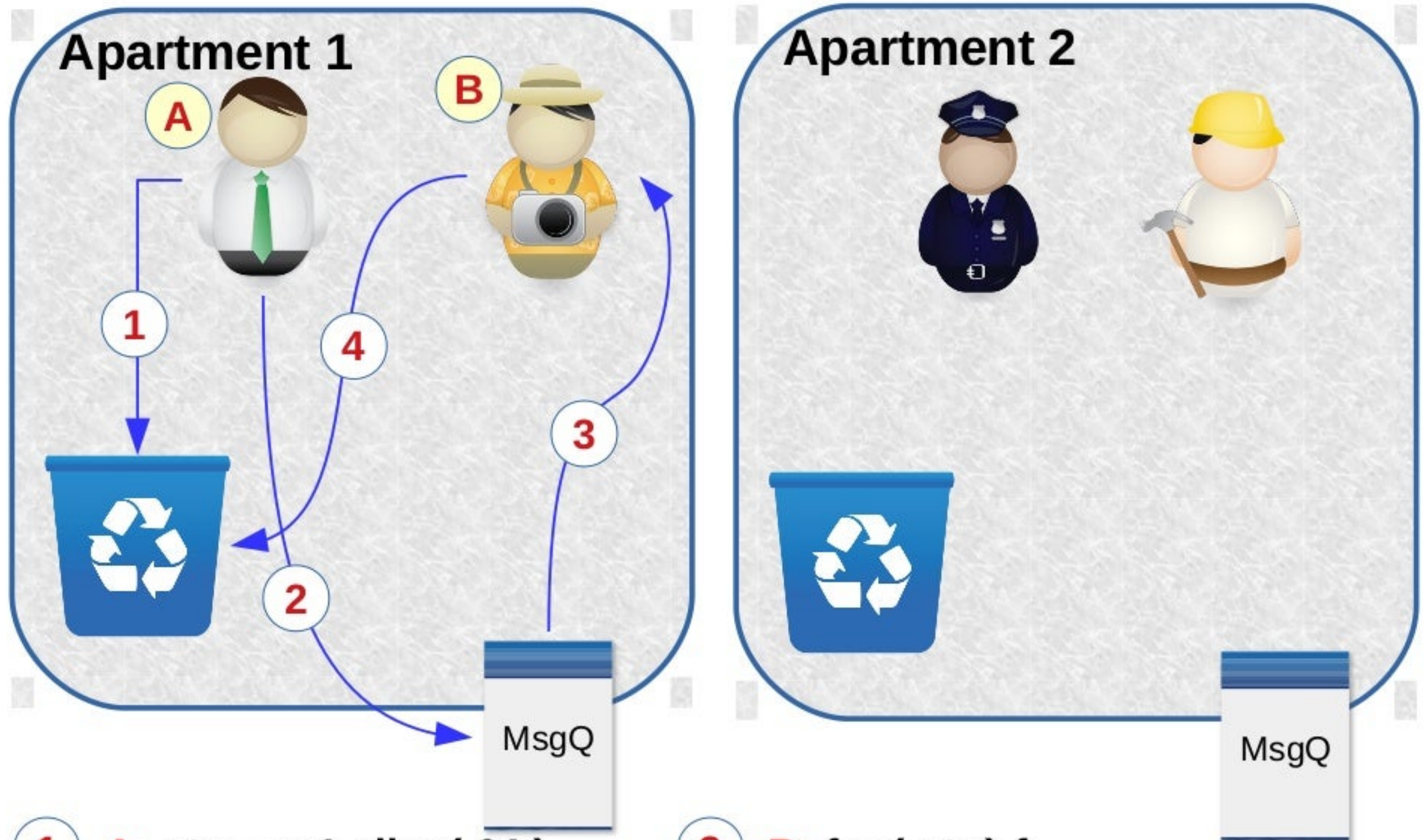
Apartment model - сервисы

Компоненты хотят:

- ✓ Распределять память
- ✓ Выполнять периодические или отложенные действия
- ✓ Следить за внешними событиями (сетевое взаимодействие)



Apartment model - allocator



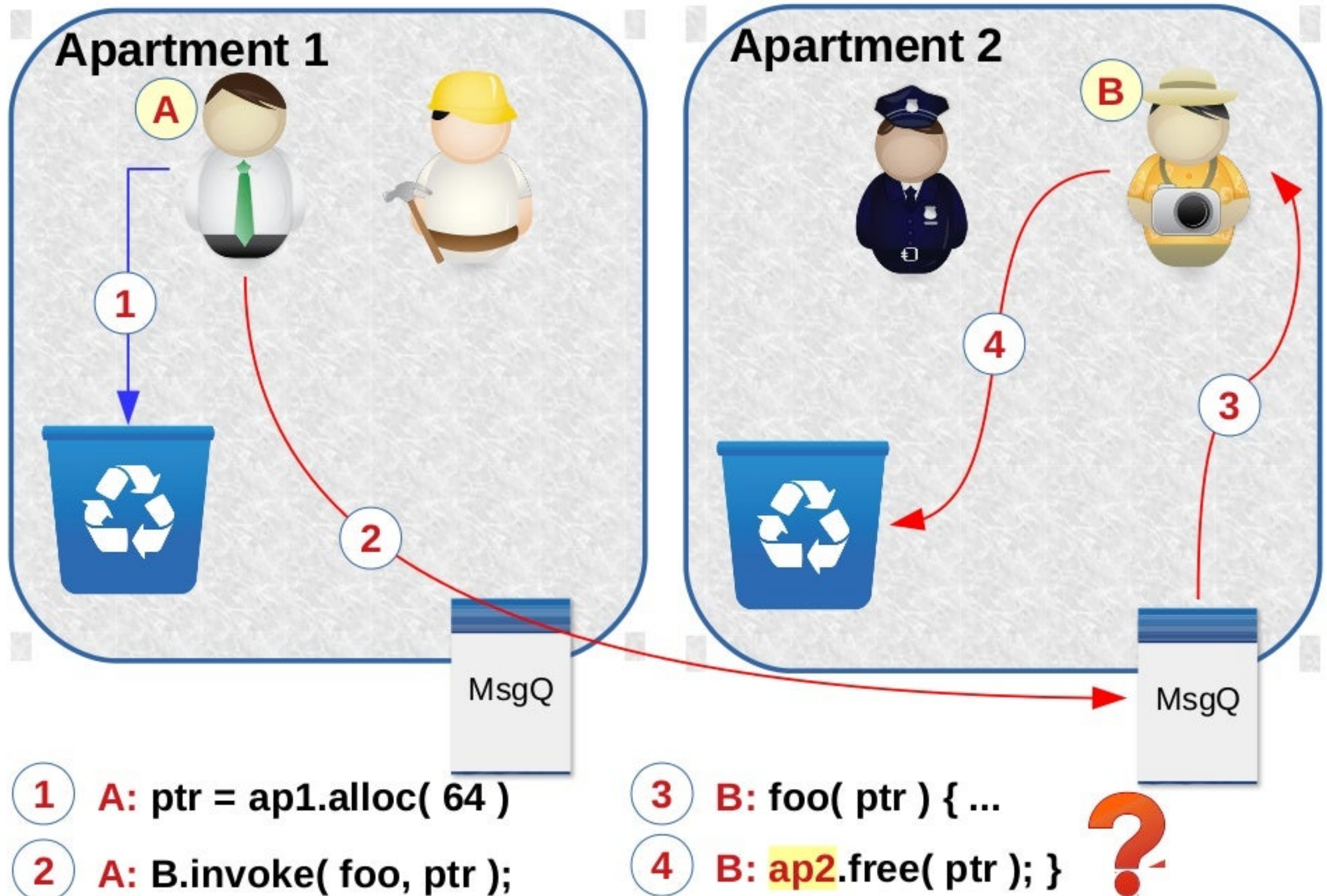
1 **A:** `ptr = ap1.alloc(64)`

2 **A:** `B.invoke(foo, ptr);`

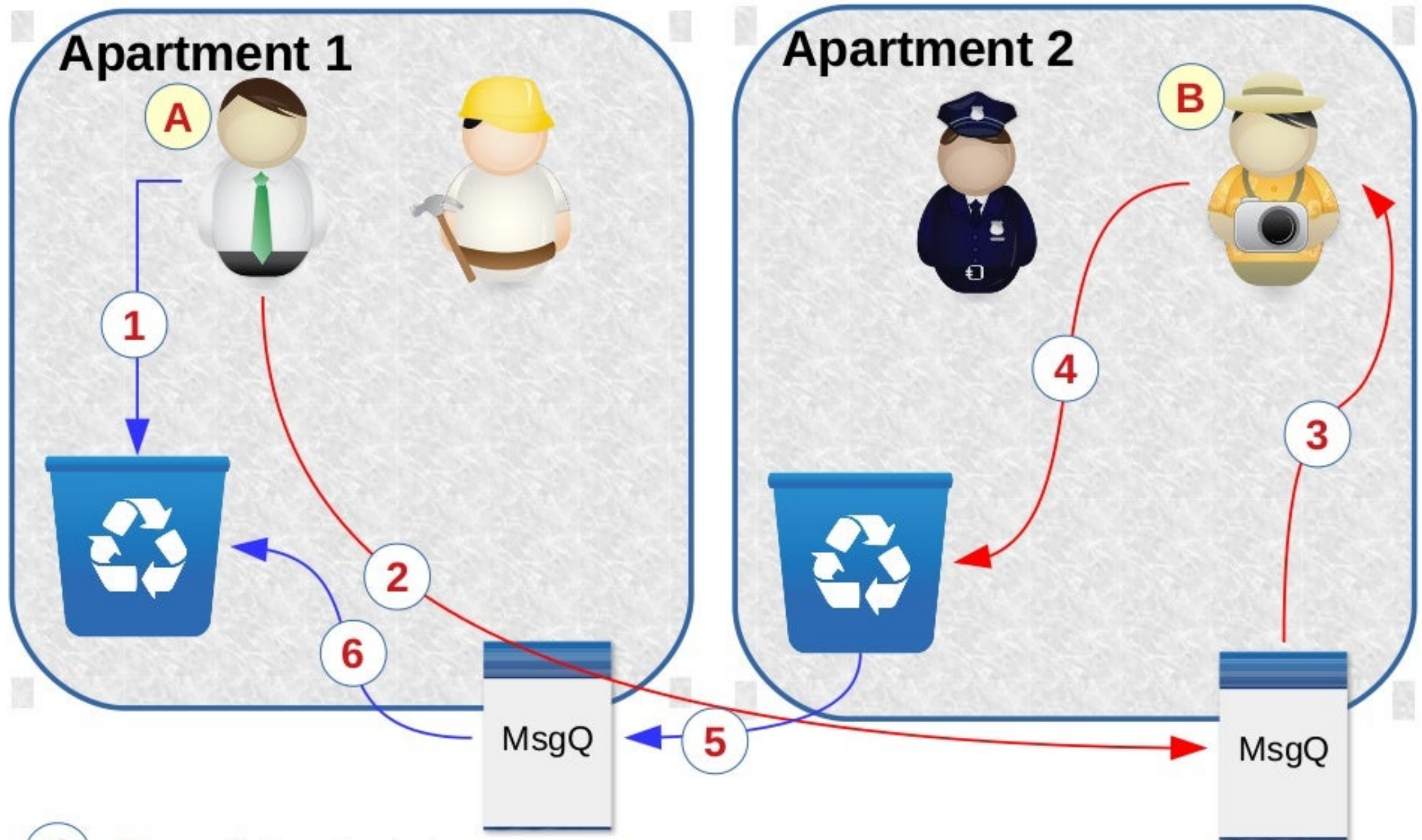
3 **B:** `foo(ptr) { ...`

4 **B:** `ap1.free(ptr); }`

Apartment model - allocator



Apartment model - allocator



4 B: **ap2.free(ptr);**

5 **ap1.invoke(free, ptr);**

6 A: **ap1.free(ptr);**

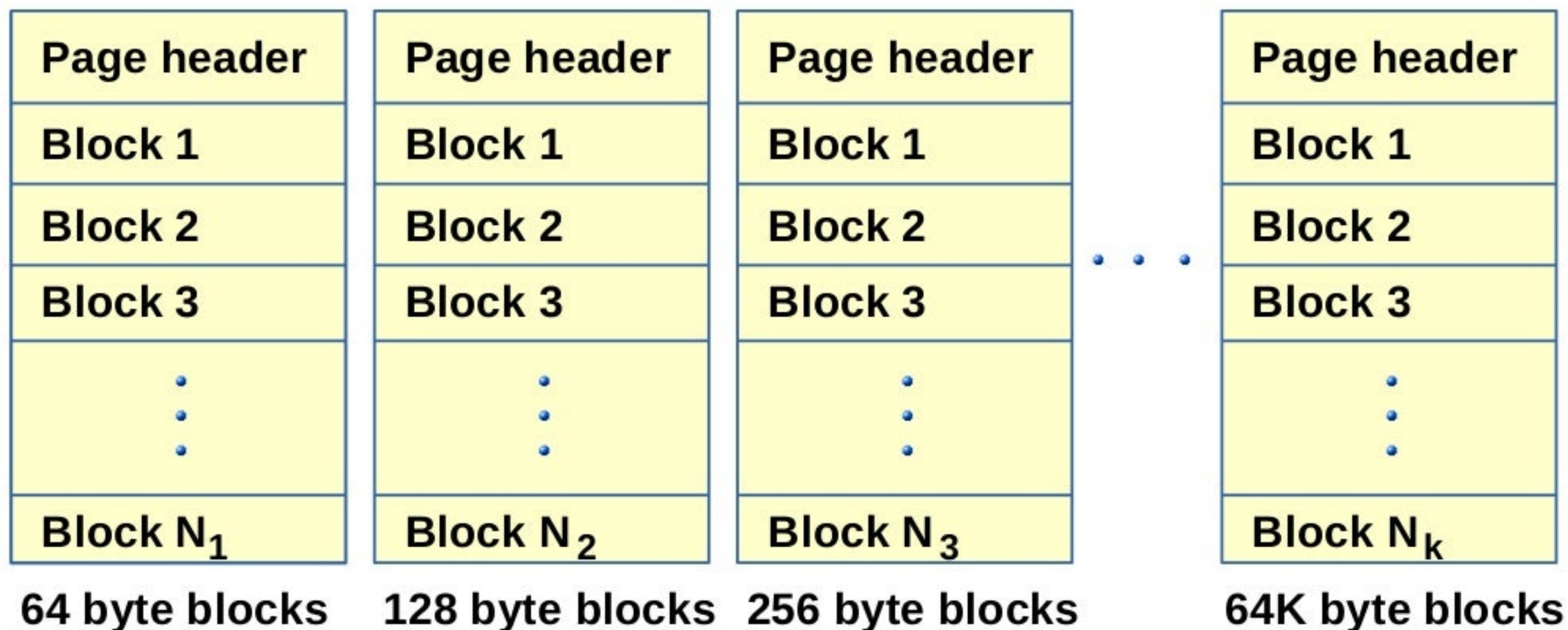
Apartment allocator

Требования:

- Избежать фрагментации / вспенивания
- Для каждого распределенного блока памяти при вызове `free()` нужно **быстро** понять, к какому аллокатору (апартменту) он относится
- Выделенный блок памяти не должен иметь префиксов / управляющих структур

Решение: страничный single-threaded субаллокатор

Apartment allocator



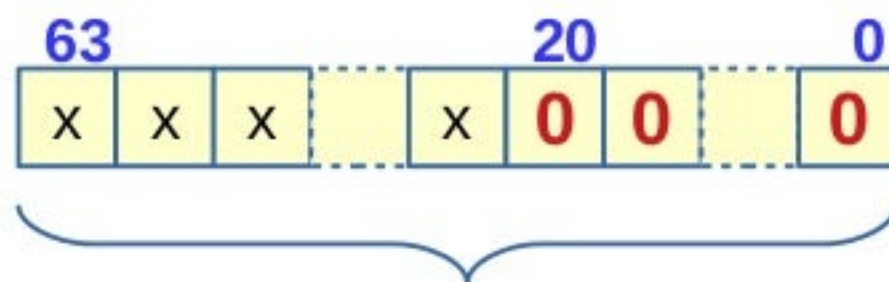
- Размер страницы — фиксированный (2M)
- Для каждой страницы - блоки фиксированного размера
- Выделение M байт:
 - `size_class` = ближайшая сверху к M степень двойки
 - Выделяем свободный блок из страницы для `size_class`

Apartment allocator

```
struct page_header {  
    size_class_desc* psc; // дескриптор size_class  
    apartment* owner;     // апартамент-владелец страницы  
    void* first_free_block; // 1-й свободный блок страницы  
    page_header* next;    // связь в списки для size_class  
};
```

```
struct size_class_desc {  
    apartment* owner;  
    page_header* current_page; // с этой страницы выделяем  
    page_header* part_page_list; // список неполных страниц  
    page_header* full_page_list; // список полных страниц  
};
```

Размер страницы — 2M (2^{21} — Linux huge page), выравненный по 2M



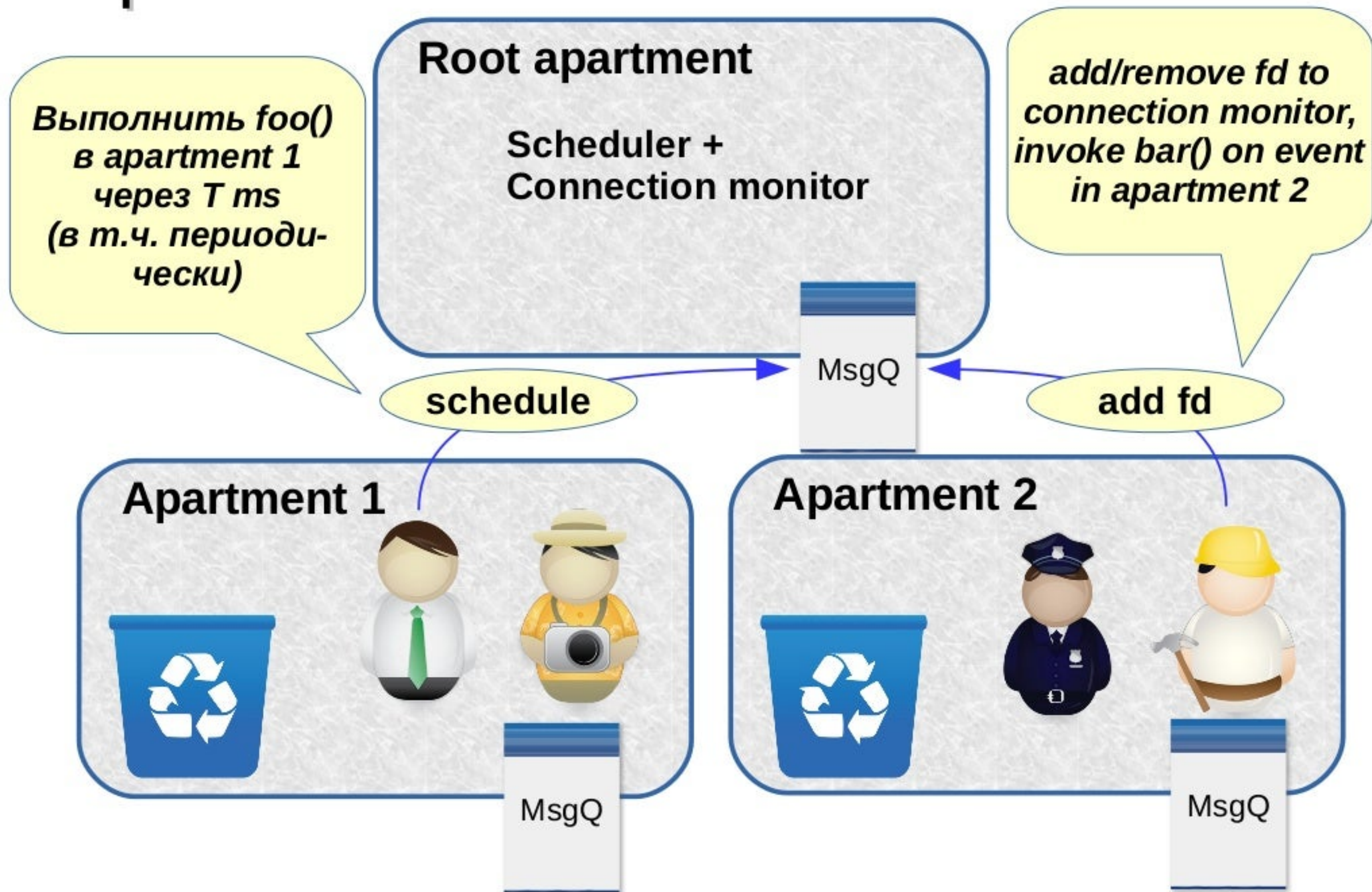
Apartment model - сервисы

Компоненты хотят:

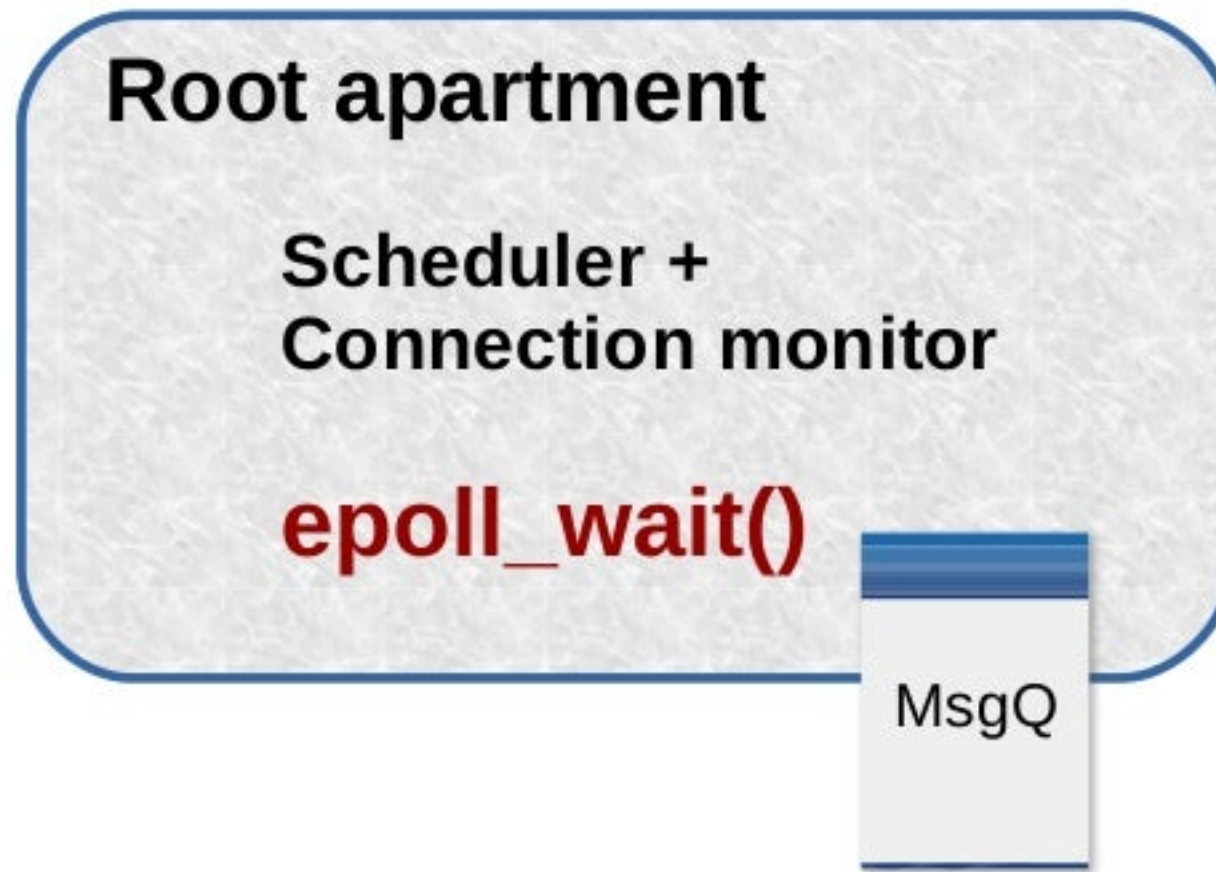
- ✓ **Распределять память - done**
- ✓ **Выполнять периодические или отложенные действия**
- ✓ **Следить за внешними событиями (сетевое взаимодействие)**



Apartment model - scheduler

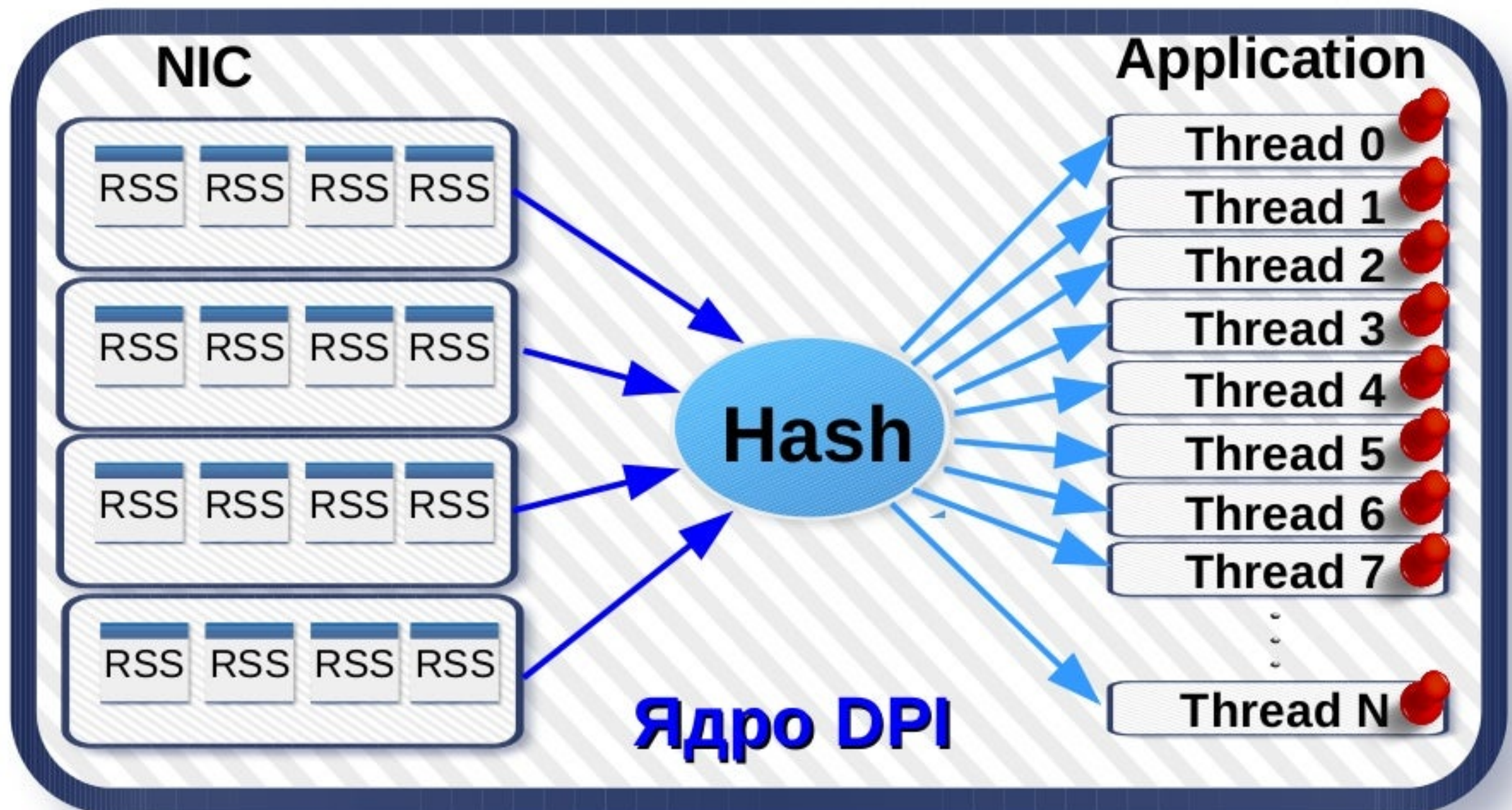


Apartment model - scheduler



```
int epoll_wait(int epfd,  
    struct epoll_event *events, int maxevents,  
    int timeout );
```


Shared data



Hash — не всегда распределит в нужный поток — появляются shared data между worker threads

Persistent data (например, сессии) — создаются в apartment, используются в worker threads

Shared data

По месту создания:

- Worker threads — нет аллокатора, на предраспределенных массивах (pointer = индекс в массиве)
- Apartment — есть аллокатор, истинно динамические контейнеры

Постоянная задача: по возможности вывести создание/удаление shared data в apartment

Shared data

Intrusive (in-place) containers

- std — хранит копии, intrusive — no copy
- Разделено создание данных и операции с контейнером
- Одни и те же данные можно одновременно хранить в разных контейнерах

пример: **boost::intrusive**

Intrusive containers

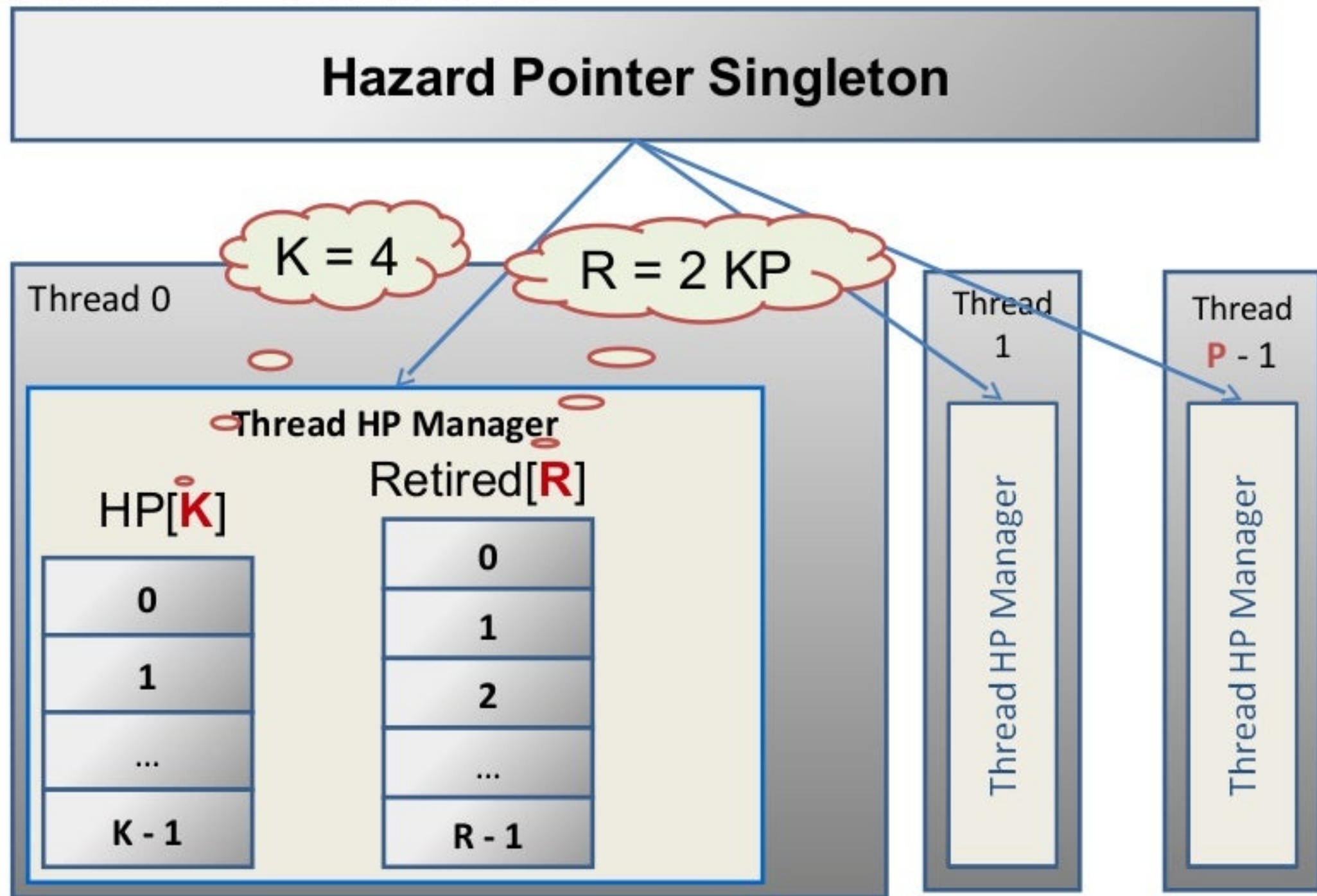
```
template <typename Tag = void>
struct slist_node {
    slist_node<Tag>* next_;
};

template <typename Tag = void>
struct rbtree_node {
    rbtree_node<Tag>* parent_, * left_, * right_;
    int color;
};

struct tag_list1 {};
struct tag_list2 {};

struct my_data: slist_node<tag_list1>,
                slist_node<tag_list2>,
                rbtree_node<>
{
    std::string key;
    // прочие данные
};
```


Hazard Pointers



$$\langle K, P, R \rangle : R > K * P$$

P – thread count

Hazard Pointers

Удаление элемента

```
void hp_retire( T * what ) {  
    push what to current_thread.Retired array  
    if ( current_thread.Retired is full )  
        hp.Scan( current_thread );  
}
```

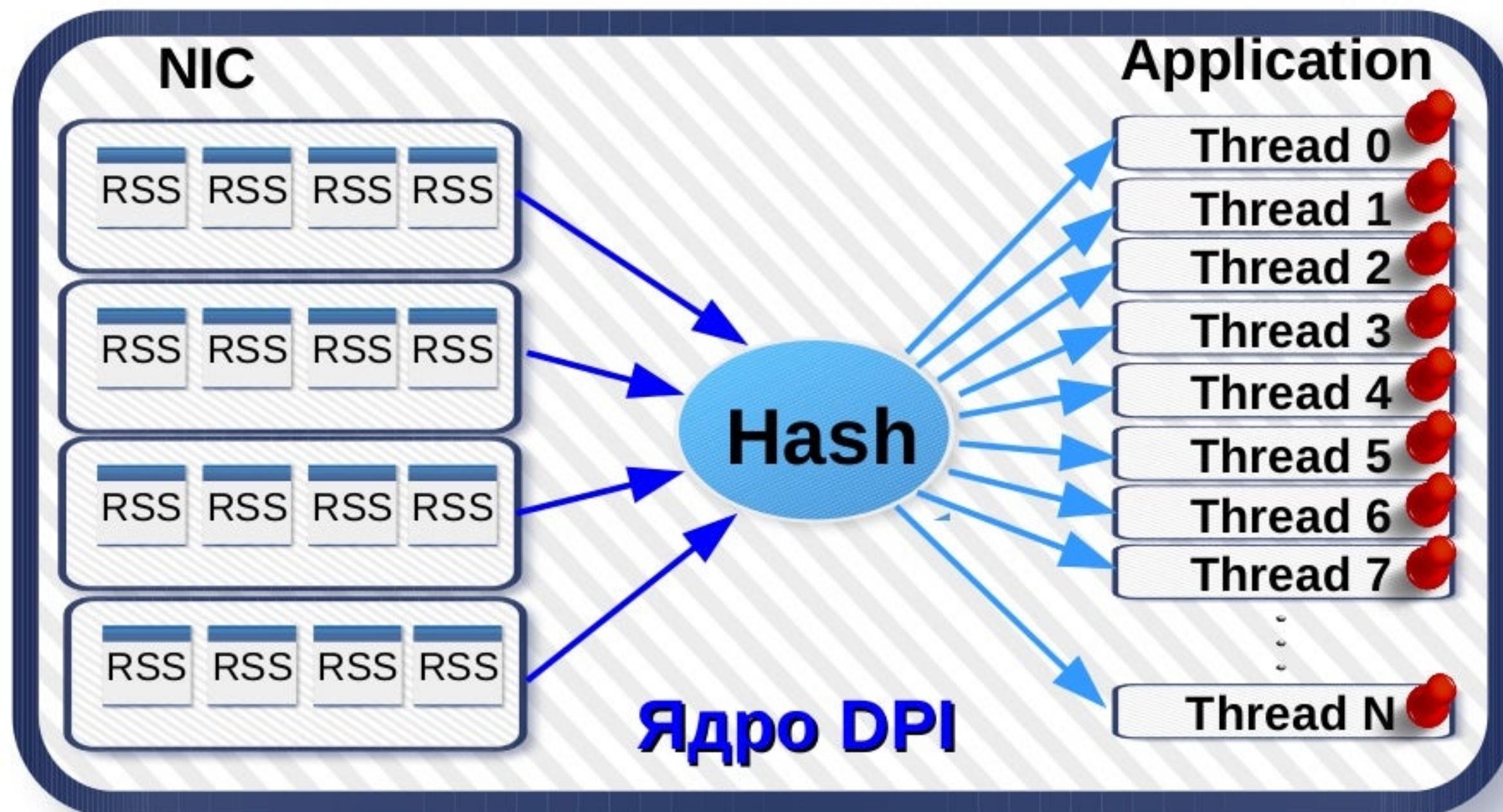
HP[K]	Retired[R]
0	0
1	1
...	2
K - 1	...
	R - 1

```
void hp::Scan() {  
    void * guarded[K*P] = union HP[K] for all P thread;  
    foreach ( p in current_thread.Retired[R] )  
        if ( p not in guarded[] )  
            delete p;  
}
```

Гарантия scan():

$\langle K, P, R \rangle : R > K * P$

Hazard Pointers



HP применима, если:

- Либо **Scan()** вызывается в apartment, не в worker thread
- Либо **Scan()** в worker threads подменяет Retired array на пустой из пула и передает заполненный Retired array в apartment

Спасибо за внимание

