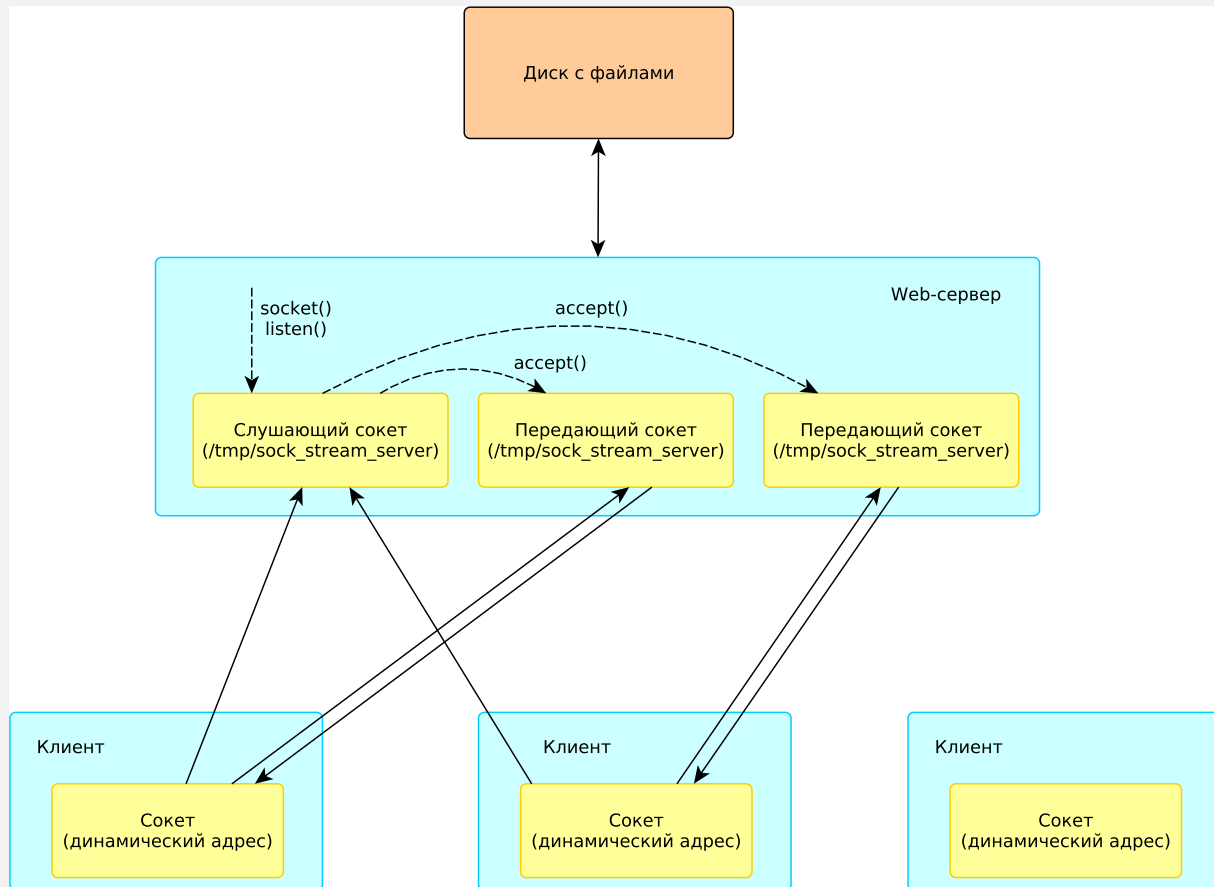


# ПО сетевых устройств

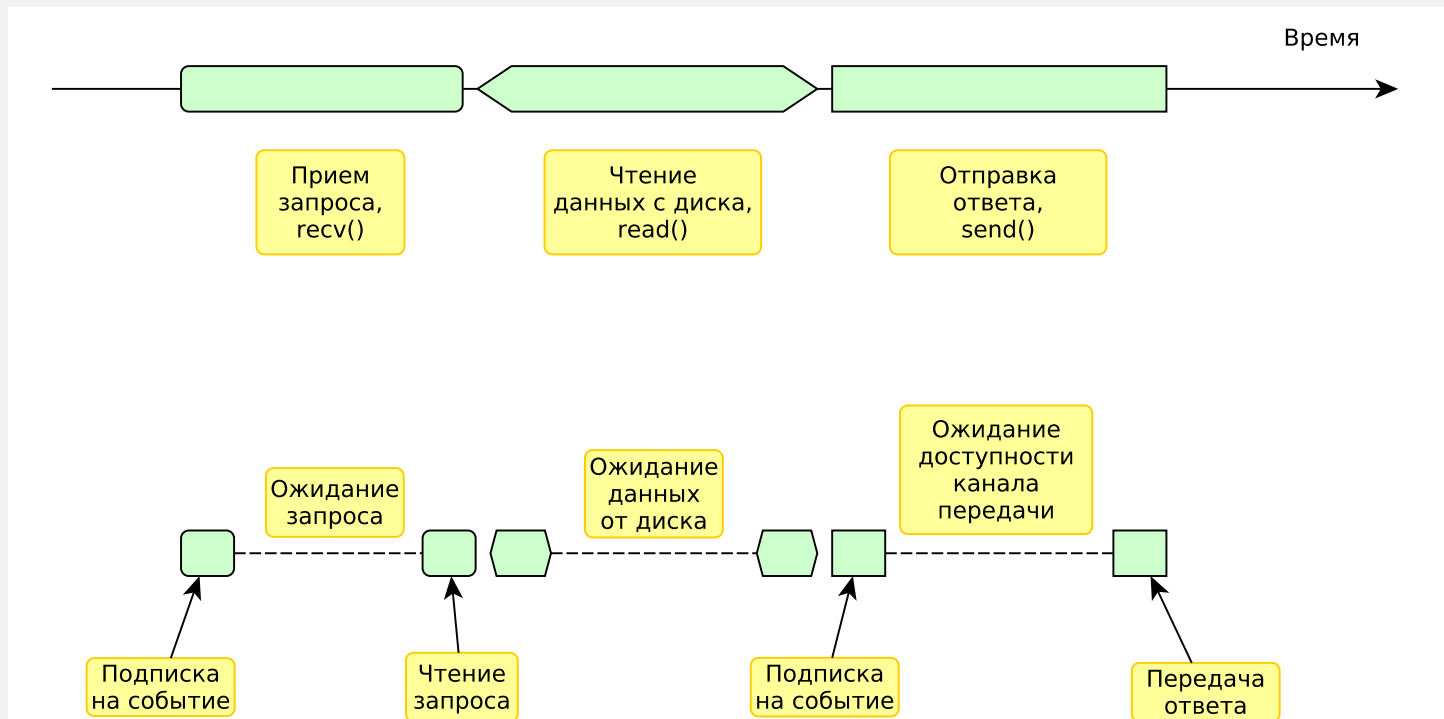
Трещановский Павел Александрович, к.т.н.

08.05.19

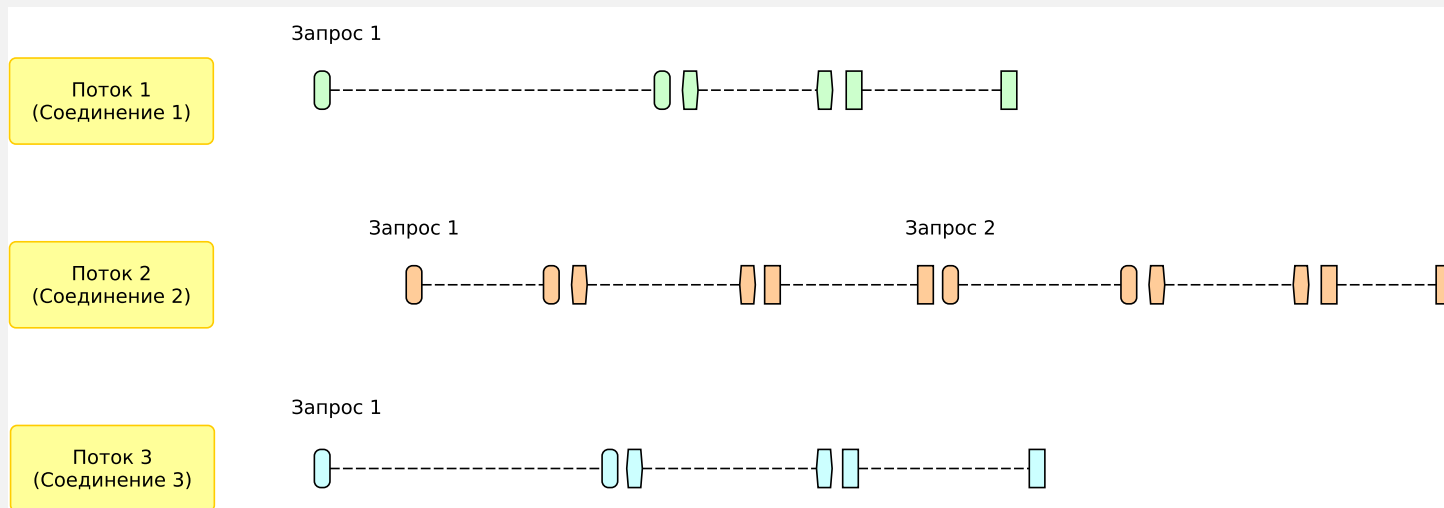
# Web-сервер с поддержкой нескольких соединений



# Обработка одного запроса



# Решение на основе параллельной обработки запросов



- Потоки - процессы с общим адресным пространством (общими данными).
- Создание с помощью `pthread_create` (обертка вокруг `fork`).
- Преимущество - использование нескольких ядер.
- На одном ядре потоки выполняются поочередно.

# Недостатки

- Возможный конфликт при доступе к общим данным:

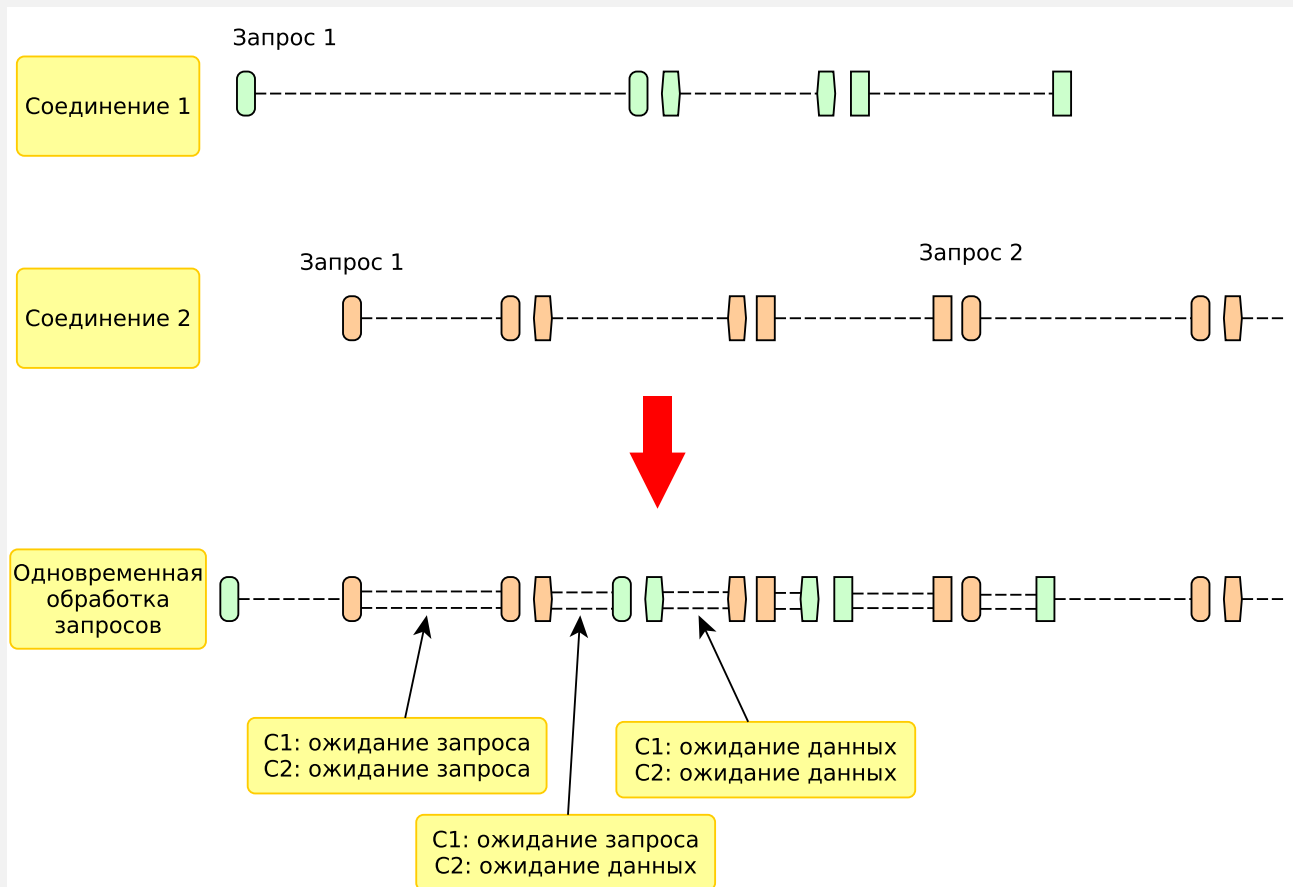
Поток 1	Поток 2	Результат
array[0] = 'a';		'a', '-'
	array[0] = 'b';	'b', '-'
	array[1] = 'b';	'b', 'b'
array[1] = 'a';		'b', 'a' - недопустимая комбинация

- Необходима защита данных с помощью семафоров или мьютексов:

```
semaphore_down();  
array[0] = 'a'; array[1] = 'a';  
semaphore_up();
```

- Многопоточные приложения тяжело разрабатывать и отлаживать.
- Если большую часть времени поток проводит в ожидании, нет выигрыша от использования нескольких ядер.

# Решение на основе одновременной (concurrent) обработки запросов



# Системный вызов `select`

```
int select(int nfds, fd_set *readfds, fd_set *, fd_set *, struct timeval *timeout);
```

- `select` выполняет подписку на события и ожидание.
- `readfds` - множество файловых дескрипторов (от файлов, сокетов и др.).
- `select` спит до тех пор, пока хотя бы один из дескрипторов не будет *готов к чтению*. Готовность к чтению означает, что последующий `read` (или `recv`) вернет данные без ожидания.
- После возвращения из `select` множество `readfds` содержит только те дескрипторы, которые готовы к чтению. Код возврата - общее количество дескрипторов, готовых к чтению.
- Аргумент `timeout` задает максимальное время ожидания готовности. Если это время истекает, `select` возвращает 0.
- Если `timeout` установлен в `NULL`, `select` ждет готовности неограниченное время.

# Работа с множеством файловых дескрипторов

## fd\_set

- fd\_set - битовая маска.

- Задание множества:

```
fd_set fds;  
int fd1 = 2, fd2 = 5;  
FD_ZERO(&fds);           ...000000002  
FD_SET(fd1, &fds);       ...000001002  
FD_SET(fd2, &fds);       ...001001002
```

- Проверка принадлежности множеству:

```
int fd1 = 2;  
if (FD_ISSET(fd1, &fds)) {  
    /* Если принадлежит. */  
} else {  
    /* Если не принадлежит. */  
}
```

- Аргумент nfdс должен быть равен  $(\max(\text{fd1}, \text{fd2}, \dots, \text{fdn}) + 1)$ .



# Работа со структурой struct timeval

## ■ Определение

```
struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;     /* microseconds */
};
```

## ■ Получение текущего времени:

```
struct timeval tv;
gettimeofday(&tv, NULL);
```

## ■ Арифметика:

```
struct timeval tv1, tv2, newtv;
timeradd(&tv1, &tv2, &newtv); /* newtv = tv1 + tv2 */
timersub(&tv1, &tv2, &newtv); /* newtv = tv1 - tv2 */
if (timercmp(&tv1, &tv2, <) {
    /* tv1 < tv2 */
} else {
    /* tv1 >= tv2 */
}
```

# Простой цикл обработки событий (event loop)

```
int listen_fd, sock_fd;
int max_fd, ret;
listen_fd = socket(...);
sock_fd = accept(listen_fd,...);

while (1) {
    struct fd_set fds;
    struct timeval timeout = {.tv_sec = 1};

    FD_ZERO(&fds);
    FD_SET(listen_fd, &fds);
    FD_SET(sock_fd, &fds);
    max_fd = listen_fd > sock_fd ? listen_fd : sock_fd;
    ret = select(max_fd + 1, &fds, NULL, NULL, &timeout);
    if (ret == 0) {
        /* Обработка таймаута. */
    } else if (FD_ISSET(listen_fd, &fds)) {
        /* Прием нового соединения. */
    } else if (FD_ISSET(sock_fd, &fds)) {
        /* Обработка запроса. */
    }
}
```

# Обобщенный цикл обработки событий (event loop)

```
while (1) {
    struct fd_set fds;
    struct timeval timeout;

    FD_ZERO(&fds);
    for (...) { /* Перебор всех сокетов */
        FD_SET(sock_fd, &fds);
        max_fd = sock_fd > max_fd ? sock_fd : max_fd;

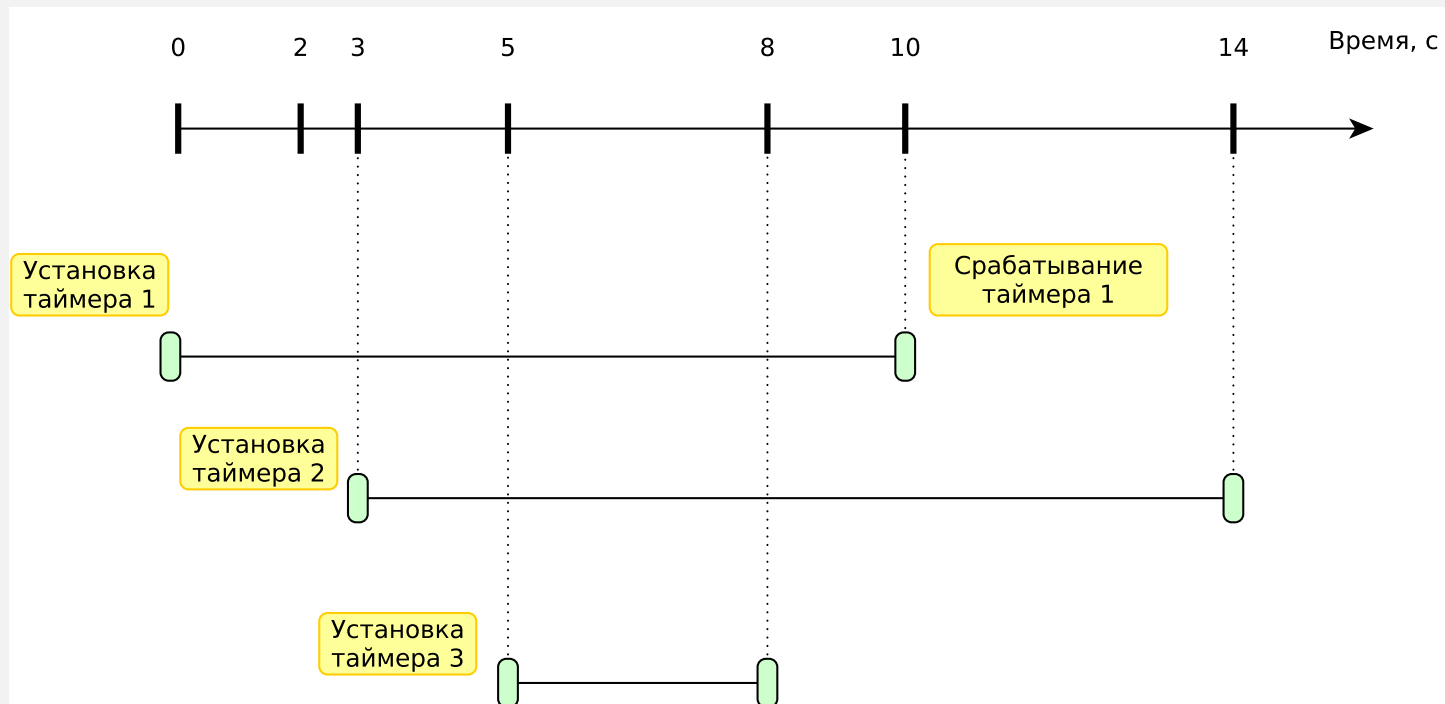
        ... /* Модификация timeout */
    }
    ret = select(max_fd + 1, &fds, NULL, NULL, &timeout);
    if (ret == 0) {
        /* Обработка таймаута. */
    } else {
        for (...) { /* Перебор всех сокетов */
            if (FD_ISSET(sock_fd, &fds)) {
                /* Обработка нового соединения или запроса. */
            }
        }
    }
}
```

# Блокирующий и неблокирующие вызовы

- `select` должен быть единственной функцией в приложении, которой разрешено спать.
- Готовность дескриптора к чтению гарантирует, что 1 последующий вызов `read` вернется без блокирования процесса. Что делать, если требуется несколько вызовов (например, приняли сразу несколько пакетов)?
- Флаг `MSG_DONTWAIT` делает вызовы `recv` и `send` неблокирующими.
- Если данных нет, возвращается ошибка `EAGAIN` без блокировки:

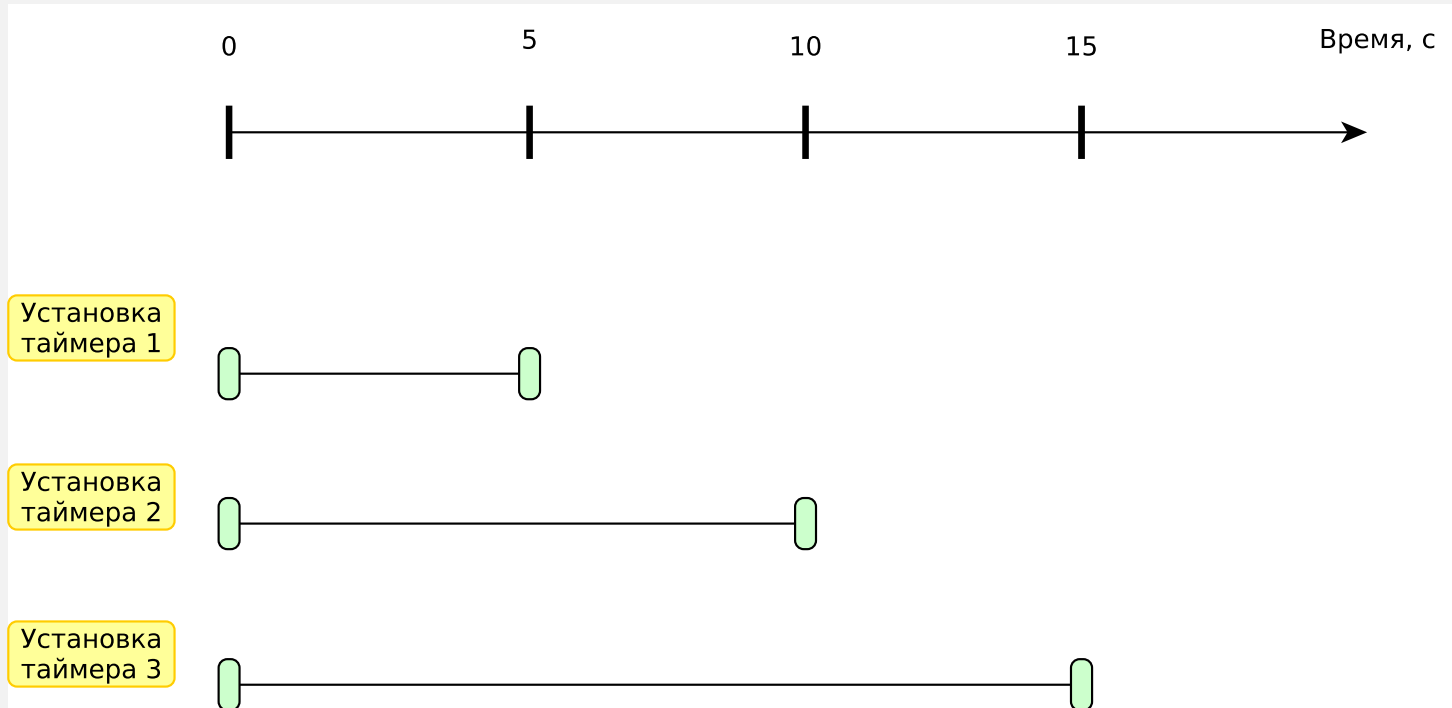
```
while (1) {
    ret = recv(sock_fd, buf, len, MSG_DONTWAIT);
    if (ret < 0 && errno == EAGAIN) {
        /* Нет данных */
        break;
    } else if (ret < 0) {
        /* Другая ошибка */
        return -1;
    }
}
```

# Определение таймаута при наличии нескольких таймеров



$$timeout = \min(T_1 - Current, T_2 - Current, \dots, T_n - Current)$$

# Обработка таймаута при наличии нескольких таймеров



# Замечания по таймерам

- `select` спит в течение времени, не меньшего, чем `timeout`. Но может спать дольше!
- После таймаута надо проверять все таймеры, а не только самый ранний.
- Вычислять новое значение таймаута надо от текущего момента, а не от времени срабатывания предыдущего таймера.

# Упражнения

- Разработать приложение копирующее данные с файлового дескриптора 0 в файловый дескриптор 1. Ожидание новых данных должно быть реализовано с помощью функции `select`. Дескриптор 0 должен быть переведен в неблокирующий режим помощью функции `fcntl`. Пример для дескриптора `fd`:

```
include <fcntl.h>
```

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);
```

- Разработать приложение, реализующее 10 одновременно работающих таймеров. Таймаут для каждого таймера - случайное число в диапазоне от 1 до 10 секунд. При срабатывании таймера необходимо выводить сообщение в терминал с номером этого таймера. Для получения случайного числа использовать функцию `rand`.