

# **Асинхронный режим передачи**

Курс «Информационные сети и телекоммуникации»

весенний семестр 2016 г.

кафедра Управления и информатики НИУ «МЭИ»

# Асинхронный режим технически

- Несколько сокетов могут одновременно:
  - получать или принимать данные;
  - соединяться или ожидать подключений.
- Один «настоящий» поток ОС:
  - Блокирующие сокет в отдельных потоках?
    - Переключения между потоками долгие
    - Сокетов слишком много
  - Принцип:
    - 1) *дождаться* первого события *любого сокета*;
    - 2) обработать событие без блокировок.

# Принцип асинхронного режима

Потенциально блокирующие функции разрешается вызывать, только когда *стало известно*, что этот вызов не приведет к блокировке:

`recv()`, `recvfrom()` —

данные уже приняты по сети, ОС готова их отдать;

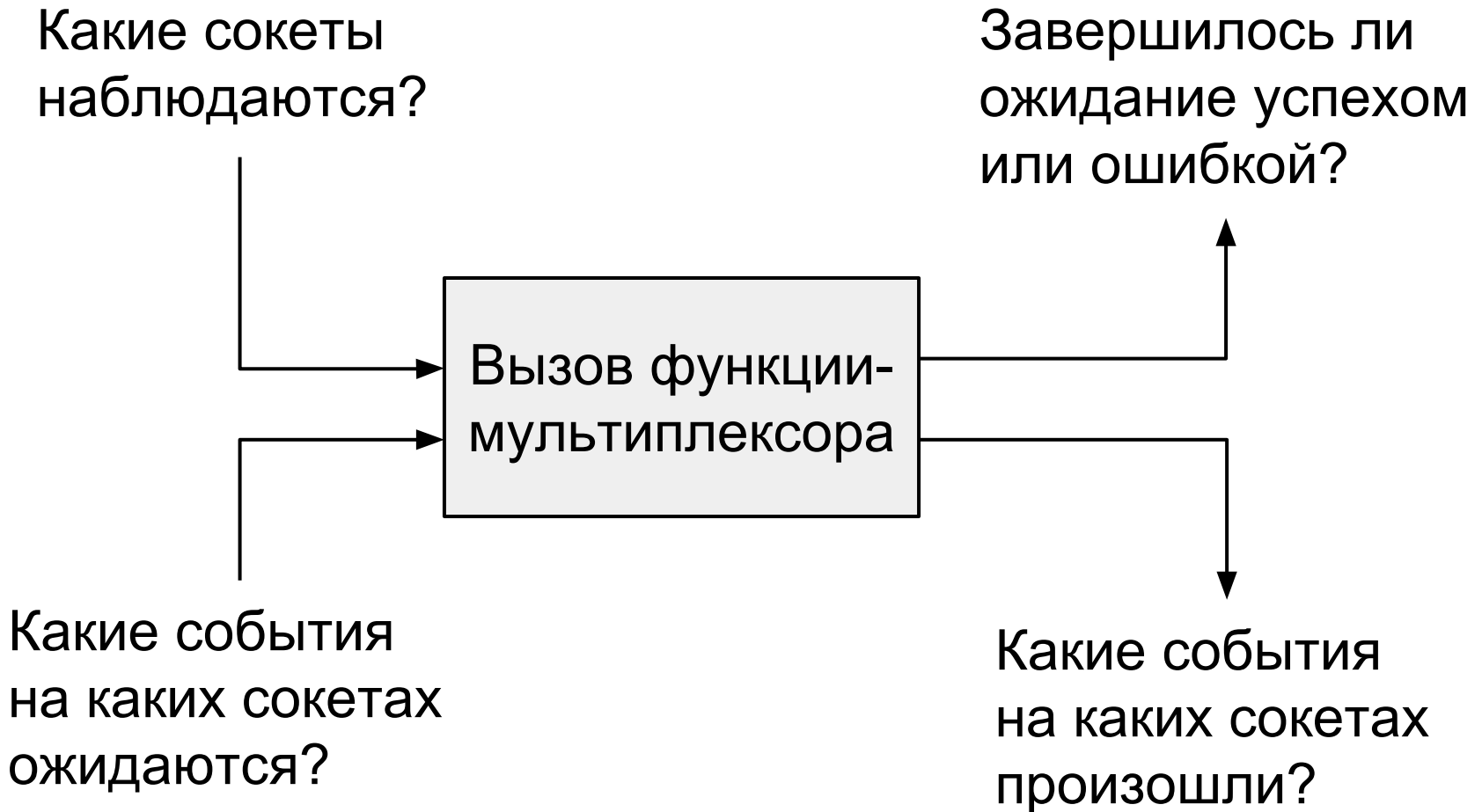
`send()`, `sendto()` —

ОС готова взять у программы данные для передачи;

`accept()` —

уже есть входящее подключение.

# Мультиплексирование ВВОДА и ВЫВОДА



# Ожидание событий

*Windows*: игнорируется.

*\*nix*: наибольшее значение во множествах, + 1 (сокеты в *\*nix* — число).

select(

max\_supplied\_fd,

&waiting\_for\_input,

&waiting\_for\_output,

&expecting\_errors,

nullptr);

Множества сокетов (fd\_set),  
ожидающих:

- 1) возможности приема данных или приема подключения;
- 2) возможности отправки данных или окончания соединения;
- 3) возможных ошибок.

Функция *изменяет* множества (остаются сокеты с событиями).

# Множество сокетов (fd\_set)

- fd = file descriptor ~= socket (= в \*nix)
- Максимальный размер — `FD_SETSIZE` (часто 256).
- fd\_set — набор сокетов (без повторов):

```
fd_set set;  
FD_ZERO(&set);
```

- Операции:
  - `FD_ZERO(&set)` — ОЧИСТИТЬ МНОЖЕСТВО;
  - `FD_SET(socket, &set)` — добавить `socket` в `set`;
  - `FD_CLR(socket, &set)` — удалить `socket` из `set`;
  - `FD_ISSET(socket, &set)` —  
проверить, включен ли `socket` в `set`.

# Ожидание событий

**Успех:** количество сокетов, где произошли события;

**ошибка:** `SOCKET_ERROR` (Windows) или `-1` (\*nix).

По истечении времени ожидания — 0.

↓

```
int result = select(
    socket_count,
    &waiting_for_input,
    &waiting_for_output,
    &expecting_errors,
    &timeout);
```

Время ожидания события.

Бесконечно — `nullptr`.

Без ожидания — 0 с. и 0 мкс.

структура `timeval`:

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

секунды → `tv_sec`  
микросекунды → `tv_usec`

# Схема работы

1) Заполнить множества сокетов:

`FD_ZERO()`, `FD_CLR()`, `FD_SET()`

2) Дождаться наступления событий:

`select()`

3) Обработать события:

`FD_ISSET()`

a) чтение: `recv()`, `recvfrom()`, `accept()`;

b) запись: `send()`, `sendto()`;

c) ошибки: проверка кода, `close()`.

4) Перейти к пункту 1).



# Управление состоянием (1)

```
char buffer[1536];
size_t total = 123;
size_t received = 0;
while (received < total)
{
    size_t left = total - received;
    auto now_received = recv(
        channel, buffer + received, left, 0);
    received += now_received;
}
```

НОВОЕ **состояние** = **действие** (**сокет**, старое **состояние**)      $X_{k+1} = f(X_k)$

# Управление состоянием (2)

```
struct Client  
{
```

```
    int channel;  
    char buffer[1536];  
    size_t total;  
    size_t received;
```

}  
Состояние каждого клиента  
(записи о подключенном клиенте)  
независимо от других.

```
};  
vector<Client> clients;
```

```
...  
select(...);
```

← Нужно определить, для каких  
клиентов произошли события,  
и обработать их по очереди.

```
...  
recv(  
    clients[i].channel,  
    clients[i].buffer + clients[i].received,  
    clients[i].total - clients[i].received,  
    0);
```

# Управление состоянием (3)

```
fd_set readers, writers, failed;  
int max_supplied_id = 0;
```

```
FD_CLR(&readers);  
FD_CLR(&writers);  
FD_CLR(&failed);
```

Готовность принять новые данные от клиента нужна всегда: или будет дослан остаток сообщения, или придет начало нового.

```
for (Client& client : clients) {  
    FD_SET(client.channel, &readers);  
    FD_SET(client.channel, &failed);  
    FD_SET(client.channel, &writers);  
    max_supplied_id = max(max_supplied_id, client.channel);  
}
```

Проверка на ошибки нужна всегда.

Обычно не нужно, т. к. очередь отправки зачастую не занята.

# Управление состоянием (4)

```
int result = select(
    max_supplied_id + 1, &readers, &writers, &failed, 0);
if (result < 0) { /* ошибка */ }
for (Client& client : clients) {
    if (FD_ISSET(client.channel, &readers)) {
        recv(client.channel, ...);
    }
    if (FD_ISSET(client.channel, &writers)) {
        send(client.channel, ...);
    }
    if (FD_ISSET(client.channel, &failed)) {
        // Обработать ошибку.
    }
}
```

# Недопустимость блокировки

*Что произойдет, если вызвать функцию, обращение к которой не разрешалось?*

*(Сокет не был готов к чтению, но вызвана `recv()`.)*

- Режим неблокирующий — блокировки не произойдет.
- Это ошибка:
  - возвращаемое значение — `SOCKET_ERROR` или `(-1)`.
- У ошибки есть код:
  - Windows: `WSAEWOULDBLOCK`;
  - \*nix: `EWOULDBLOCK` или `EAGAIN`.
- Работу с сокетом можно продолжить.

# Переход в асинхронный режим

- Действует на сокеты по отдельности.
- Можно отменить (вызов с `yes == 0`).

Успех: 0,  
ошибка: `SOCKET_ERROR` (Windows) или `-1` (\*nix).

```
int yes = 1;
```

Windows: `ioctlsocket`

```
int result = ioctl(
```


```
socket, FIONBIO, &yes);
```

# Асинхронный прием подключений

- Синхронный режим:

```
transmitter = accept(acceptor, ...);
```

Блокировка  
до первого  
подключения.



- Асинхронный режим:

- Начать ожидание подключений:

```
listen(acceptor, ...);
```

- Дождаться события:

```
FD_SET(acceptor, &waiting_for_input);
```

```
select(..., &waiting_for_input, ...);
```

- Принять подключение:

```
if (FD_ISSET(acceptor, &waiting_for_input))
```

```
    transmitter = accept(acceptor, ...);
```

# Ожидание подключения

Успех: 0,  
ошибка: SOCKET\_ERROR (Windows) или -1 (\*nix).

↓

```
int result = listen(  
    listener, backlog);
```

Сокет-слушатель

↑

Размер очереди входящих подключений  
(накапливаются между вызовами `accept()`).

`SOMAXCONN` — «сделать размер наибольшим».



# Диагностика ошибок

- Несколько сокетов работают в одном потоке.
- Значение `errno` или `WSAGetLastError()` общие для потока.

Функция позволяет получить о сокете разные полезные сведения.  
Здесь: получение **и сброс** кода ошибки.

`int error_code;`

`getsockopt(socket, SO_ERROR, &error_code);`

Примет значение  
кода ошибки.