ПО сетевых устройств

Трещановский Павел Александрович, к.т.н.

14.05.20

Текущее положение

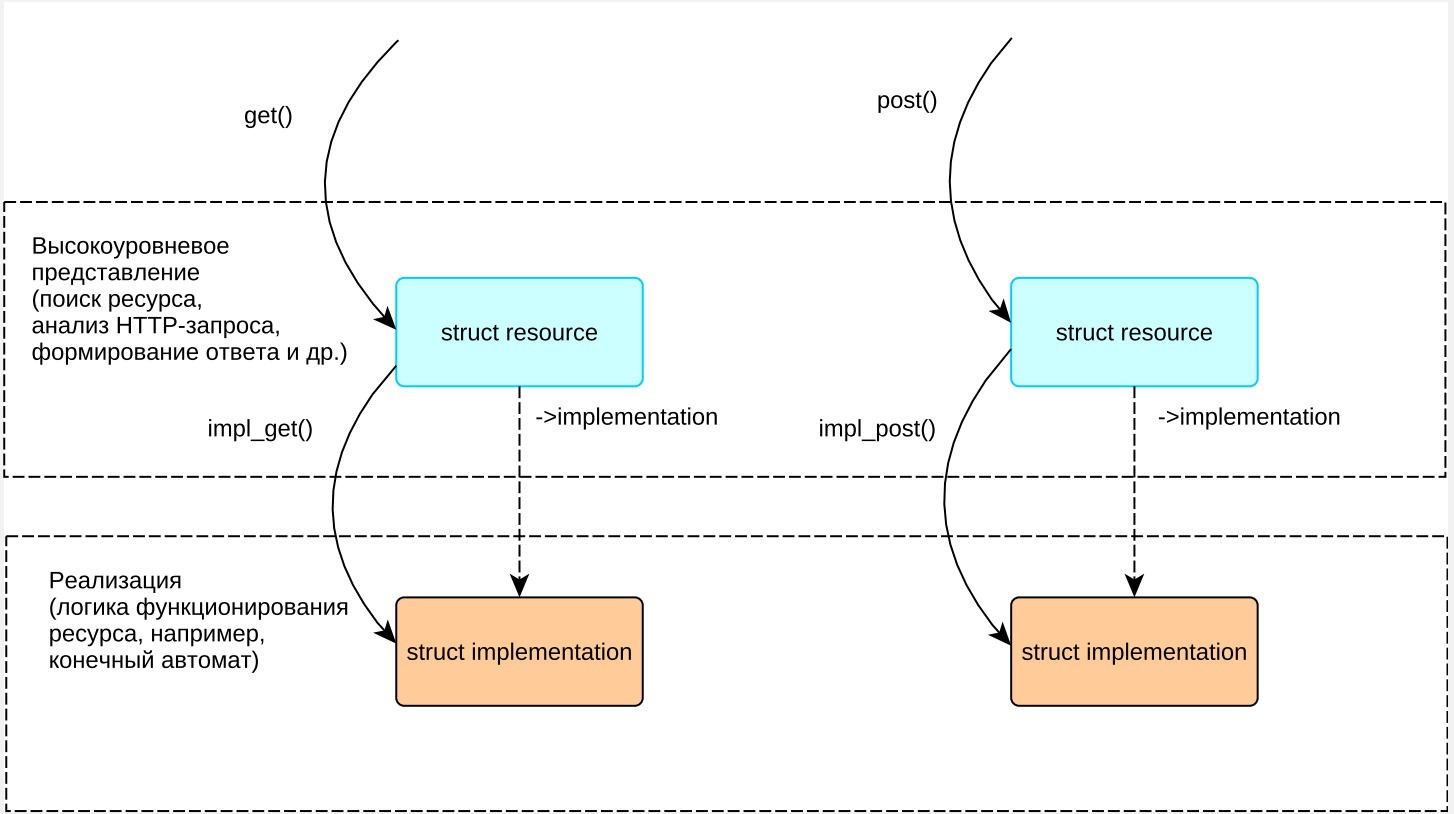
- Сервер, принимающий и обрабатывающий TCP-соединения с помощью сокета.
- Внутренние объекты сервера могут представляться в виде набора сетевых ресурсов. Чтение и изменение ресурсов производится через вызов методов (GET, POST, PUT и т.д.).
- Сервер поддерживает одновременное исполнение нескольких задач с помощью цикла обработки событий.
- Задача может представлять собой исполнение экземпляра конечного автомата.

Какие еще бывают задачи? Какого их внутренняя структура? Как организовано взаимодействие между их компонентами?

Отдельные аспекты программной архитектуры

- Разбиение на уровни абстракции.
- Динамическое создание и удаление ресурсов.
- Полиморфные ресурсы - общий API для ресурсов с разной реализацией.
- Эффективное индексирование ресурсов с помощью словарей.
- Обмен данными между уровнями абстракции через очередь событий.

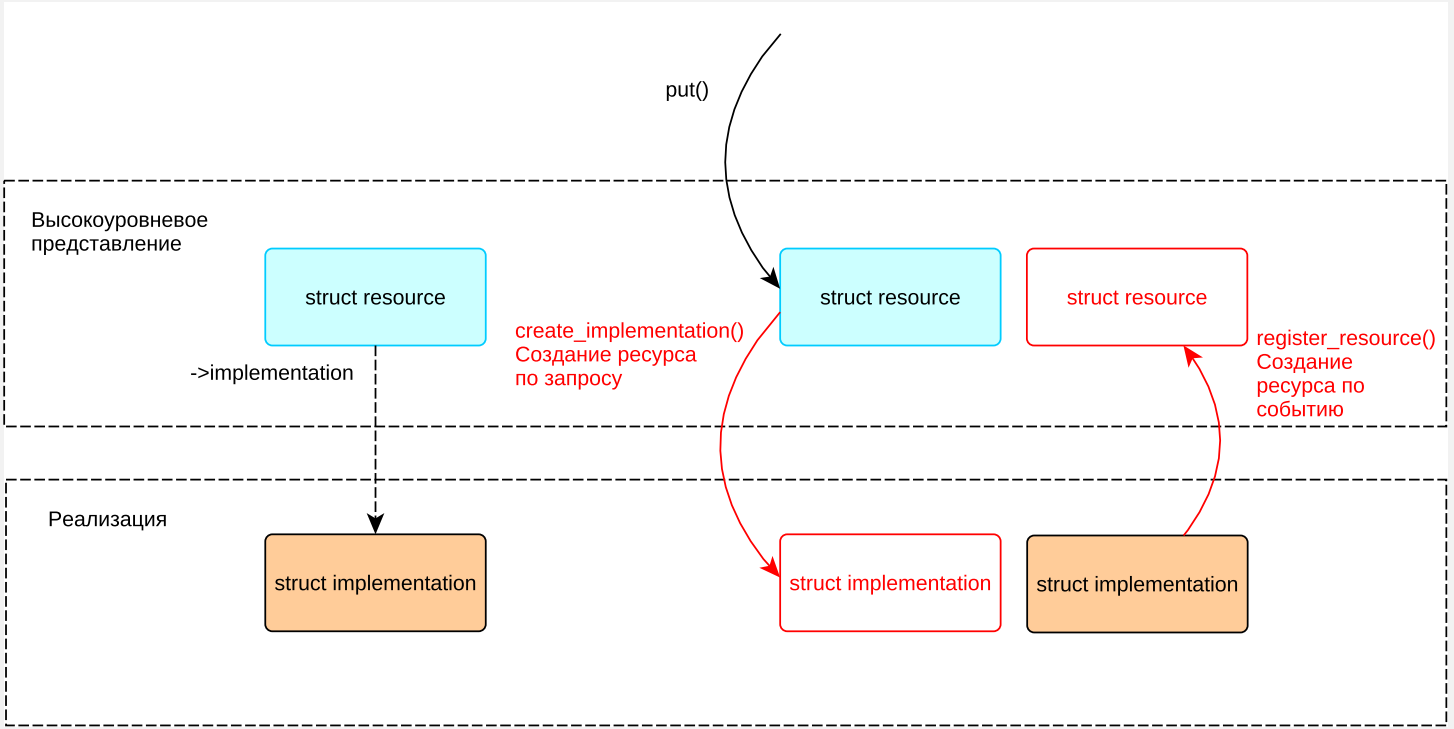
Многоуровневое приложение



Многоуровневое приложение, код

```
int post(struct resource *res, const char *http_request)
{
	struct implementation *impl = res->implementation;

	parse_http_request(http_request, ...);

	impl_post(impl, ...);

	send_http_response(...);
}

int impl_post(struct implementation *impl, ...)
{
	run_state_machine(impl->state, event);
}

/* Функция скрыта от высокоуровневого модуля */
static run_state_machine(enum state state, enum event event)
{
	...
}
```

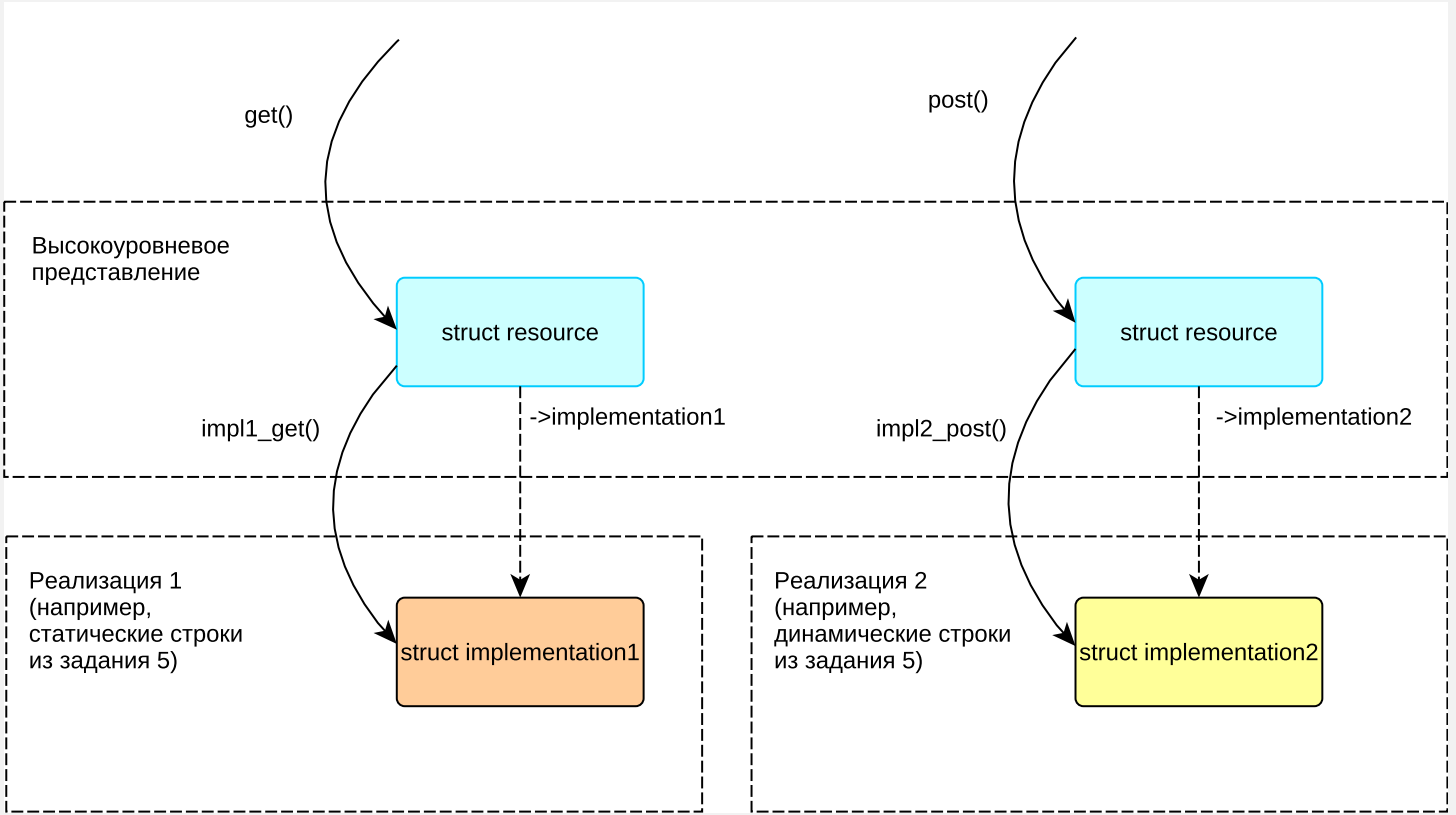
Динамическое создание ресурсов



Создание и регистрация ресурсов

```
int register_resource(struct implementation *impl, ...)
{
       struct resource *res;

       res = calloc(1, sizeof(*res));
       res->implementation = impl;
}

int put(const char *http_request)
{
       struct resource *res;
       struct implementation *impl;

       ...

       res = calloc(1, sizeof(*res));

       impl = create_implementation(res, ...);
       res->implementation = impl;
}
```

Поддержка нескольких реализаций



Простой вариант

```
int register_resource(enum resource_type type, struct implementation1 *impl1, ...)
{
	struct resource *res = calloc(1, sizeof(*res));

	res->type = type;
	if (type == IMPL1)
		res->implementation1 = impl1;
}

int post(struct resource *res, const char *http_request)
{
	...

	switch (res->type) {
	case IMPL1:
		impl1_post(impl, ...);
		break;
	case IMPL2:
		impl2_post(impl, ...);
		break;
	}
}
```

Обратный вызов

```
typedef int (*callback_t)(void *callback_data);

int register_resource(callback_t callback, void *callback_data)
{
       struct resource *res = calloc(1, sizeof(*res));

       res->callback = callback;
       res->callback_data = callback_data;
}

int post(struct resource *res, const char *http_request)
{
       ....

       res->callback(res->callback_data);
}

static int impl1_post(void *callback_data)
{
       struct implementation1 *impl = callback_data;
       ...
}
```

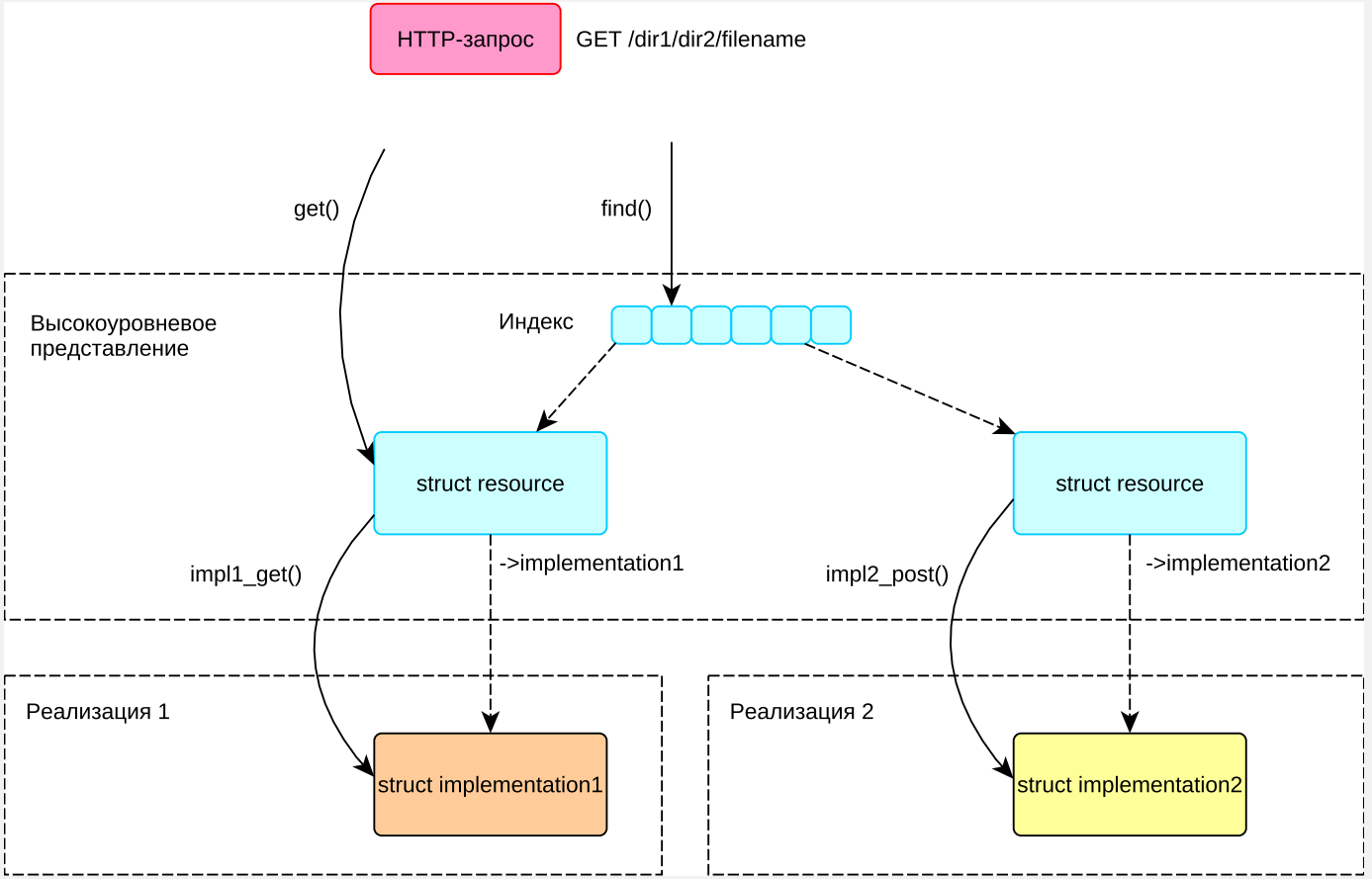
Замечания

- Обратный вызов - функция, которая вызывается через указатель.
- `callback_t` - тип этой функции.
- `callback_data` - данные, которые надо передать в обратный вызов во время его исполнения. Тип `void *` позволяет каждой реализации ресурса использовать данные своего типа (например, `struct implementation1` или `struct implementation2`). Приведение от указателя на структуру к `void *` и наоборот происходит автоматически.
- Указатель на функцию и указатель `callback_data` записывается в объект `struct resource` во время регистрации ресурса (`resource_register()`).

Таблица функций

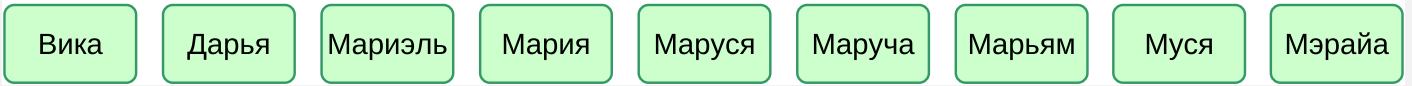
```
struct resource_operations {
       int (*get)(struct resource *res, ...);
       int (*post)(struct resource *res, ...);
};

int register_resource(struct resource_operations *ops, void *private_data)
{
       res->operations = ops;
       res->private_data = private_data;
}

int post(struct resource *res, const char *http_request)
{
       ....
       res->operations->post(res);
}

static int impl1_post(struct resource *res)
{
       struct implementation1 *impl = res->private_data;
       ...
}
```

Индексация ресурсов



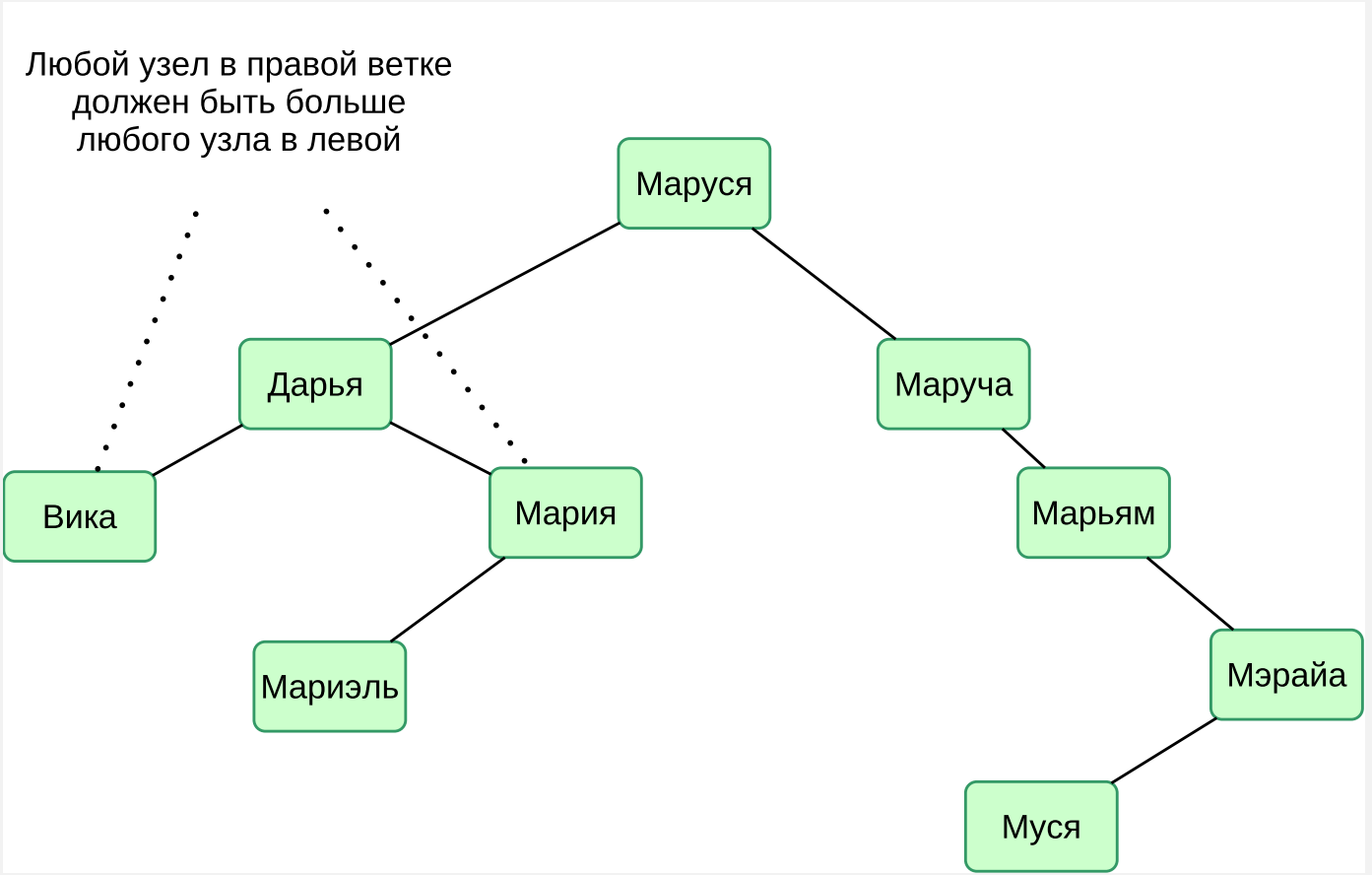
Обычный массив

Неупорядоченный массив, индексируемый строкой:

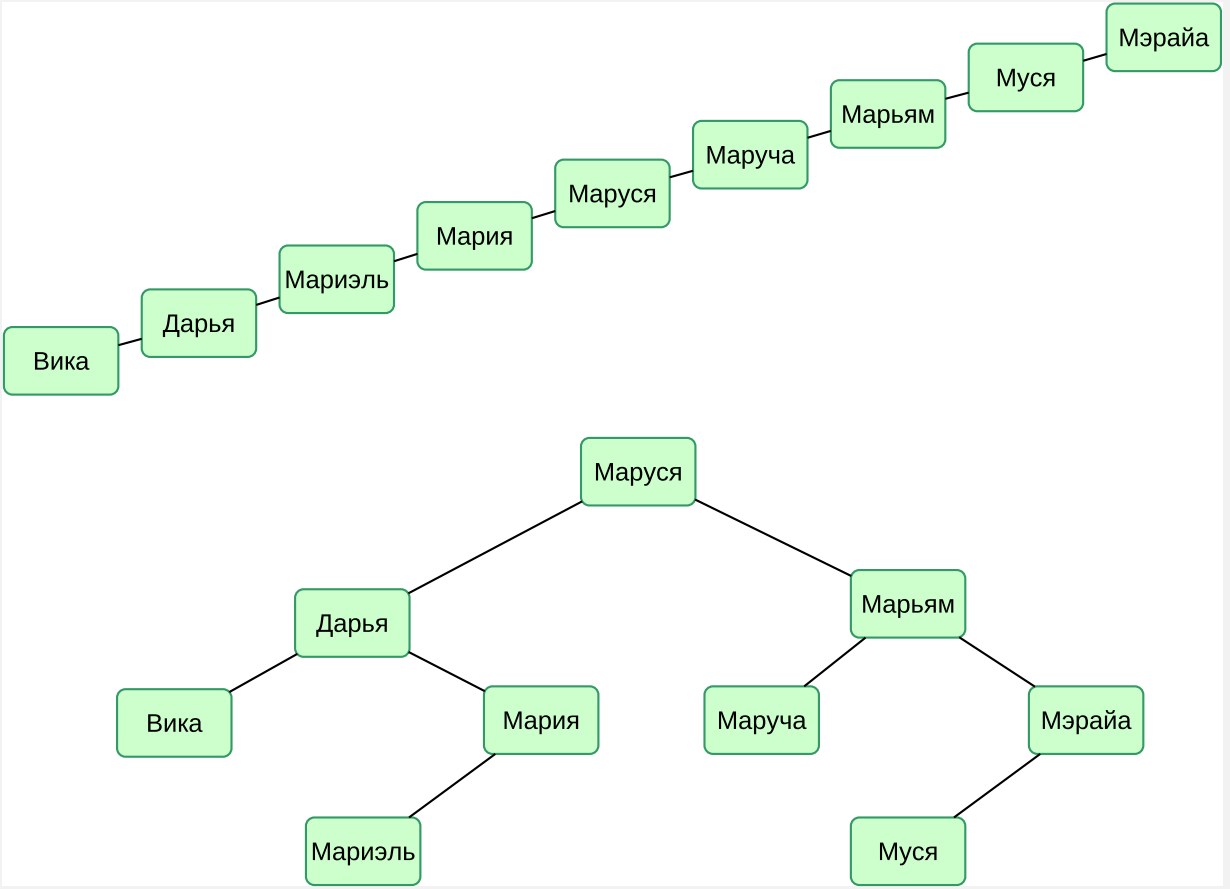



```
struct resource {
	char *key;
	struct implementation *impl;
} *resource[MAX_RESOURCES];

struct resource *find_resource(const char *key)
{
	int i;

	for (i = 0; i < MAX_RESOURCE; i++) {
		struct resource *r = &resources[i];

		if (r && !strcmp(r->key, key))
			return r;
	}
	return NULL;
}
```

Двоичное дерево



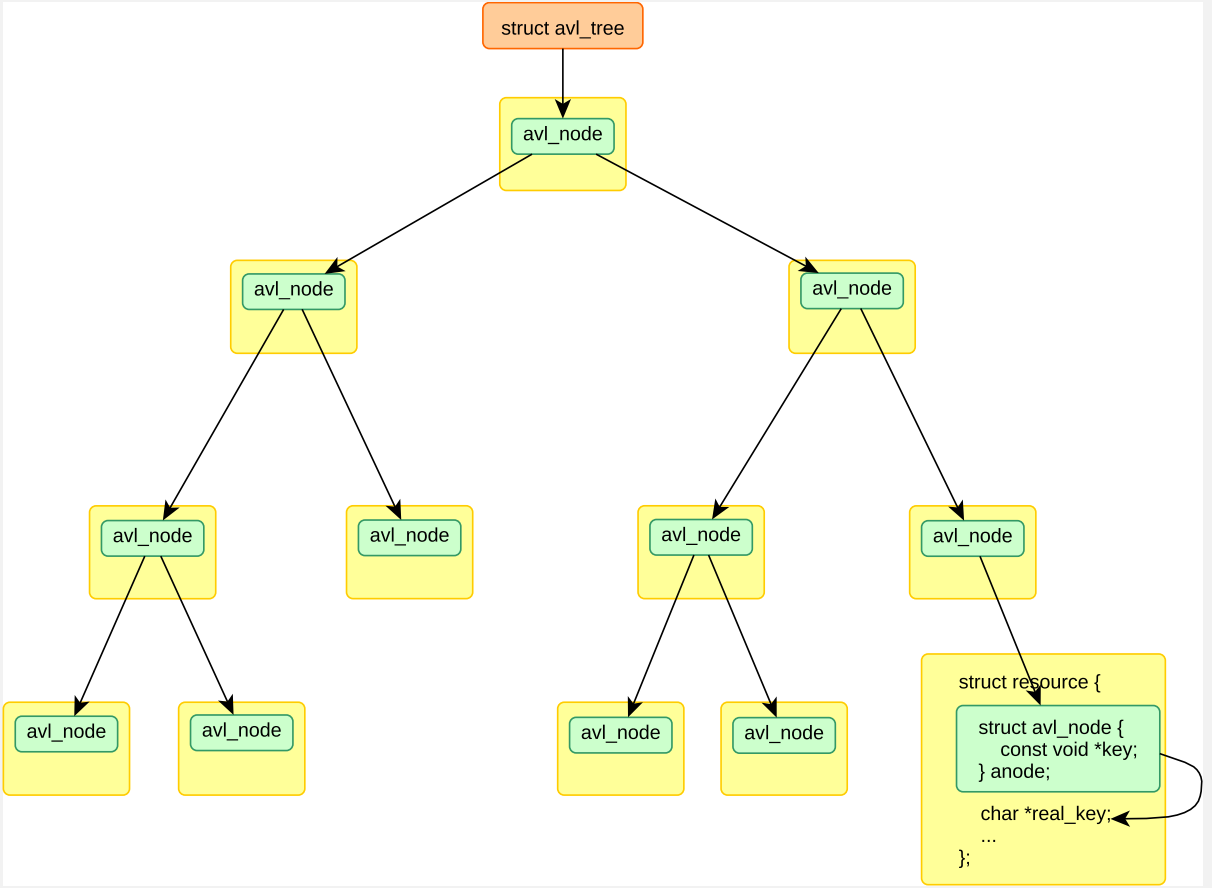
Балансирование дерева



Самобалансирующееся дерево

- Время поиска элемента словаря в худшем случае определяется глубиной дерева. Наименьшее время достигается при равномерном распределении узлов по всем ветвям дерева.
- Самобалансирующееся дерево - дерево, у которого распределение узлов по ветвям происходит автоматически при добавлении нового узла.
- AVL - одно из самобалансирующихся деревьев.
- Библиотека libubox предоставляет реализацию деревьев AVL - `struct avl_tree`.
- `struct avl_tree` скрывает двоичное дерево и предоставляет приложение API ассоциативного массива (словаря) с возможностью добавления и поиска по ключу.
- Ключом может быть объект произвольного типа, в частности, строка. При инициализации словарю надо передавать функцию сравнения ключей данного типа (`avl_strcmp` для строк).

struct avl_tree



Преобразование указателей

```
struct resource {
	struct avl_node anode;
};
```

Мы добавляем и извлекаем из словаря объекты struct avl_node, встроенные в struct resource. Как из указателя на struct avl_node получить указатель на struct resource?

```
/* Смещение между адресом родительского объекта типа type и адресом его поля
member. */
#define offsetof(type, member)	((size_t)&((type *)0)->member)

/* Получение указателя на родительский объект типа type из указателя ptr на его
поле member. */
#define container_of(ptr, type, member) ({			\
	void *__mptr = (void *)(ptr);				\
	((type *)(__mptr - offsetof(type, member))); })

int function(struct avl_node *n)
{
	struct resource *res = container_of(n, struct resource, anode);
}
```

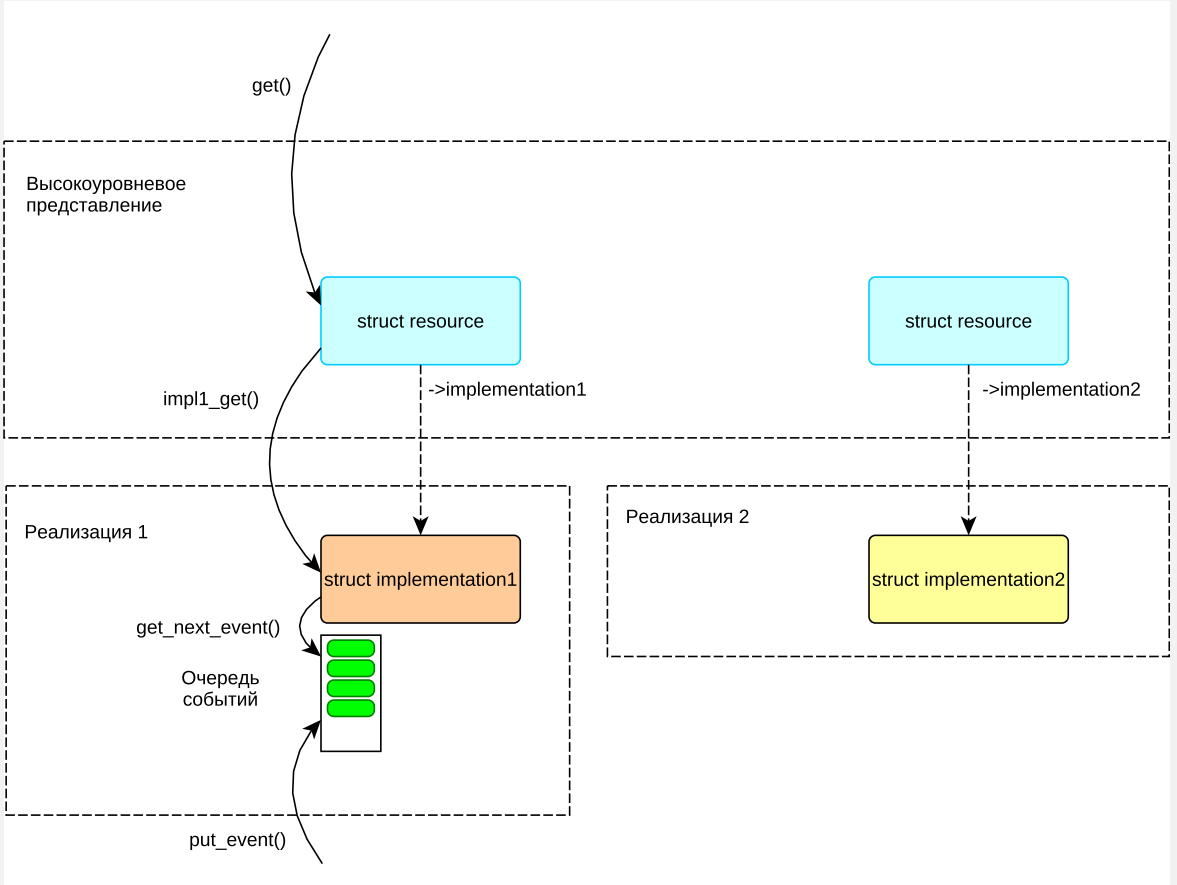
Добавление, поиск и удаление

```
struct avl_tree dict;
struct resource {
       char *key;
       struct avl_node anode;
} *res1, *res2;

/* avl_strcmp - функция сравнения узлов двоичного дерева. */
avl_init(&dict, avl_strcmp, 0, NULL);

res1 = calloc(1, sizeof(*res1));
res1->key = strdup("some-resource-key");
res1->anode.key = res1->key;

ret = avl_insert(&dict, &res1->anode);
if (ret < 0)
       fprintf(stderr, "Key '%s' is already present", res1->key);

res2 = avl_find_element(&dict, "another-key", res2, anode);

avl_delete(&dict, &res2->anode);
free(res2->key);
free(res2);
```

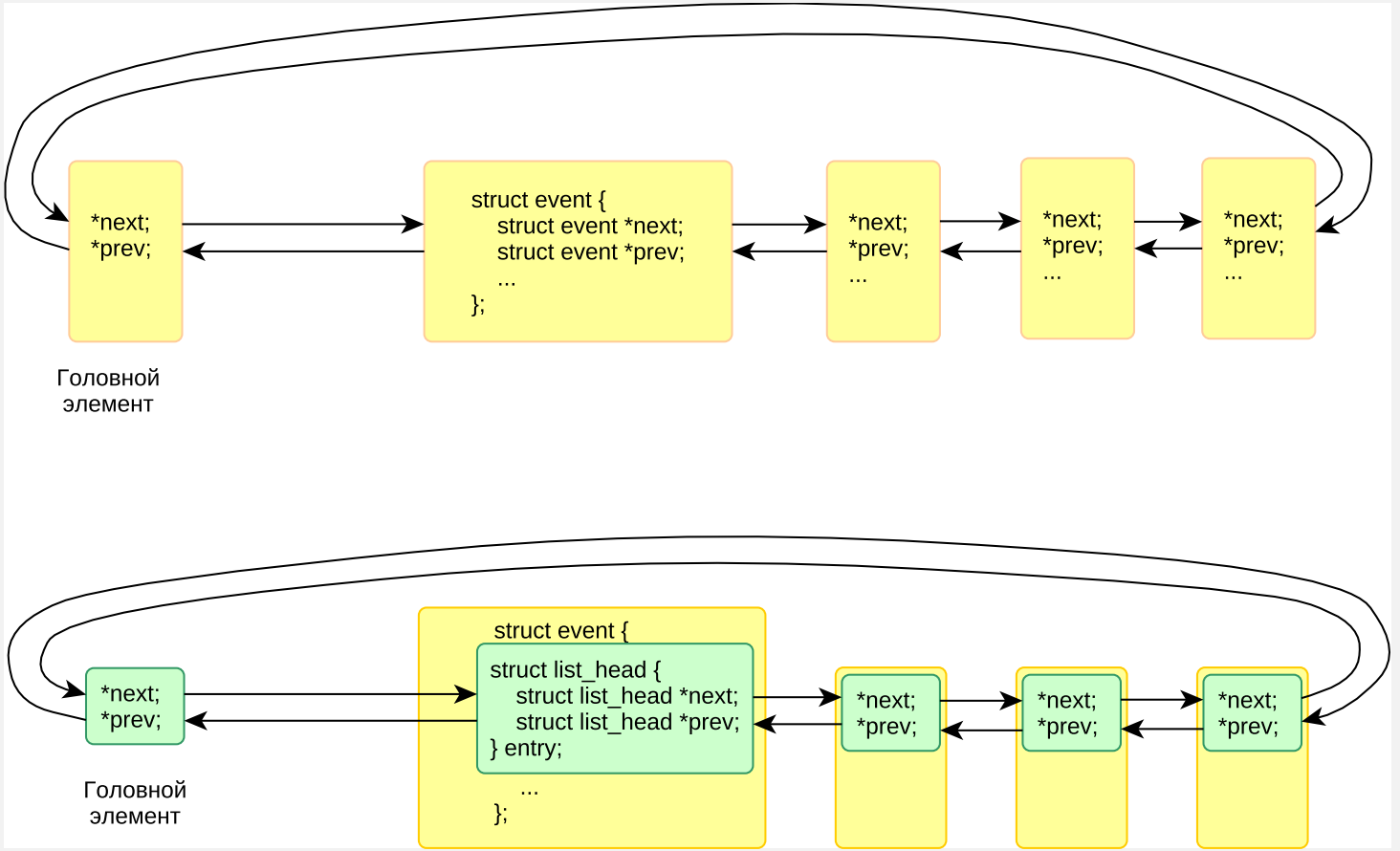
Перебор всех элементов

```
struct avl_tree dict;

struct resource *res, *next_res;

avl_init(&dict, avl_strcmp, 0, NULL);

...

avl_for_each_element(&dict, res, anode) {
       printf("%s\n", res->key);
       /* Нельзя удалять res из словаря!! */
}

/* Если в теле цикла удаляются элементы словаря, то используется вариант _safe. */
avl_for_each_element_safe(&dict, res, anode, next_res) {
       avl_delete(&dict, &res->anode);
       free(res->key);
       free(res);
}
```

Сетевой ресурс с очередью событий



Связный список struct list_head



Добавление, поиск и удаление

```
struct list_head list;
struct event {
       struct list_head entry;
} *event1, *event2;

INIT_LIST_HEAD(&list);

event1 = calloc(1, sizeof(*event1));

/* Добавление в конец списка. */
list_add_tail(&event1->entry, &list);

event2 = list_first_entry(&list, struct event, entry);

list_del(&event2->entry);
```

Перебор всех элементов

```
struct list_head list;

struct event *event, *next_event;
int i = 0;

list_for_each_entry(event, &list, entry) {
       printf("List entry %d\n", i++);
       /* Нельзя удалять event из списка!! */
}

/* Если в теле цикла удаляются элементы списка, то используется вариант _safe. */
list_for_each_entry_safe(event, next_event, &list, entry) {
       list_del(&event->entry);
       free(event);
}
```

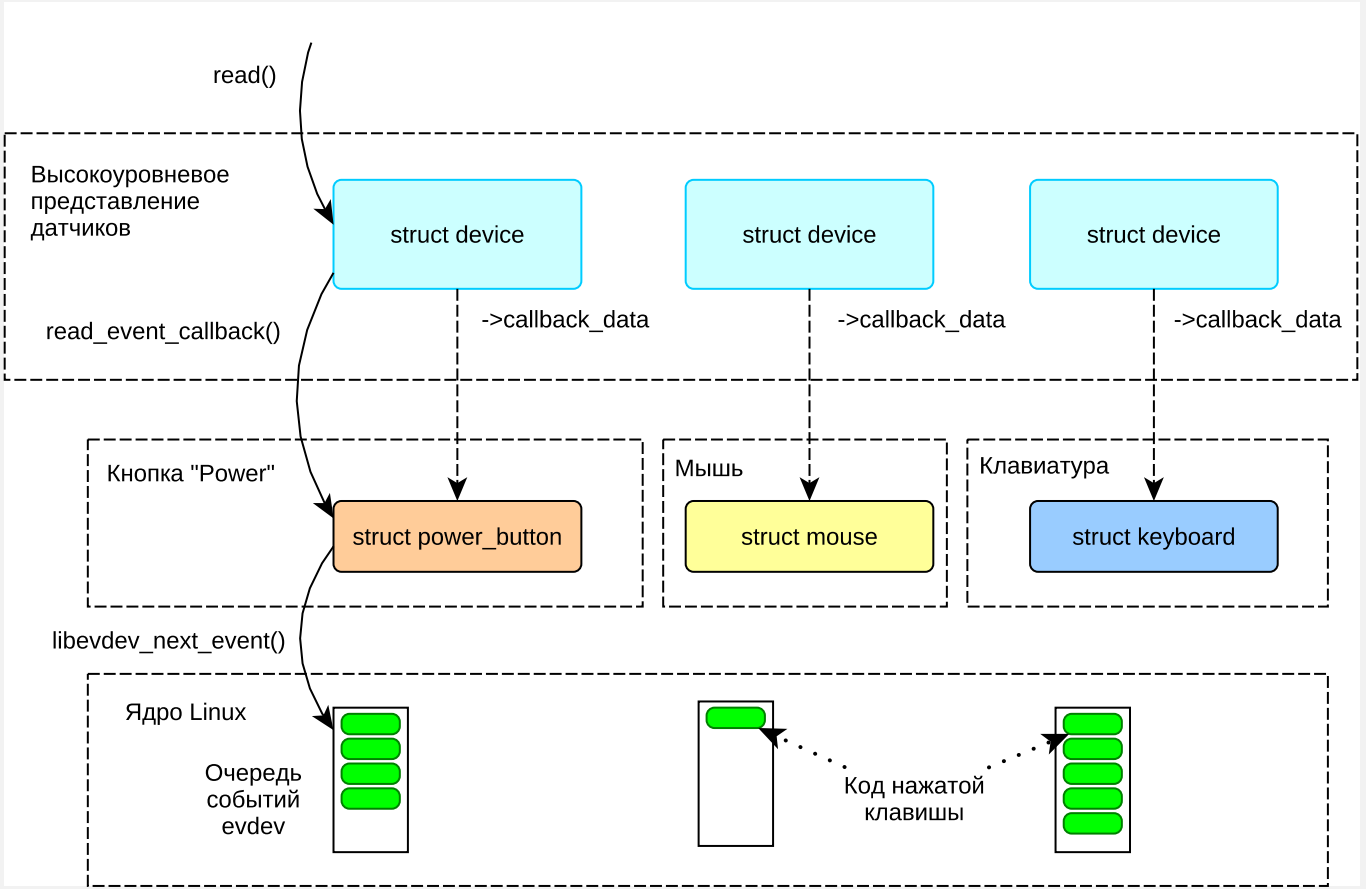
Замечания

- Объекты типа `struct list_head` выполняют 2 роли: заголовок (список в целом) и элемент списка. Элемент списка list_head должен быть встроен в `struct event` (или другую пользовательскую структуру). Заголовок *не* включается в `struct resource`.
- Пустой список представляется в программе заголовком без элементов списка. Заголовок в таком случае указывает сам на себя.
- Каждое поле типа `struct list_head` внутри `struct resource` отвечает за включение ресурса в один список. Если требуется включить ресурс в N списков, то в структуре должно быть N полей `struct list_head`.

Устройства ввода в Linux

- Устройства ввода (клавиатура, мышь, кнопки на корпусе и пр.) представлены в Linux в виде файлов *символьных устройства* в каталоге /dev/input.
- Физическое устройство ввода соответствует одному или нескольким файлам в /dev/input.
- Каждое символьное устройство содержит очередь событий.
- Событие имеет тип (нажатие клавиши, движение мыши), код (какая клавиша нажата) и значение (нажата или отжата).
- Приложения работают с устройствами через библиотеку libevdev.

Опрос датчиков



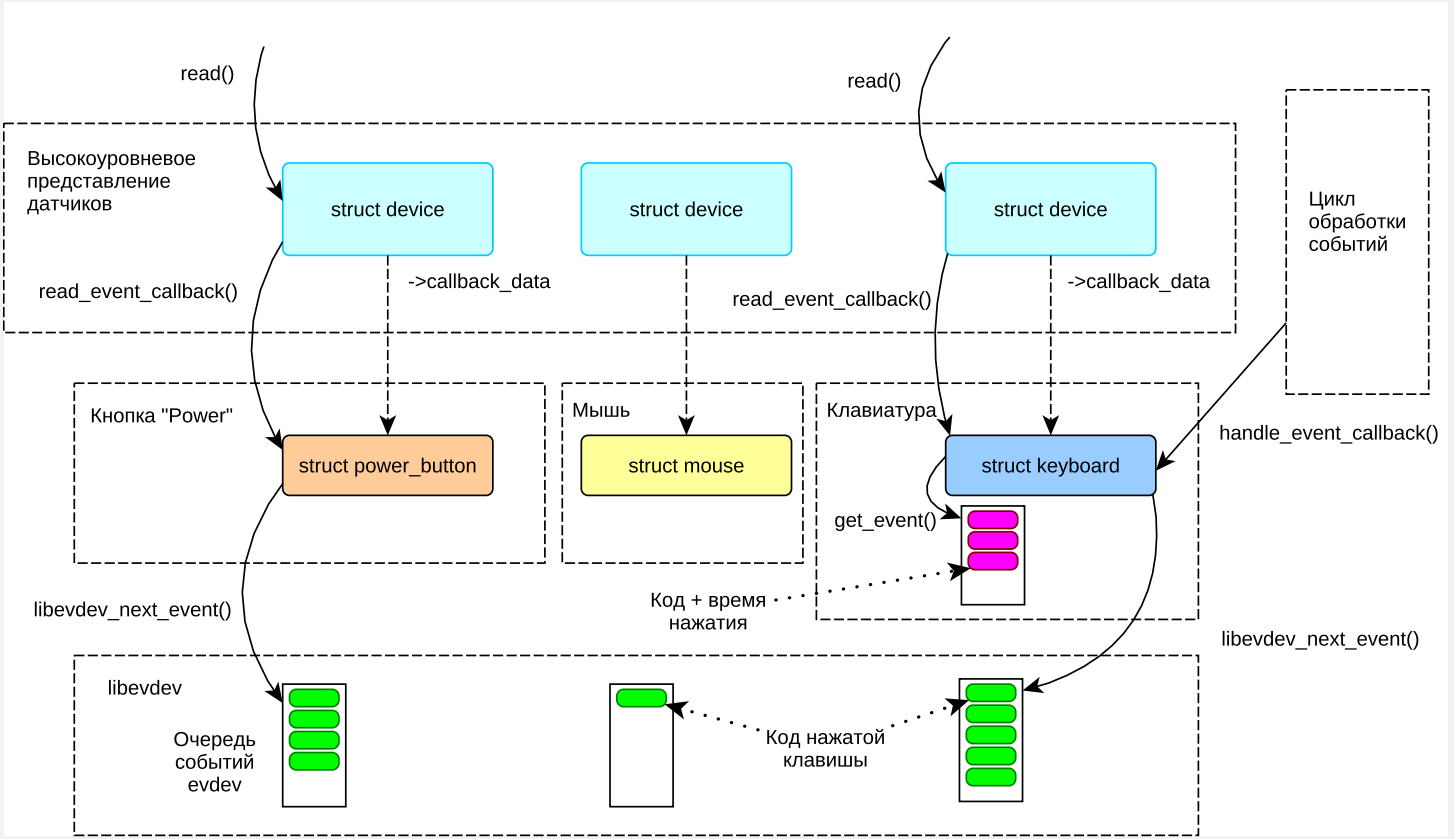
Сканирование

```
static int scan_devices(struct avl_tree *devices)
{
       int fd;
       struct libevdev *evdev;

       fd = open("/dev/input/event5", O_RDONLY | O_NONBLOCK);
       libevdev_new_from_fd(fd, &evdev);

       if (libevdev_has_event_type(evdev, EV_REL) &&
           libevdev_has_event_type(evdev, EV_ABS)) {
              struct mouse *mouse;

              mouse = calloc(1, sizeof(*mouse));
              mouse->evdev = evdev;

              /* Добавление устройства в словарь,
                 регистрация обратного вызова read_mouse_event. */
              ret = register_device(devices, "mouse", evdev,
                                    NULL, read_mouse_event, mouse);
       }
}
```

Чтение событий evdev

```
static int read_mouse_event(char value[], void *dev)
{
	struct mouse *mouse = dev;
	struct input_event ev;

	while (1) {
		ret = libevdev_next_event(mouse->evdev, LIBEVDEV_READ_FLAG_NORMAL, &ev);
		/* События может не быть в очереди. libevdev передает значение
		errno в коде статуса.*/
		if (ret == -EAGAIN)
			return 0;

		if (ev.type == EV_KEY && ev.code == BTN_LEFT && ev.value == 1) {
			/* Нажата левая кнопка мыши. */
			break;
		} else if (ev.type == EV_KEY && ev.code == BTN_RIGHT && ev.value == 1) {
			/* Нажата правая кнопка мыши. */
			break;
		}
	}
	return 0;
}
```

Задание


read()
read()
Высокоуровневое представление датчиков
struct device
struct device
struct device
Цикл обработки событий
read_event_callback()
->callback_data
read_event_callback()
->callback_data
Кнопка "Power"
Мышь
Клавиатура
struct power_button
struct mouse
struct keyboard
handle_event_callback()
get_event()
libevdev_next_event()
Код + время нажатия
libevdev_next_event()
libevdev
Очередь событий evdev
Код нажатой клавишы