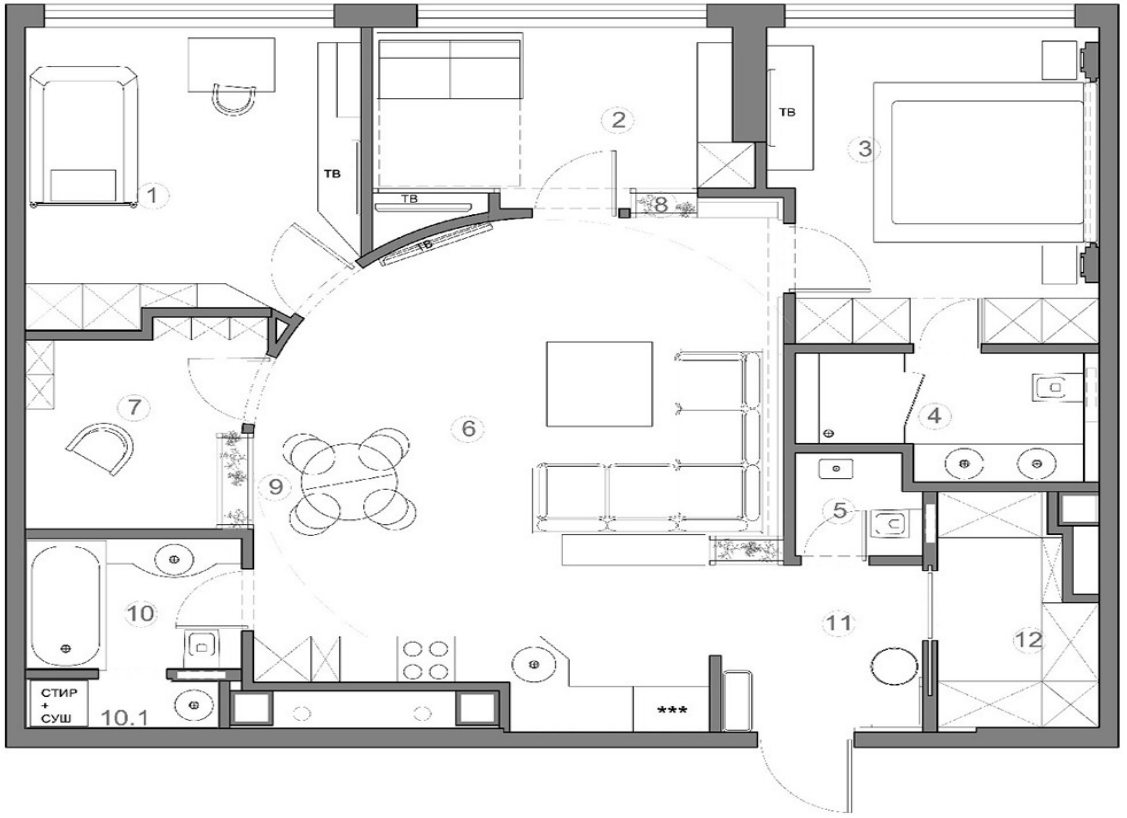


Жилье комфорт-класса для акторов и хендлеров

std

Асинхронная разработка:

- **std::async**
- **ASIO**

Что здесь плохого?

std

Асинхронная разработка:

- **std::async**
- **ASIO**

Что здесь плохого?

Thread pool

Что такое хорошо...

- На каждое ядро – ровно один поток
- Потоки статические: создаются при старте программы, уничтожаются при её остановке
- Поток прибивается гвоздем к своему ядру
- Все данные – локальные для потока

… и что такое плохо

- **Разделяемые данные**
- **Переключение контекста потоков**
- **Создание потока на лету, под кратковременную задачу**
- **Синхронизация**

Thread pool

- [Pool] Разделяемые данные
- [Pool] Переключение контекста потоков
- Создание потока на лету, под кратковременную задачу
- [Pool] Синхронизация

Девелопмент

Программа = совокупность компонентов (микросервисов, акторов)

Компонент:

- статический: время жизни = времени жизни программы

- не надо заботиться о «живости»

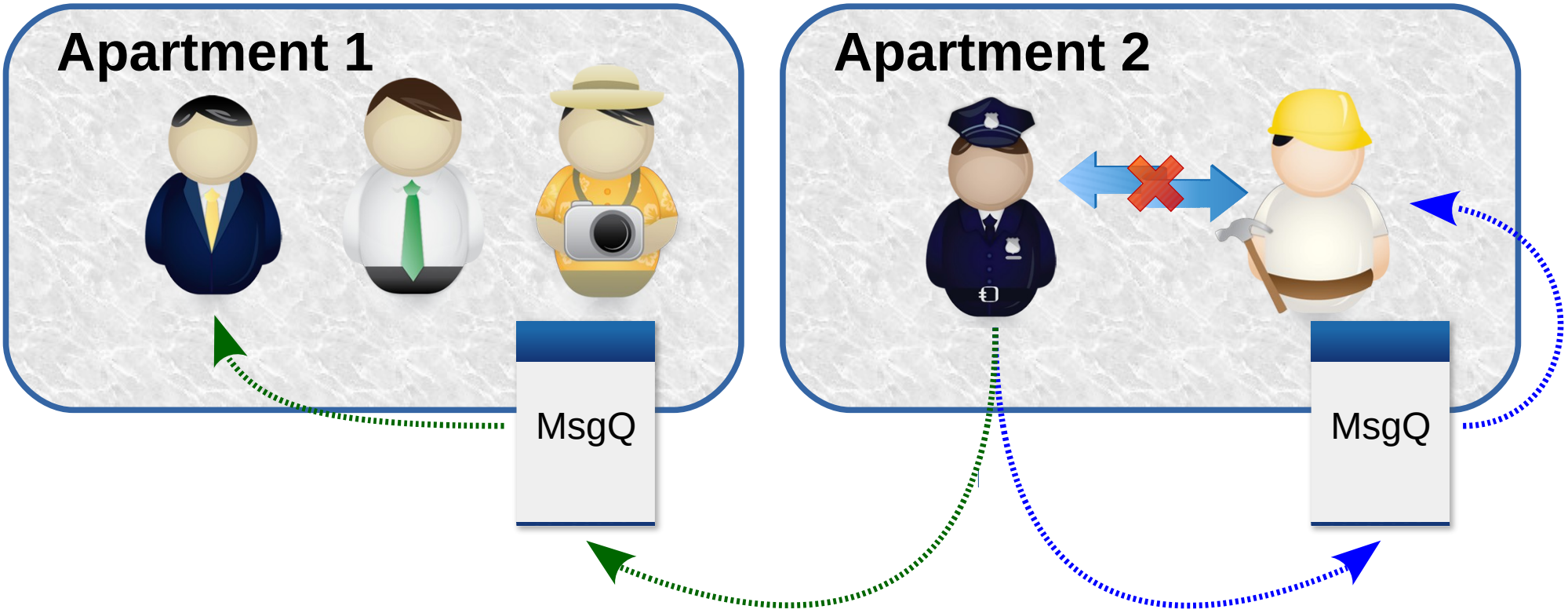
- однопоточный: API вызывается в одном и том же потоке

- нет синхронизации

Апартмент

- Апартмент = поток + MPSC queue
- Число апартментов = числу ядер

Апартменты



Каждый компонент — single threaded
Общение между компонентами — async call
Компоненты не знают о расположении друг друга

Apartment API

```
class apartment
{
public:
    // Вызов функции в данном апартменте
    // sizeof(ArgN) <= sizeof(uintptr_t)
    void call( void(*func)());

    template <typename Arg>
    void call( void(*func)(Arg), Arg arg );

    template <typename Arg1, typename Arg2>
    void call( void(*func)(Arg1, Arg2), Arg1 arg1, Arg2 arg2 );

    template <typename Arg1, typename Arg2, typename Arg3>
    void call( void(*func)(Arg1, Arg2, Arg3),
            Arg1 arg1, Arg2 arg2, Arg3 arg3 );

    // вызов с бОльшим числом аргументов
    template <typename Arg >
    void call( void(*func)(am::unique_ptr<Arg>), am::unique_ptr<Arg> args );
};
```

Apartment queue

```
class apartment
{
private:
     struct queue_record {
      union {
           void(*func0)();
           void(*func1)(uintptr_t);
           void(*func2)(uintptr_t, uintptr_t);
           void(*func3)(uintptr_t, uintptr_t, uintptr_t);
      };
       unsigned arg_count;
       uintptr_t arg1, arg2, arg3;
   };
   mpsc_queue<queue_record, 4096> queue_;
};
```

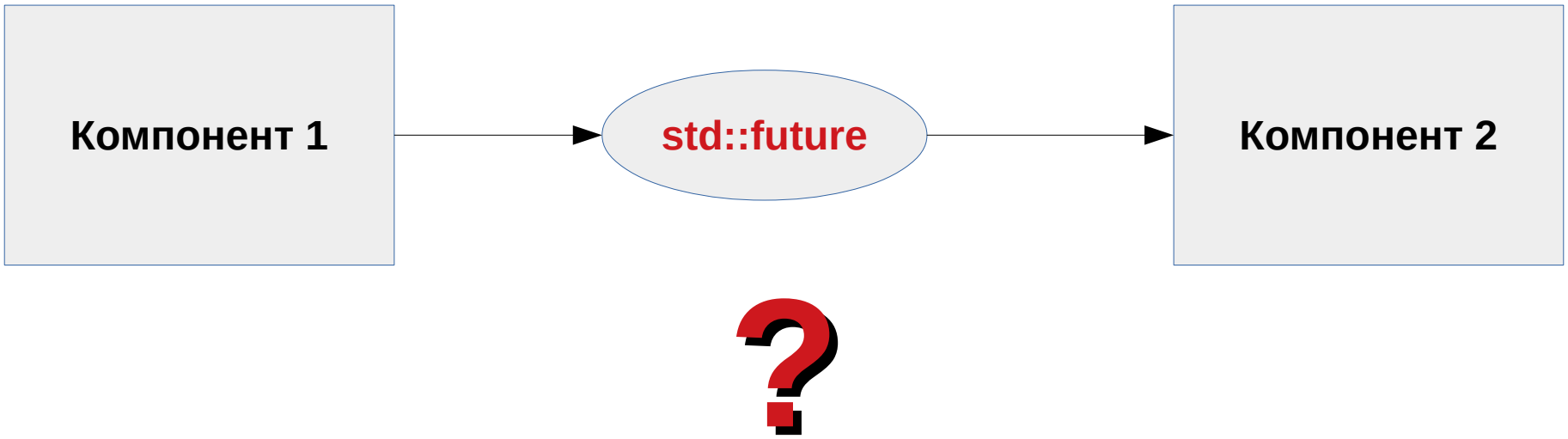
Component API example

```
class Component {
public:
    static void populate( apartment* ap ) { apartment_ = ap; }
    static apartment* get_apartment() { return apartment_; }

    // Безусловно асинхронный вызов
    static void action1()
    { apartment_->call( do_action1 ); }

    // Условно асинхронный вызов
    static void action2( unsigned arg )
    {
        if ( current_apartment() != apartment_ )
            apartment_->call( action2, arg );
        else {
            // находимся в своем апартменте
            // делаем что должно
        }
    }

private:
    static void do_action1() { /* . . . */ }
private:
    static apartment* apartment_;
};
```

No getters!

Все методы API компонента не имеют возвращаемых значений!

API компонента — это вызов действий



Линеаризация std::future

```
Component1::action([](am::unique_ptr<future> args)
    {
      // находимся в апартменте компонента 1
      // заполняем поля args
      args->field = …

      // передаем вызов в апармент компонента 2
      Component2::action2( std::move( args ));
    }

  );
```

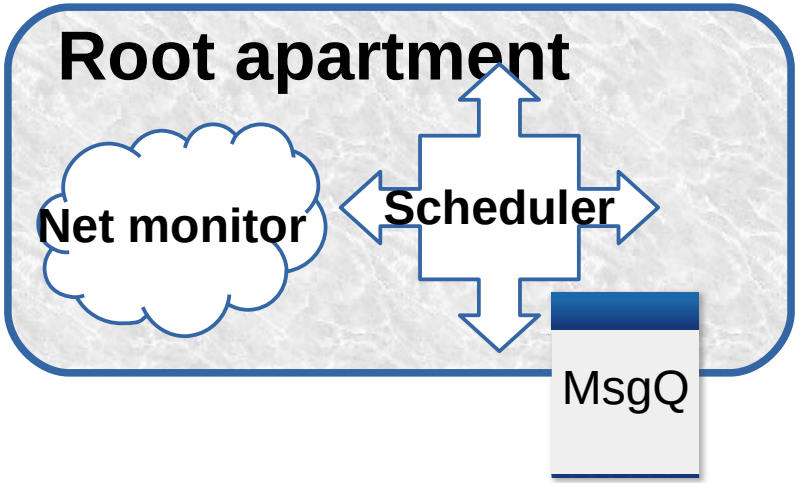
Наращиваем сервисы

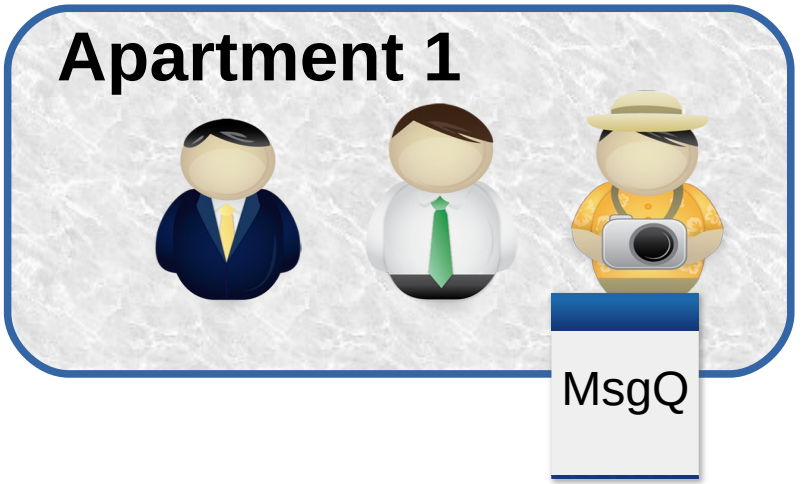
Необходимо:

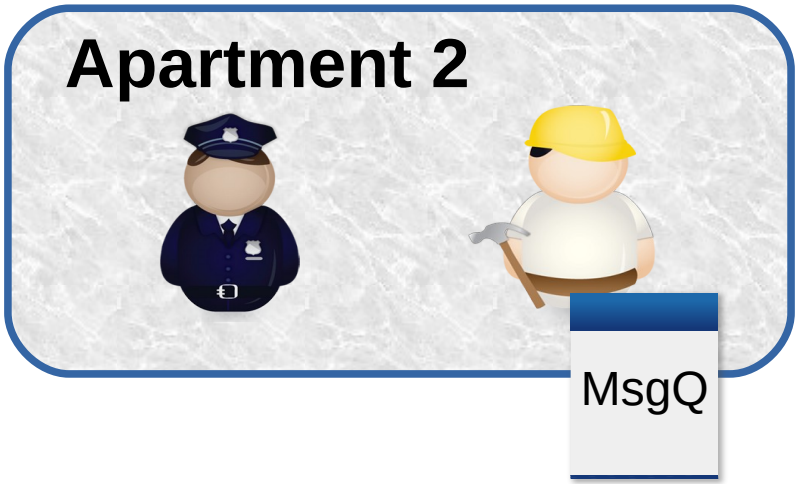
- **Сетевое взаимодействие**
- **Планировщик — выполнение периодических действий и действий по расписанию**

Root apartment

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







Connection monitor API

```
struct connection_monitor::descriptor {
        typedef void (* handler )( connection_monitor* mon, descriptor* desc,
              int flags
        );
        int fd = -1; // socket

        // чего ждем - флаги epoll: EPOLLIN, EPOLLOUT, EPOLLONESHOT, EPOLLET
        unsigned mode = 0;

        // Целевой apartment, в котором вызывать функции-обработчики
        apartment*  target_apartment = nullptr;

        // Обязательный обработчик готовности сокета.
        handler on_data_ready = nullptr;

        // Обязательный обработчик ошибок.
        handler on_error = nullptr;

        // Вызывается при событиях EPOLLHUP или EPOLLRDHUP - закрытия сокета
        handler on_socket_closed = nullptr;
    };
```

Connection monitor API

```
class connection_monitor {
public:
    // Добавляет дескриптор \p desc в монитор
    // @warning Функция не копирует \p desc,
    // а хранит указатель на переданный аргумент
    bool add( descriptor& desc);

    // Удаляет дескриптор desc из списка мониторируемых
    bool remove( descriptor& desc );

    // Временно прекращает мониторинг дескриптора
    bool suspend( descriptor& desc );

    // Возобновляет мониторинг дескриптора
    bool resume( descriptor& desc );
};
```

Scheduler API

```
class scheduler {
public:
    // запуск функции func через timeout в апартменте ap
    task_id_t schedule( apartment& ap, timeout_t timeout, void (* func)());

    // регистрация периодической задачи func, выполняющейся в апартменте ap
    task_id_t schedule_periodic( apartment& ap, timeout_t start_timeout,
                timeout_t period, void (* func)());

    // Отзыв задачи
    void revoke( task_id_t task_id );
};
```

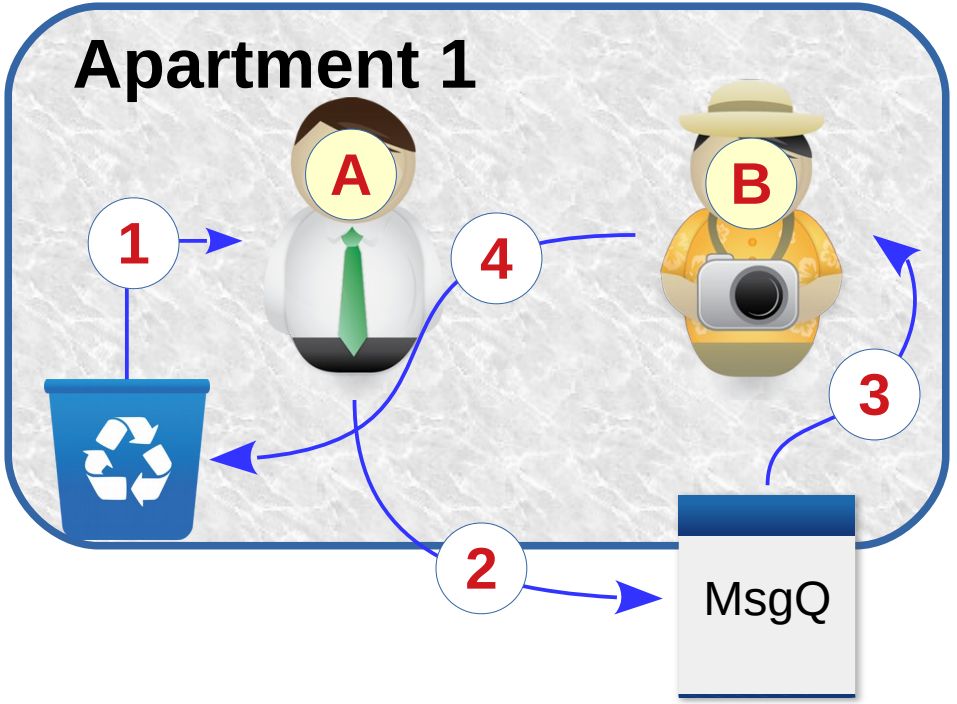
Apartment allocator

Требования:

- Thread-local
- Избежать фрагментации
- **free**(**ptr**): нужно быстро понять, каким апартментом аллоцирован **ptr**
- Выделенный блок памяти не должен иметь префиксов / управляющих структур

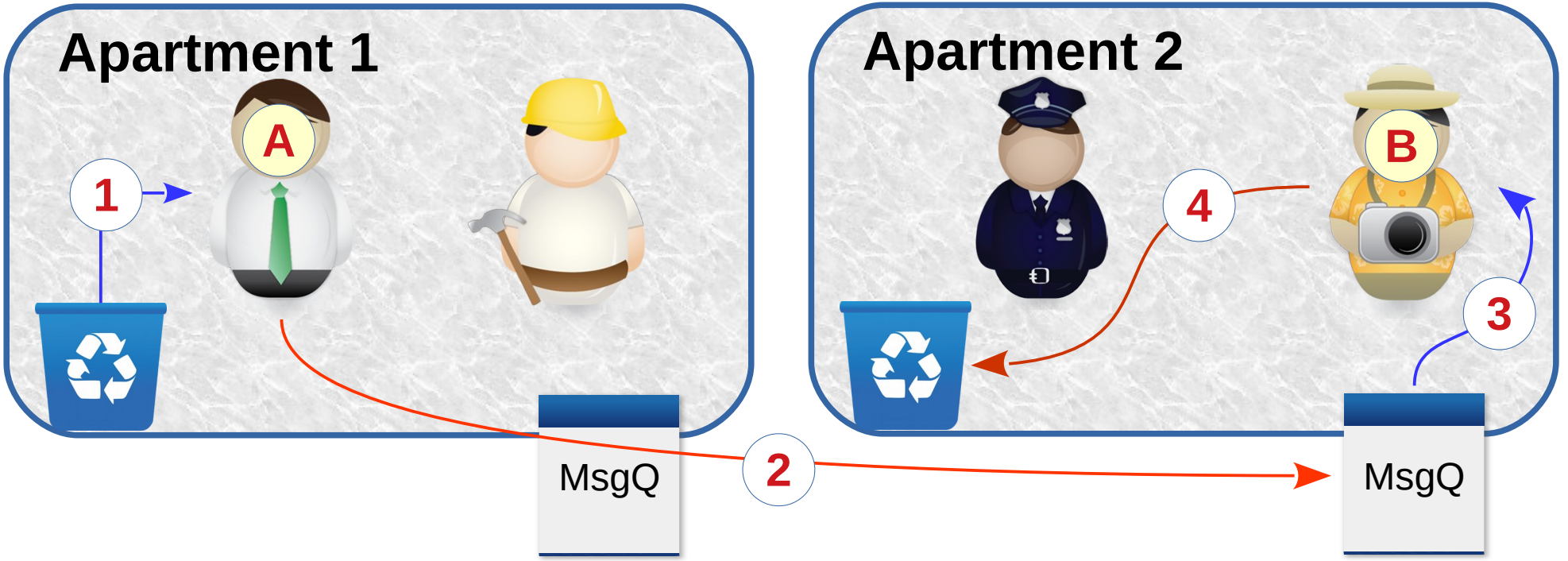
Решение: **страничный single-threaded субаллокатор**

Apartment allocator



1. A: ptr = ap1.alloc(64)
2. A: B.invoke(foo, ptr);
3. B: foo(ptr) { ...
4. B: ap1.free(ptr); }

Apartment allocator



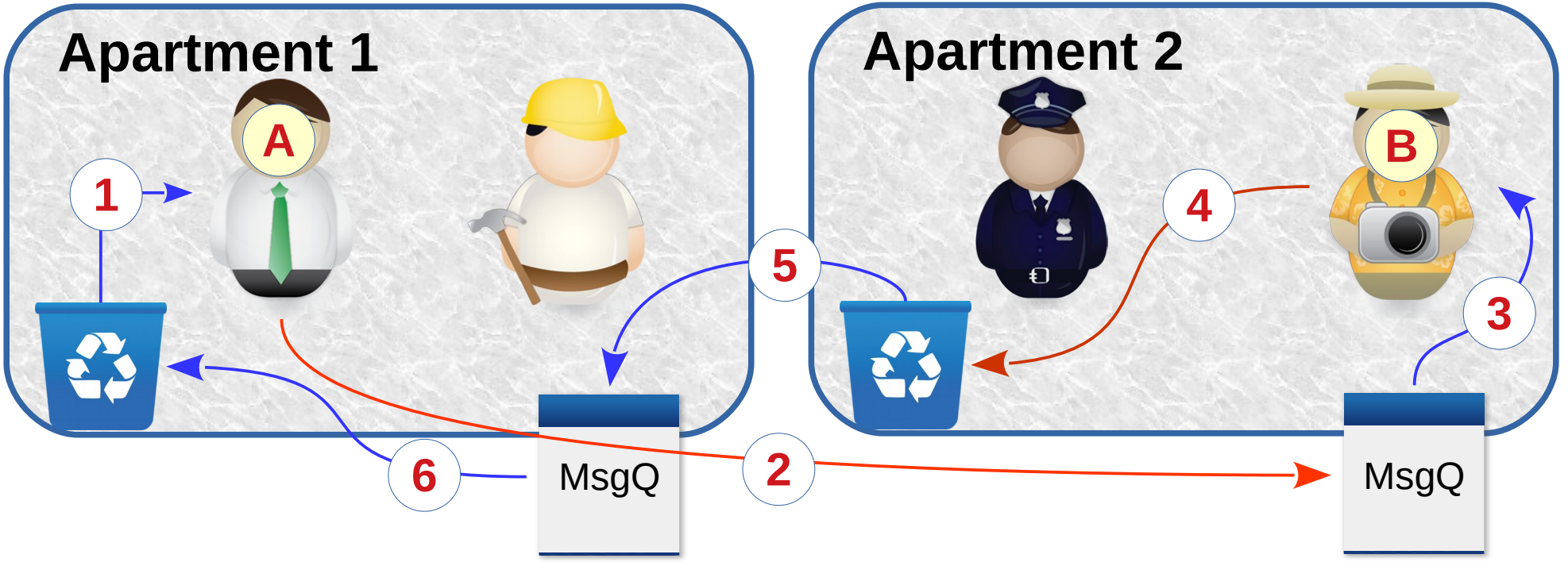
1 **A:** ptr = ap1.alloc(64)

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

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

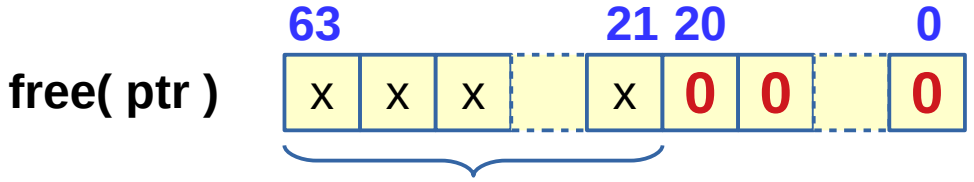
4 **B:** ap2.free(ptr); } - oops!!!

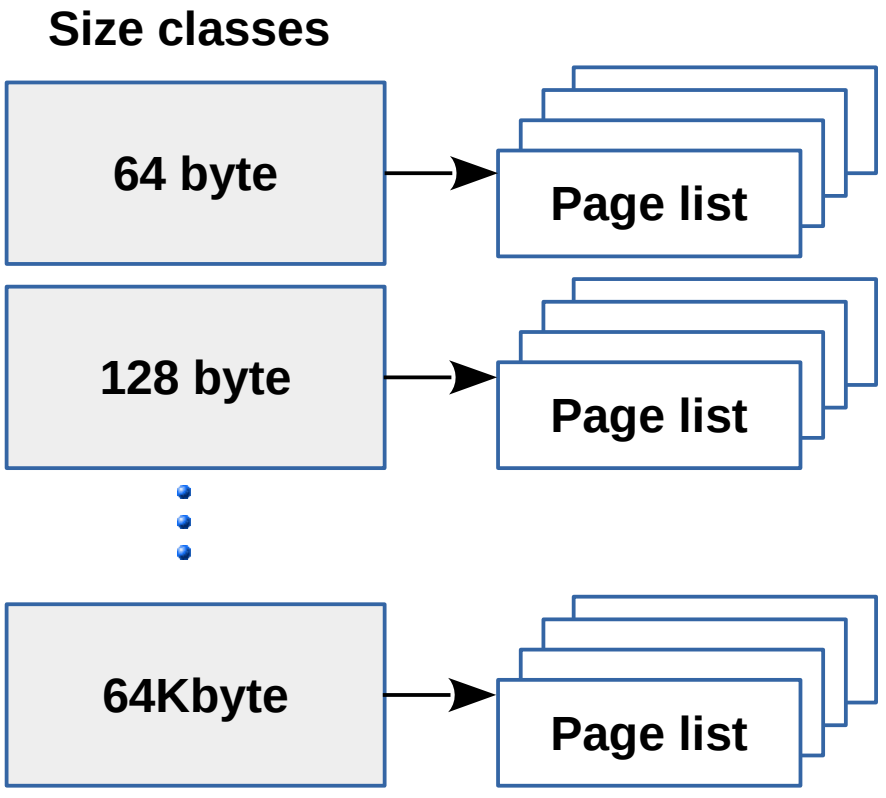
Apartment allocator



1. **A:** ptr = ap1.alloc(64)
2. **A:** B.invoke(foo, ptr);
3. **B:** foo(ptr) { ...
4. **B:** ap2.free(ptr); }
5. **ap2 allocator**: ap1.invoke(free, ptr)
6. **ap1:** ap1.free(ptr)

Apartment allocator

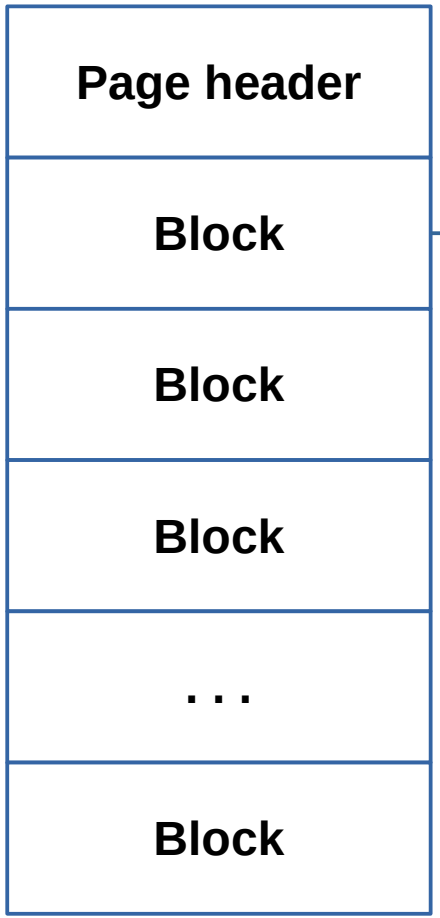


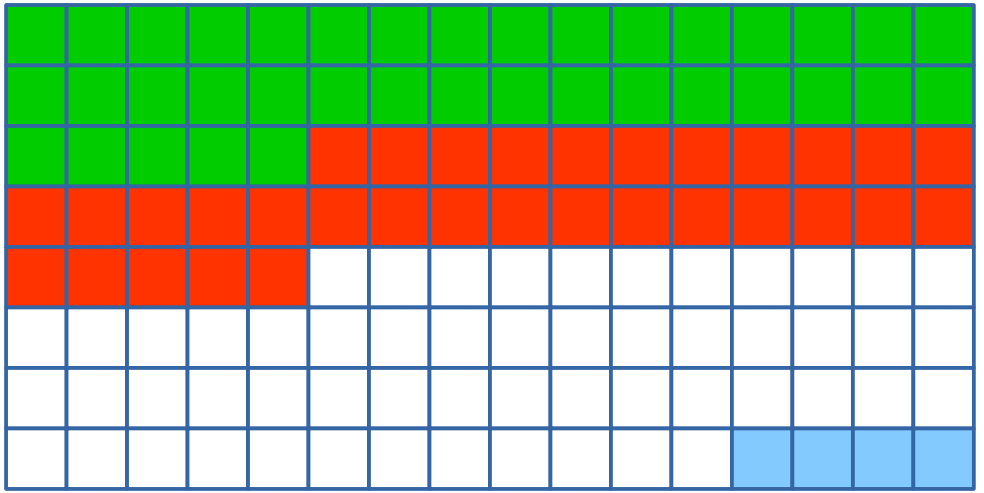


page_size = 2M, 2M-aligned

```
struct page_header {
  uintptr_t magic;       // magic word
  size_class_desc* psc; // size_class
  // апартмент-владелец страницы
  apartment* owner;
  // 1-й свободный блок страницы
  void* first_free_block;
  // связь в списки для size_class
  page_header* next;
  //...  прочие данные
};
```

Dynamic memory debugging





Data area

Red zone 0xFD

Blocktail guard

Unused

```
struct blocktail_guard {
  // сколько байт
  // запрашивалось alloc()
  uint32_t  real_size;
  // magic word
  uint32_t  guard;
};
```

alloc(N) -> alloc(N + sizeof(red_zone) + sizeof(blocktail_guard))

Спасибо за внимание!