

Углубленное программирование на C/C++

Лекция №5



Илья Санеев

Некоторые вопросы проектирования сетевых приложений



Базовые функции (linux glibc)

Архитектура "one client - one process" (apache)

Архитектура с мультиплексированием (nginx)

Архитектура в асинхронном стиле (asio)

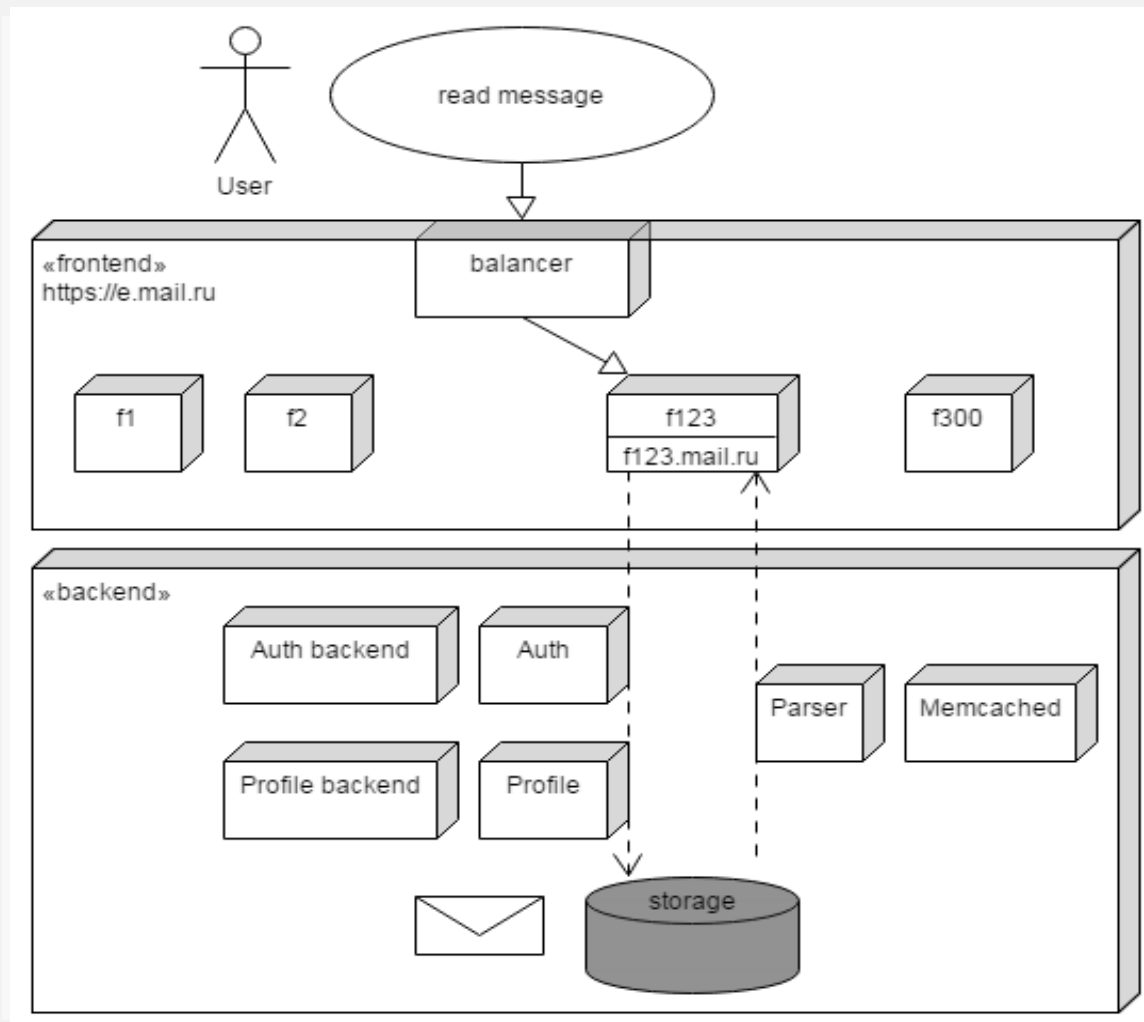
Где (используется сеть)?



- браузер <---> http-сервер (соц сети, почта, вообще все)
- игровой клиент <---> игровой сервер (CS, SC, Q3 e.t.c.)
- база данных <---> клиент БД (интернет-магазин)
- ...
- поисковые роботы
- вирусы, антивирусы,...
- банкоматы, модемы ...

PS: прокси! нынче каждый инженер должен уметь написать http-прокси.

Пример: чтение письма в почте



Почему не сделать одну программу, которая примет запрос и в ответ отдаст письмо?

1) удобнее поддерживать (авторизация - отдельно, парсинг письма - отдельно)

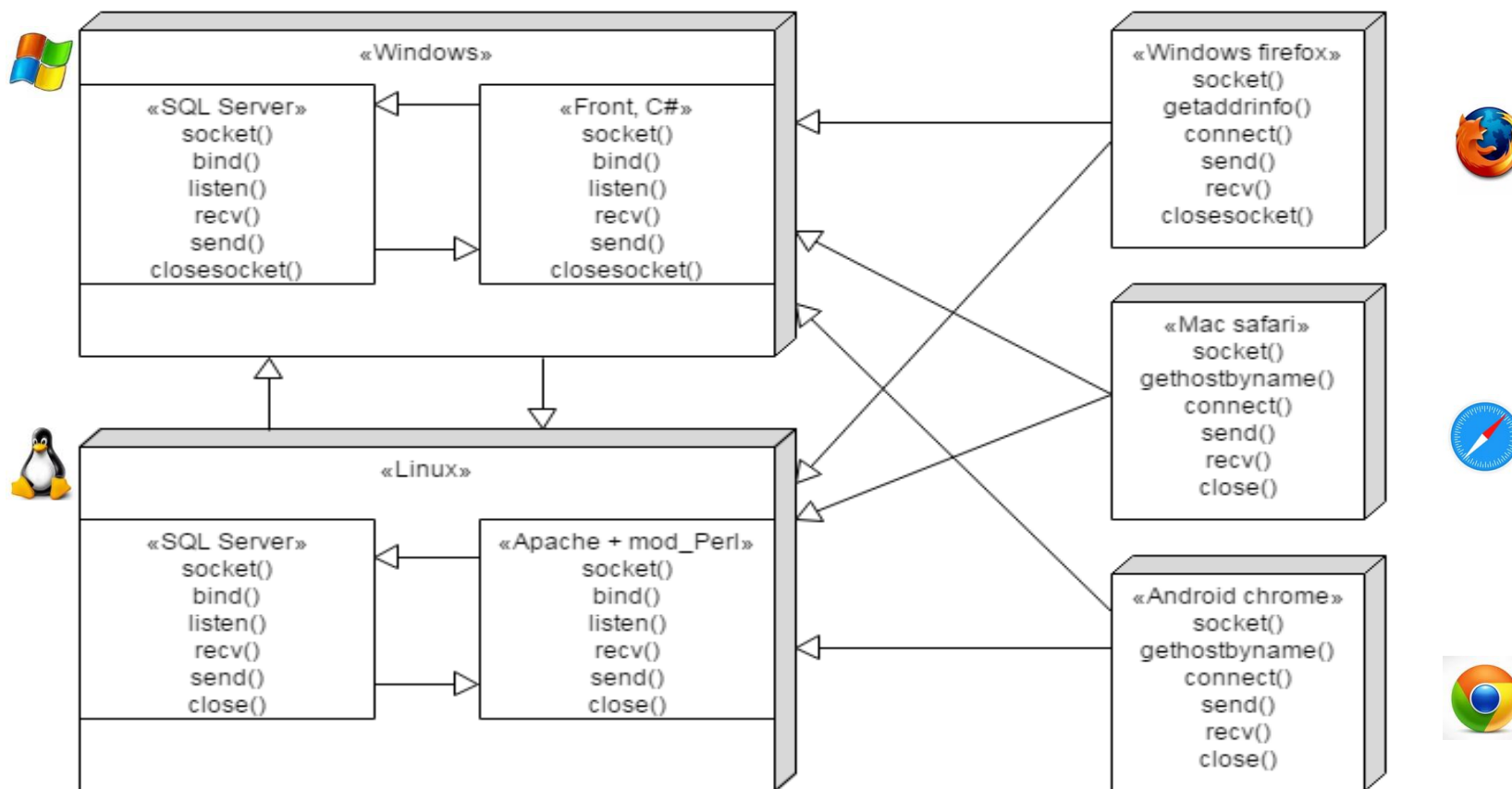
2) уменьшение взаимного влияния

3) RAM для всех не хватит

Итог: внутри крупных сервисов реально много сети!

Почему (используется сеть)?

Универсальность и “простота”



Базовые функции: socket()



```
#include <sys/socket.h>
```

```
int socket(int domain, int type, int protocol);
```

domain - Protocol Family - набор протоколов (сетевой уровень)

- PF_UNIX // unix-сокеты, локальный ipc внутри системы
- PF_INET/PF_INET6 // IPv4/IPv6 сокет
- PF_PACKET // низкоуровневый интерфейс (linux only)

type - тип коммуникации

- SOCK_STREAM // гарантирует доставку, последовательный протокол
- SOCK_DGRAM // UDP
- SOCK_RAW // Raw-сокеты, низкий уровень (доступ к tcp/ip-пакетам)

protocol - конкретный протокол (транспортный уровень)

- /etc/protocols // список возможных
- 0 // использовать протокол по умолчанию для выбранного семейства
- IPPROTO_TCP // можно задать явно

Базовые функции: socket()



```
// TCP
int sd = socket( /*Protocol Family*/ PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sd <= 0)
    throw std::runtime_error(std::string(strerror(errno)));

// UDP
int sd = socket( /*Protocol Family*/ PF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sd <= 0)
    throw std::runtime_error(std::string(strerror(errno)));
```

Базовые функции: connect()



Создали сокет, теперь можно “коннектиться”. Но куда?
Утилиты: ping, host, dig, nslookup, telnet

e.mail.ru vs e.mail.ri

```
host e.mail.ru
```

```
e.mail.ru has address 94.100.180.215
```

```
e.mail.ru has address 94.100.180.216
```

```
telnet e.mail.ru 443          # OK!
```

```
host e.mail.ri
```

```
Host e.mail.ri not found: 3(NXDOMAIN)
```

```
telnet e.mail.ri 443          # telnet: e.mail.ri: Name or service not known
```

```
dig -x 217.69.139.215 +short
```


Базовые функции: connect()



```
#include <sys/socket.h> // connect, AF_INET
#include <netdb.h>       // gethostbyname
#include <netinet/in.h>  // struct sockaddr_in

// STREAM and DGRAM
int connect(int sockfd,
            const struct sockaddr *serv_addr,
            socklen_t addrlen);

struct hostent *gethostbyname(const char *name);
```

Базовые функции: connect()



```
void Client::connect(const std::string &host /*e.mail.ru*/, int port) {  
    int sd = socket(PF_INET, SOCK_STREAM, 0);  
    struct hostent* hp = gethostbyname(host.c_str());  
  
    struct sockaddr_in addr;           // КУДА ИДЕМ?  
  
    addr.sin_family = AF_INET;         // Address Family only AF_INET !  
    addr.sin_port = htons(port);       // big/little endian  
    memcpy(&addr.sin_addr, hp->h_addr, hp->h_length);  
    int connected = connect(sd, (struct sockaddr*)&addr, sizeof(addr));  
    if (connected != 0)  
        throw runtime_error(std::string(strerror(errno)));  
}
```

Базовые функции: send()



```
#include <sys/socket.h>
```

```
ssize_t send(int sd, const void *msg, size_t len, int flags);
```

```
ssize_t sendto(int sd,  
               const void *msg,  
               size_t len,  
               int flags,  
               struct sockaddr *to,  
               socklen_t elen);
```

```
ssize_t sendmsg(int sd, const struct msghdr *msg, int flags);
```

Базовые функции: send()



```
// плохо!
```

```
std::string msg = "hello!";
```

```
ssize_t n = send(sd, msg.data(), msg.size(), 0);
```

```
// хорошо!
```

```
void Client::send(int sd, const std::string &msg) {
```

```
    size_t left = msg.size();
```

```
    ssize_t sent = 0;
```

```
    while (left > 0) {
```

```
        sent = ::send(sd, msg.data() + sent, msg.size() - sent, flags);
```

```
        if (-1 == sent)
```

```
            throw std::runtime_error(std::string(strerror(errno)));
```

```
        left -= sent;
```

```
    }
```

```
}
```

Базовые функции: recv()



```
#include <sys/socket.h>
int recv(int sd, void *buf, size_t len, int flags);
```

Базовые функции: recv()



```
std::string Client::recv(int sd) {  
    char buf[128]; // Почему именно 128?  
  
    int n = ::recv(sd, buf, sizeof(buf), /*flags*/0);  
    if (-1 == n)  
        throw runtime_error(std::string(strerror(errno)));  
  
    return std::string(ret, ret + n);  
}
```

Проблема: как узнать, сколько байт нужно читать?

Данные есть в ядре, но их надо доставить в клиентский код.

Решение: зависит от ситуации (от протокола).

Базовые функции: close()



```
#include <unistd.h>    // закрывает и файлы, и сокеты...  
int close(int sd);
```

class Client (tcp client)



```
class Client {
    int m_Sd;
public:
    Client() : m_Sd(-1) {}
    ~Client() { if (m_Sd > 0) close(m_Sd); }
public:
    void connect(const std::string &host, int port) throw (exception);
    void send(const std::string &s) throw (exception);
    std::string recv() throw (exception);
};
```


Пример использования: telnet



```
telnet park.mail.ru 80
```

```
Trying 185.5.138.251... // resolv
```

```
Connected to park.mail.ru. // connect
```

```
Escape character is '^['.
```

```
123 // send
```

```
HTTP/1.1 400 Bad Request // recv
```

```
Server: nginx
```

```
Content-Type: text/html
```

```
Content-Length: 166
```

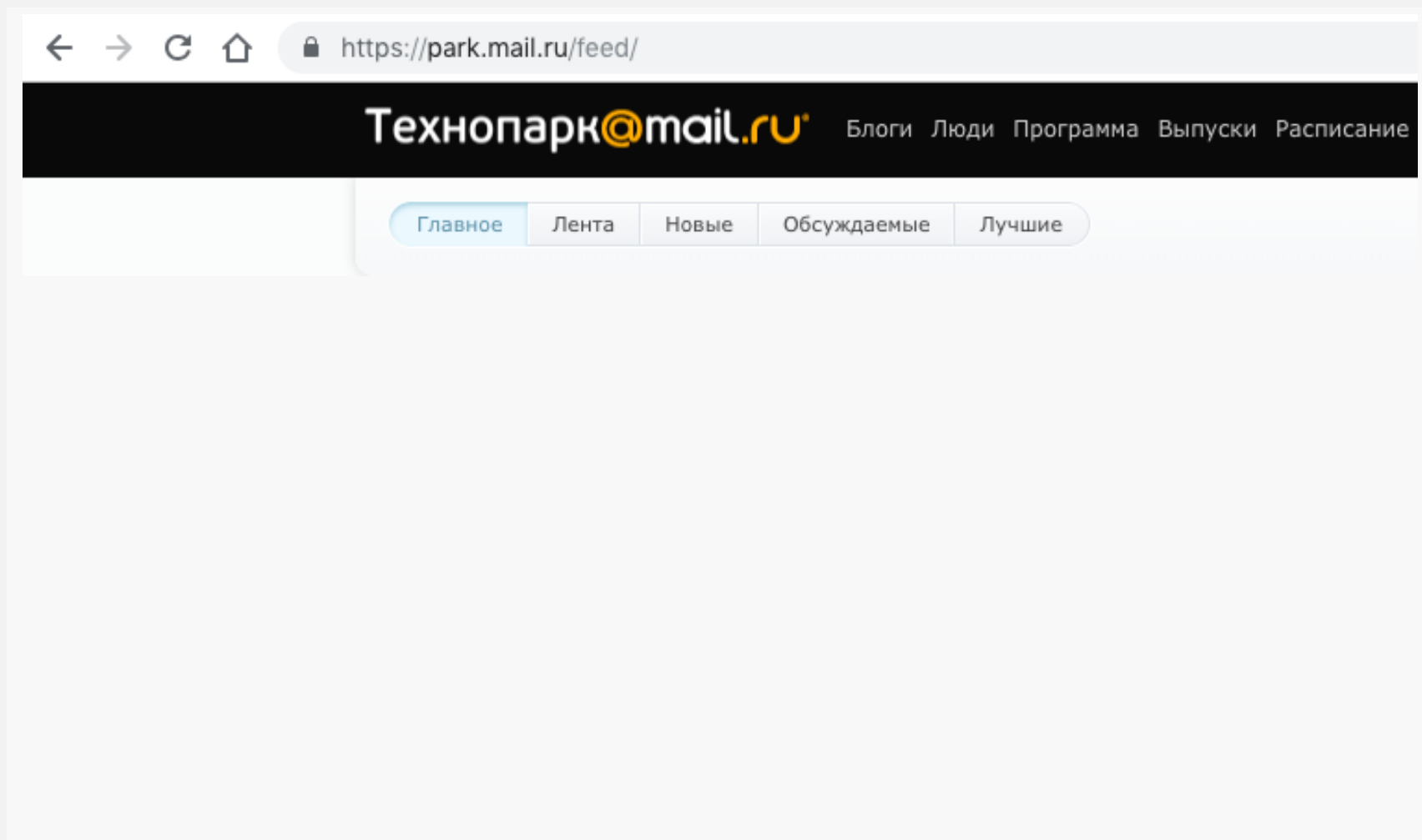
```
<html>
```

```
<head><title>400 Bad Request</title></head>
```

```
</html>
```

```
Connection closed by foreign host. // close
```

Пример использования: http-запрос



Http-клиент: query



```
GET /people/?q=i.saneev HTTP/1.1\r\n
Host: park.mail.ru\r\n
\r\n
```

Или:

```
std::string q =
    "GET /people/?q=i.saneev HTTP/1.1\r\nHost: park.mail.ru\r\n\r\n";
```

Http-клиент: query подробно (curl)



```
curl -v http://park.mail.ru/feed
* Connected to park.mail.ru (185.5.138.251) port 80 (#0)
> GET /feed HTTP/1.1
> User-Agent: curl/7.35.0
> Host: park.mail.ru
>
< HTTP/1.1 302 Moved Temporarily
< Date: Sun, 27 Mar 2016 19:43:12 GMT
< Content-Type: text/html
< Content-Length: 154
< Location: https://park.mail.ru/feed
<
<html>
<head><title>302 Found</title></head>
<body><h1>302 Found</h1><hr>nginx</body></html>
```

Http-клиент: неудачная попытка



```
Client http_client;  
http_client.connect("e.mail.ru", 80);  
http_client.send("GET /inbox HTTP/1.1\r\nHost: e.mail.ru\r\n\r\n");  
std::string response = http_client.recv();
```

#Приняли не все, 128 байт - мало!

Http-клиент: удачная попытка



Попробуем пофиксить:

```
std::string Client::recv() throw (std::exception)
{
    char buf[1024];          // было 128 байт
    ...
}
```

Теперь ОК! Но правильно ли это?

Клиент может:

- прислать еще больше данных
- прислать ответ в 2 пакетах (нужно 2 вызова recv())

Http-клиент: нужен цикл!



```
std::string Client::recv() throw (std::exception) {  
    std::string ret;  
    char buf[1024];  
    while (true) {  
        int n = ::recv(m_Sd, buf, sizeof(buf), /*flags*/0);  
        if (-1 == n) handle_error();  
        if (0 == n) break;  
        ret.append(buf, n);  
    }  
    return ret;  
}
```

#...oooops! Программа заблокировалась!

оказывается, есть блокирующие и неблокирующие сокеты

recv() на блокирующем сокете



```
int n = ::recv(m_Sd, buf, sizeof(buf), /*flags*/0);
```

1. $n > 0$, есть данные;
2. $n = -1$, ошибка чтения;
3. $n = 0$, endpoint закрылся - можно делать `close()` и все сначала.

Следствие: если данных нет и ошибок нет и сервер не закрывает соединение - `recv()` "висит" бесконечно.

Как борются с "блокированием"?

setsockopt(): свойства сокета, пример с установкой таймаута



```
#include <sys/socket.h>
```

```
int getsockopt(int s, int level, int optname, void *val, socklen_t *len);
```

```
int setsockopt(int s, int level, int name, const void *v, socklen_t len);
```

level: SOL_SOCKET - уровень сокета, TCP - уровень tcp, ...

```
void Client::setRcvTimeout(int sec, int microsec) throw (std::exception) {  
    struct timeval tv;  
    tv.tv_sec = sec;  
    tv.tv_usec = microsec;  
    if (setsockopt(m_Sd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) != 0)  
        throw runtime_error("rcvtimeout: " + string(strerror(errno)));  
}
```

Http-клиент: установим таймаут в блокирующий сокет



```
Client http_client;  
http_client.connect("e.mail.ru", 80);  
  
// Теперь, если n == -1 && errno != EAGAIN,  
// значит за отведенный таймаут данные не поступили  
http_client.setRcvTimeout(/*sec*/1, /*microsec*/0);  
http_client.send("GET /inbox HTTP/1.1\r\nHost: e.mail.ru\r\n\r\n");  
std::string response = http_client.recv();
```

OK! Теперь recv() не заблокируется, но осталась одна проблема:
как определить, что сервер отдал все, что хотел?
ведь сервер может "тормозить" больше 1 секунды!

Http-клиент: как правильно?



Используем длину контента + таймауты!

Прикладной протокол содержит длину сообщения:

В http - "Content-Length: 10385"

В memcached - количество "\r\n"

Для реализации нам понадобятся:

1. парсер протокола (получаем длину);
2. сокет с таймаутом.

Http-клиент: длина контента + таймауты - итог



```
std::string Client::recv() throw (std::exception) {  
    std::string ret;  
    char buf[128];  
    while (true) {  
        int n = ::recv(m_Sd, buf, sizeof(buf), 0);  
        if (-1 == n && errno != EAGAIN) throw runtime_error("read");  
        if (0 == n || -1 == n) break;  
        ret.append(buf, n);  
        size_t bytes_left = get_length_with_protocol_http(ret);  
        if (bytes_left == ret.size())    // OK!  
            break;  
    }  
    return ret;  
}
```

Http-клиент: неблокирующий recv()



Иногда лучше использовать неблокирующие сокеты.
Пример: таймаут на connect() - будет дальше.

recv() на неблокирующем сокете:

1. $n > 0$, есть данные;
2. $n = 0$, endpoint закрылся, можно делать close();
3. $n = -1$ && $errno == EAGAIN$, пока что нет данных, но может будут;
4. $n = -1$ && $errno != EAGAIN$, ошибка;

Не блокируются:

- connect (возвратит -1)
- recv/send
- accept (будет позже, для сервера)

fcntl() - обычный способ сделать сокет неблокирующим (unix only)



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

```
#include <unistd.h>
```

```
int fcntl(int fd, int cmd, long arg);
```

```
void Client::setNonBlocked(bool opt) throw (std::exception) {  
    int flags = fcntl(m_Sd, F_GETFL, 0);  
    int new_flags = (opt)? (flags | O_NONBLOCK) : (flags & ~O_NONBLOCK);  
    if (fcntl(m_Sd, F_SETFL, new_flags) == -1)  
        throw runtime_error("nonblocked: " + string(strerror(errno)));  
}
```

Базовые функции: bind()



```
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);

struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(port);
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    throw std::runtime_error("bind: " + std::string(strerror(errno)));
```

Теперь у сокета есть адрес!

Базовые функции: bind() - что такое INADDR_ANY?



```
./simple_server --port 7777
```

```
1. serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

- telnet localhost 7777 # OK
- telnet 127.0.0.1 777 # OK
- telnet 192.168.70.129 7777 # OK

```
2. serv_addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
```

- telnet localhost 7777 # Connection
- telnet 127.0.0.1 7777 # OK
- telnet 192.168.70.129 7777 # Connection refused

```
3. serv_addr.sin_addr.s_addr = inet_addr("192.168.70.129");
```

- telnet localhost 7777 # Connection refused
- telnet 127.0.0.1 777 # Connection refused
- telnet 192.168.70.129 7777 # OK

Базовые функции: listen()



```
#include <sys/socket.h>

int listen(int sd, int backlog); // backlog - используется ядром

void Server::createServerSocket(uint32_t port, uint32_t queue_size) {
    int sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(port);

    bind(sd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    ::listen(sd, listen_queue_size);

    m_Sd = sd;
}
```

Базовые функции: listen()



Теперь к сокету можно сделать connect!

./simple_server --port 7777

Linux:

```
me@ubuntu:~$ netstat -antp | fgrep simple_server
tcp 0 0.0.0.0:7777 0.0.0.0:* LISTEN 15884/simple_server
```

MacOS:

```
netstat -antp tcp | grep LISTEN
tcp4      0      0 *.7777          *.*          LISTEN
tcp6      0      0 *.63957         *.*          LISTEN
```

lsof -nP -i4TCP:7777 или lsof -i :7777

```
simple_se 78289 isaneev 3u IPv4 0x7d47332cda97a0d3 0t0 TCP
*:7777 (LISTEN)
```

Позволяет просматривать список и состояние сокетов в системе:

- TCP/UDP/Unix
- LISTEN, ESTABLISHED, TIMEWAIT e.t.c.

Базовые функции: accept()



```
#include <sys/socket.h>
```

```
// sd = bind() + listen()
```

```
int accept(int sd, struct sockaddr *addr, socklen_t *addrlen);
```

Самый важный вызов для tcp сервера:

- Извлекает первый запрос на соединение из очереди listen;
- Создает новый сокет, возвращает его дескриптор;
- Заполняет информацию о клиенте и кладет ее в addr (с какого порта и адреса пришел клиент?).

Наконец клиент и сервер могут общаться!

Базовые функции: accept()



```
// m_Sd = слушающий сокет, bind() + listen()

std::shared_ptr<Client> Server::accept() throw (std::exception) {
    struct sockaddr_in client;           // Откуда клиент?
    socklen_t cli_len = sizeof(client);

    int cli_sd = ::accept(m_Sd, (struct sockaddr*)&client, &cli_len);

    std::cout << "+client: " << cli_sd
               << ", from: " << int2ip4(client.sin_addr.s_addr)
               << std::endl;
    return std::make_shared<Client>(cli_sd);
}
```

```
+client: 4, from: 127.0.0.1
```

class Server



```
class Server {  
    public:  
        Server() : m_Sd(-1) {}  
        ~Server() { if (m_Sd > 0) close(m_Sd); }  
  
        void createServerSocket(uint32_t port, uint32_t queue_size);  
  
        std::shared_ptr<Client> accept() throw (std::exception);  
  
        void setNonBlocked(bool opt) throw (std::exception);  
    private:  
        int m_Sd;  
};
```

echo-сервер для одного клиента



```
Server serv;
serv.createServerSocket(/*port*/7777, /*backlog*/25);
while(true) {
    std::shared_ptr<Client> client = serv.accept();
    client_work(client);
}

void client_work(std::shared_ptr<Client> client) {
    client->setRcvTimeout(/*sec*/30, /*microsec*/0);
    while (true) try {
        std::string line = client->recv();
        client->send("echo: " + line + "\n");
    }
    catch(const std::exception &e) { return; }
}
```

Клиентский и серверный код: рекомендации



Сеть - это нюансы использования конкретного протокола:

- знать протокол, уметь считать длину сообщения для гесv

Сеть - медленная:

- таймауты - это наше все
- сжатие (gzip в HTTP, меньше данных - больше скорость)

Сеть - ненадежная:

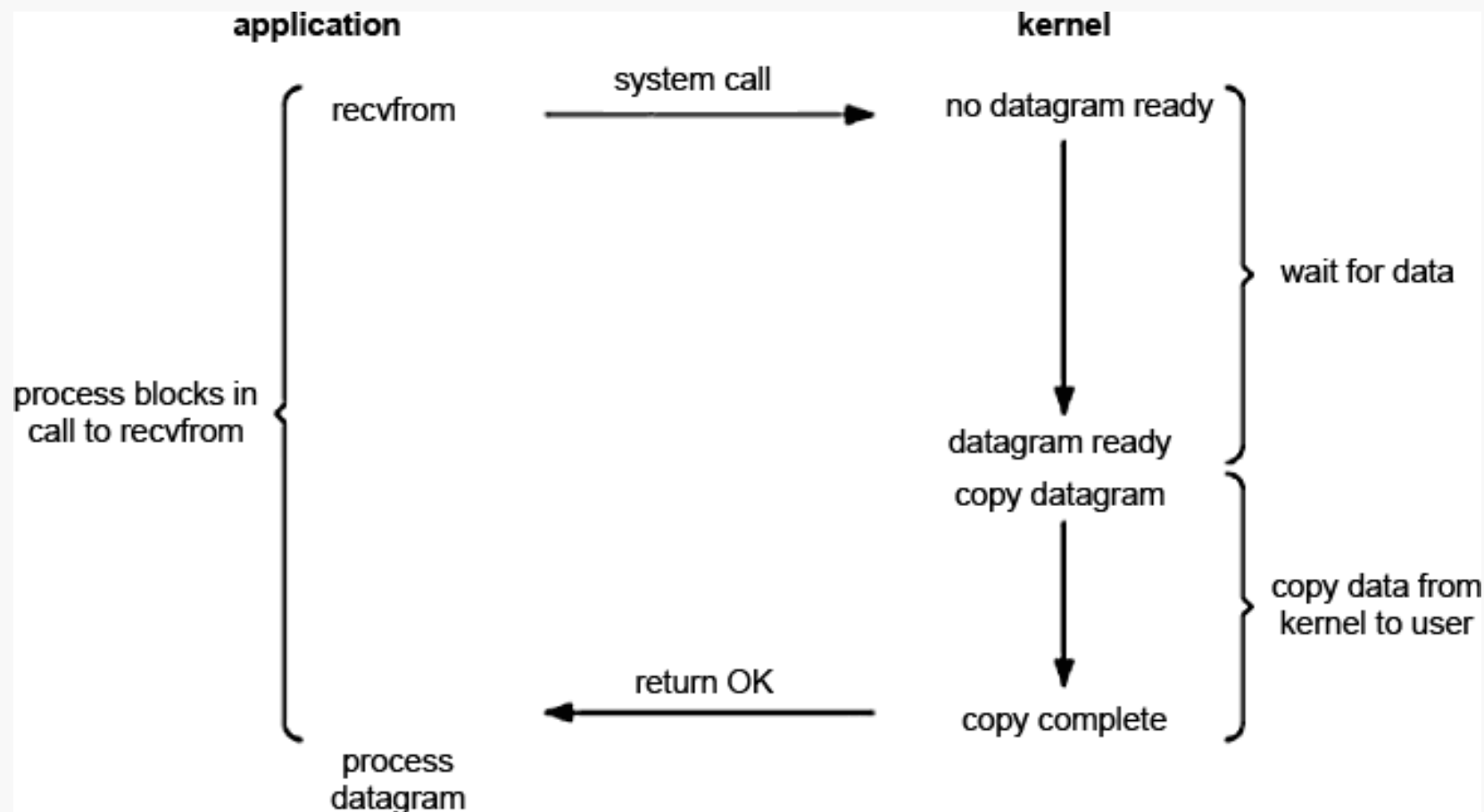
- обязательна проверка get-кодов функций и егпо

Сеть - сильно подвержена влиянию системы:

- не забываем close(), free(), delete()
- держим в уме: нагружая систему по сри, проседает производительность сетевого сервера

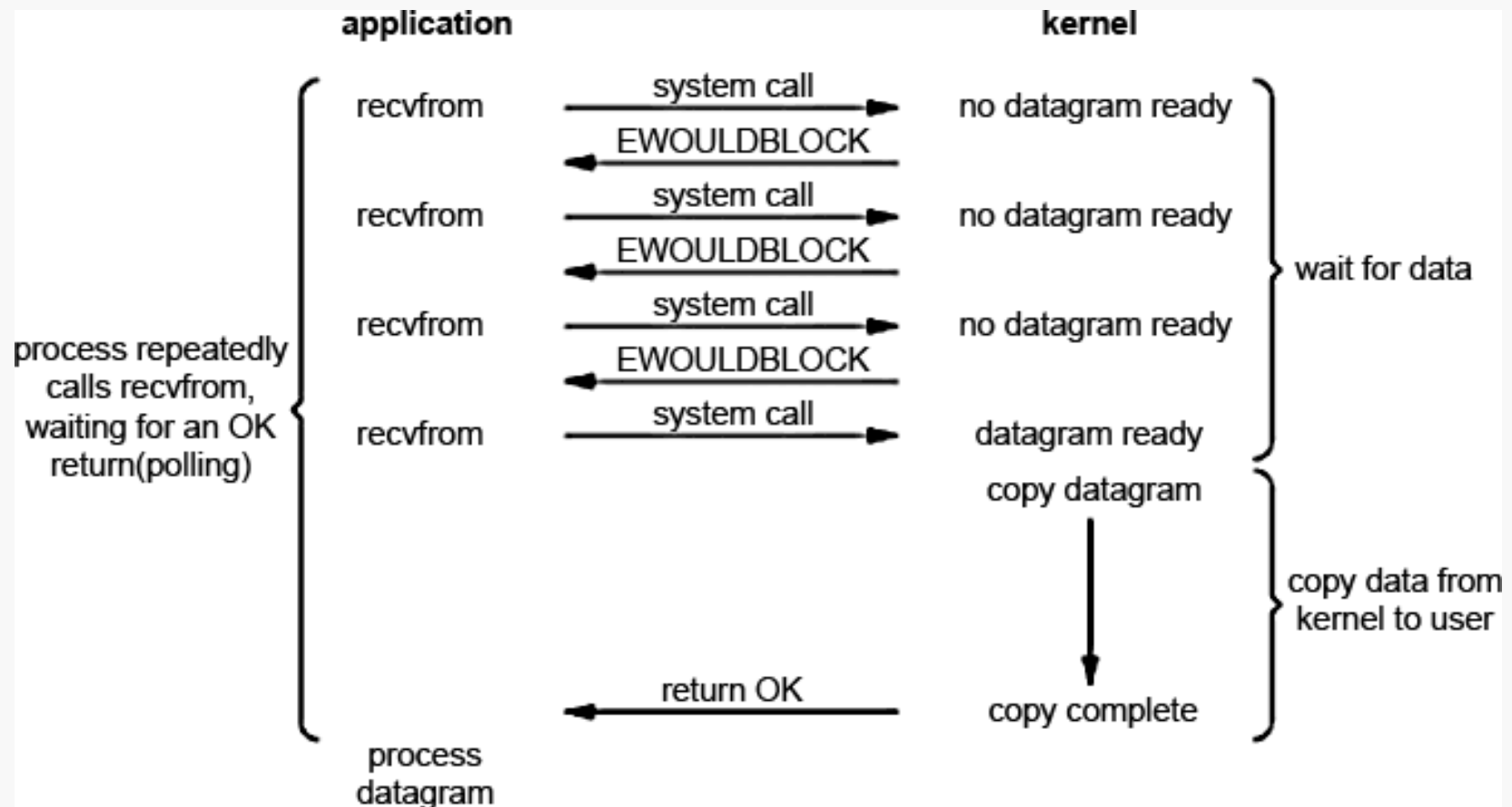
Модели ввода-вывода.

Блокирующий I/O



Модели ввода-вывода.

Неблокирующий I/O



Архитектура: как обрабатывать нескольких клиентов одновременно?



fork()

- копирует процесс, включая таблицу дескрипторов;
- ОС решает, в какой принимать соединения (асепт);
- ОС решает, в какой принимать данные (гесv);
- писать можно в оба дескриптора после форка;
- закрытие в одном оставляет в другом живым!

thread()

- копирует лишь часть процесса;
- дескрипторы остаются общими.

multiplexing

- ничего не копирует, работает в ядре.

Prefork



```
Socket serv;  
serv.createServerSocket(port, 25);  
  
pid_t pid = fork();          // N раз форкаемся  
if (pid > 0) std::cerr << "parent: " << getpid() << std::endl;  
else std::cerr << "child: " << getpid() << std::endl;  
  
while(true) {  
    std::shared_ptr<Client> client = serv.accept();  
    client_work(client);  
}
```

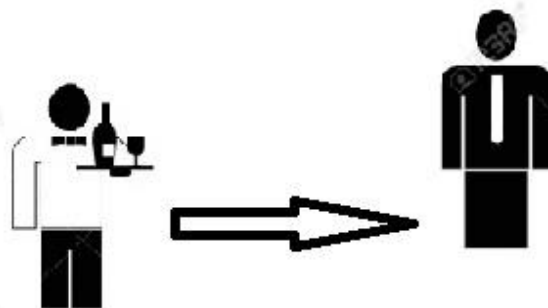
Apache = этот_пример + master-process.

- + Надежно: у каждого воркера - свое адресное пространство;
- + Просто программировать
- Два медленных клиента "убивают" сервер; плюс ресурсы

one client - one process



new/malloc



Простой вариант: fork/thread на каждое соединение

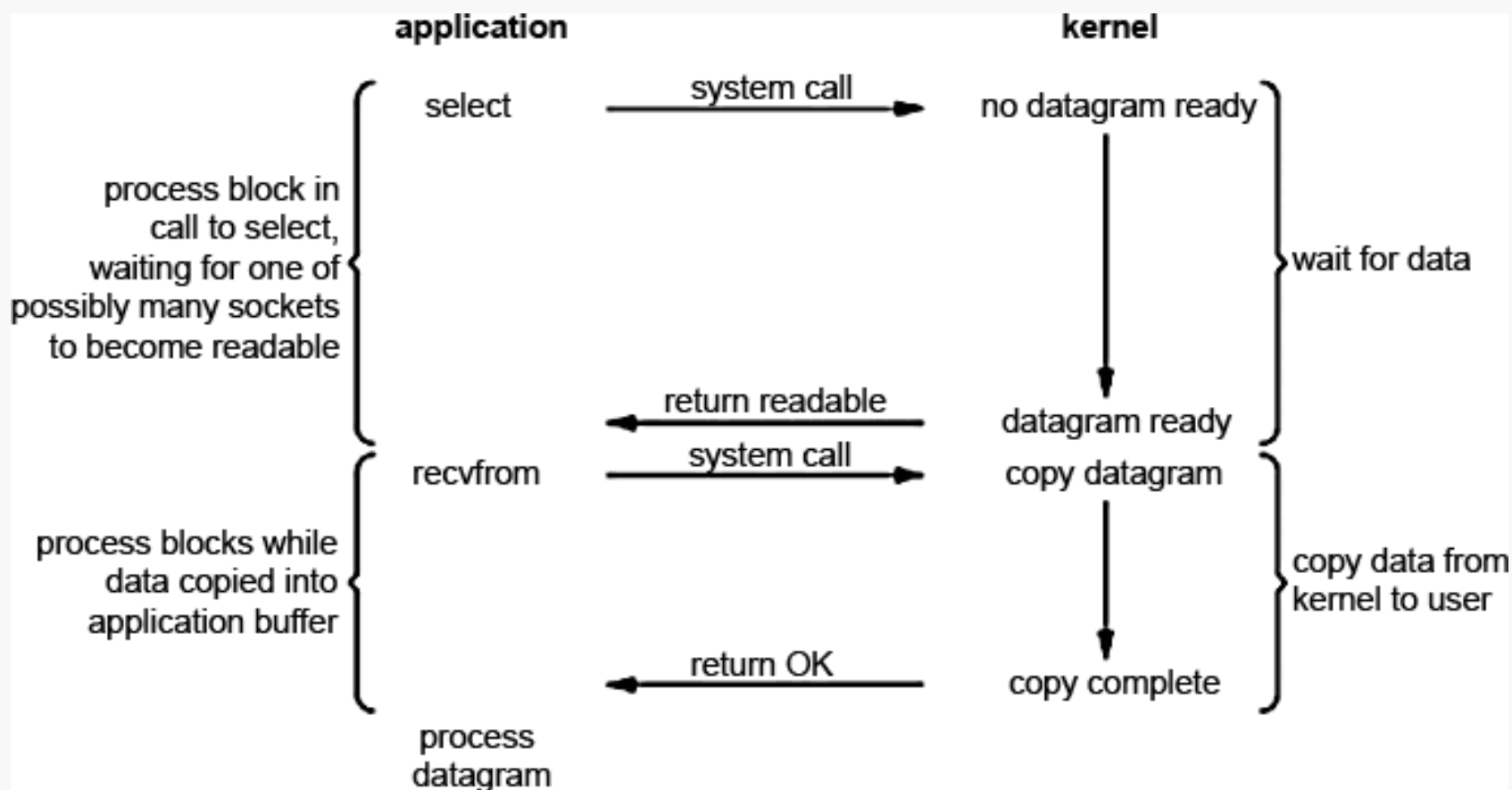


```
Server serv;  
serv.createServerSocket(port, 25);  
  
while(true)  
{  
    std::shared_ptr<Client> client = serv.accept();  
  
    std::thread t (client_work(client)); // или fork() + client_work()  
}
```

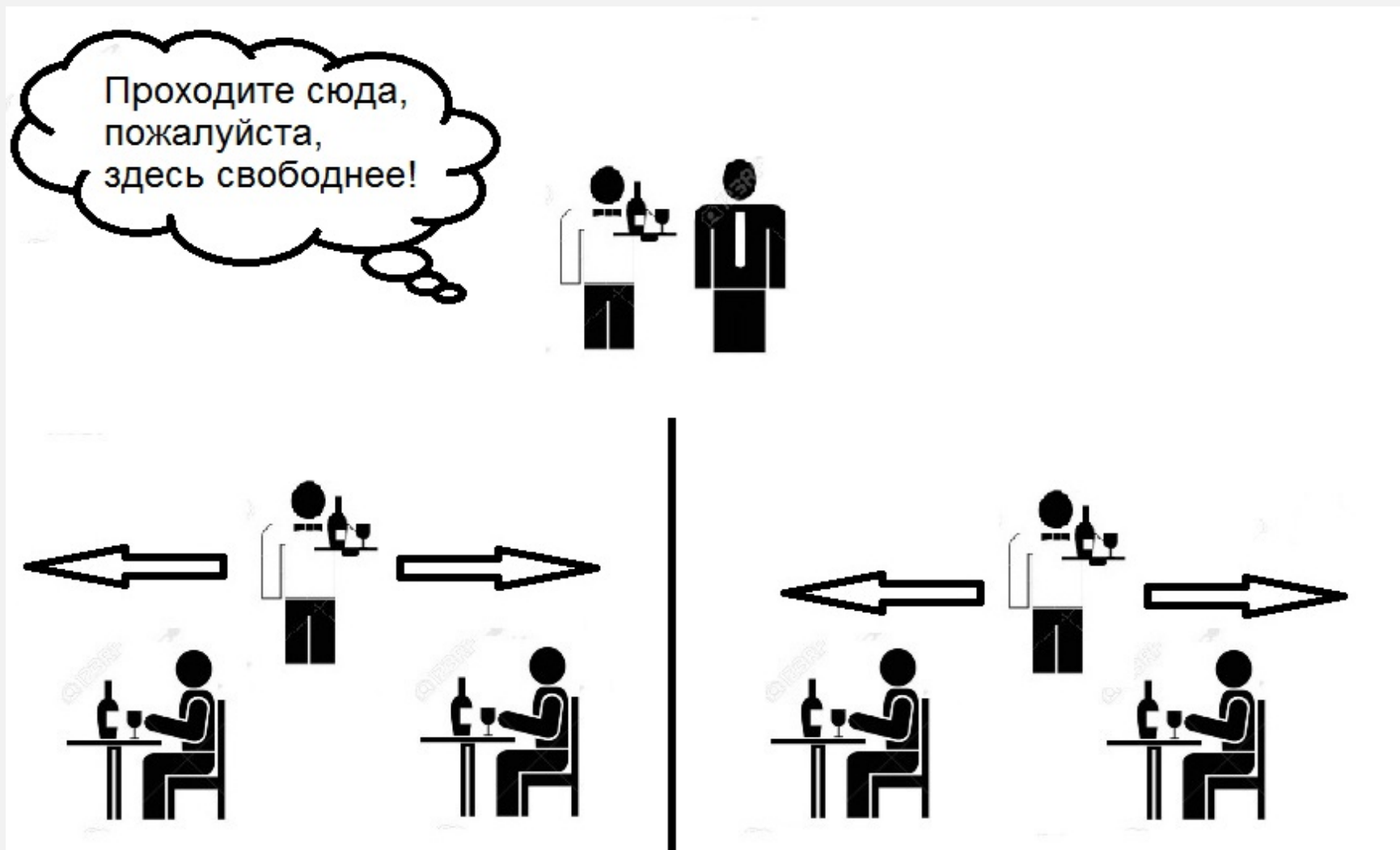
- + Суперпросто реализовать.
- Рост потребления ресурсов не контролируется (fork-бомба).

Модели ввода-вывода.

Мультиплексирование I/O



Мультиплексирование



Мультиплексирование



Это возможность одним потоком обслуживать десятки тысяч клиентов.

Для этого необходимо уметь (быстро) понимать, доступны ли данные.
(Как официант понимает, что клиент готов сделать заказ).

Представим, что у нас есть такая НЕБЛОКИРУЮЩАЯ функция:

```
bool Client::dataAvailable() throw (std::exception)
{
    // мгновенно возвращает true, если в ядре есть данные для Client
    // иначе return false;
}
```


Мультиплексирование: псевдо реализация



```
std::vector<std::shared_ptr<Client>> clients;
Server serv;
serv.createServerSocket(port, 25);
serv.setNonBlocked(true);

while(true) {
    usleep(100);
    while (auto client = serv.accept()) {
        client->setNonBlocked(true);
        clients.push_back(client);
    }
    std::for_each(clients.begin(), clients.end(), [](shared_ptr<Client> s){
        if (s->dataAvailable()) s->onRead();
    });
}
```

Мультиплексирование: каждый сервер делает это



```
Server serv;  
serv.createServerSocket(port, 25);  
  
while(true) {  
    res = multiplex_method(clients, timeout);  
    for(client : res)  
    {  
        client_work(client);  
    }  
};  
}
```

nginx = этот_пример ^ количество_ядер + master-process.
+ одновременная обработка 100К клиентов
+ при малом количестве затраченных ресурсов
- нельзя делать "тяжелые" синхронные операции

Мультиплексирование: select



```
#include <sys/select.h>

int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

- + достаточно эффективен
- + супер-портабелен
- макс 1024 дескриптора
- максимальный fd должен быть < 1024
- fd_set – модифицируется после каждого вызова
- нужно вычислять максимальный дескриптор
- ограниченное количество событий

Мультиплексирование: select, вспомогательные классы



```
// Базовый движок, реализует слушающий сокет
class Engine {
    int m_Listener;
    int createServerSocket(uint32_t port, uint32_t listen_queue_size);
public:
    explicit Engine(int port);
    virtual void run() =0;
    int listener() const { return m_Listener; }
};

// Просто оболочка для сокета
struct Client {
    explicit Client(int _sd) : sd(_sd) {}
    int sd;
};
```

Мультиплексирование: select



```
class SelectEngine: public Engine {
    public:
        explicit SelectEngine(int port): Engine(port) {}
        virtual void run() override;
    private:
        int prepare(fd_set *read_fds);
        void eventLoop();
        std::vector<Client> m_Clients;           // Здесь храним
клиентов
};
```

Мультиплексирование: select



```
int SelectEngine::prepare(fd_set *read_fds) {  
  
    // Инициализация структур для вызова select()  
  
    FD_ZERO(read_fds);  
    FD_SET(listener(), read_fds);  
    int fdmax = listener();  
    for (auto c: m_Clients) {  
        if (c.sd > fdmax) fdmax = c.sd;  
        FD_SET(c.sd, &read_fds);  
    }  
    return fdmax;  
}
```

Мультиплексирование: select event loop



```
while (true) {
    fd_set read_fds;
    int fd_max = prepare(&read_fds);
    int sel = select(fdmax + 1, &read_fds, NULL, NULL, /*timeout*/NULL);
    for (size_t i = 0; i < m_Clients.size(); ++i) {
        if (!FD_ISSET(m_Clients[i].sd, &read_fds)) continue;
        if (m_Clients[i].sd == listener()) {
            m_Clients.push_back(Client(cli_sd));
        }
        else {
            recv(m_Clients[i].sd, buf, sizeof(buf));
            send(m_Clients[i].sd, "hello!", sizeof("hello!"));
        }
    }
}
```

Мультиплексирование: poll



```
#include <sys/poll.h>
```

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

- + нет ограничения на макс кол-во дескрипторов
- + достаточно широко портабелен (но уже внутри linux, bsd, solaris)
- + есть список наблюдаемых событий – удобно для детектирования отключения клиентов (нет необходимости в операции чтения и сравнения с 0)
- чуть менее эффективен, чем select (8-16 байт на сокет)

Мультиплексирование: poll



```
enum class client_state_t: uint8_t { WANT_READ, WANT_WRITE };

struct Client {
    explicit Client(int _sd): sd(_sd), state(client_state_t::WANT_READ) {}
    Client(int _sd, client_state_t _state) : sd(_sd), state(_state) {}
    int sd;
    client_state_t state;
};

class PollEngine: public Engine {
    std::vector<Client> m_Clients;
    void eventLoop();
public:
    explicit PollEngine(int port): Engine(port) {}
    virtual void run() override;
};
```

Мультиплексирование: poll



```
int PollEngine::prepare(struct pollfd *fds) {  
    for (size_t i = 0; i < m_Clients.size(); ++i) {  
        fds[i].fd = m_Clients[i].sd;  
        if (m_Clients[i].state == client_state_t::WANT_READ)  
            fds[i].events = POLLIN;  
        else  
            fds[i].events = POLLOUT;  
    }  
}
```

Мультиплексирование: poll event loop



```
struct pollfd fds[32768];
while (true) {
    prepare(fds);
    poll(fds, m_Clients.size(), /* timeout in msec */ 0);
    for (size_t i = 0; i < m_Clients.size(); ++i) {
        if (fds[i].revents & POLLIN) {
            recv(m_Clients[i]);
            m_Clients[i].state = client_state_t::WANT_WRITE;
        }
        else if (fds[i].revents & POLLOUT) {
            send(fds[i].fd, "hello", sizeof("hello"));
            m_Clients[i].state = client_state_t::WANT_READ;
        }
    }
}
```

Мультиплексирование: epoll



```
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events,
               int maxevents, int timeout);
```

- + позволяет добавлять, удалять, модифицировать дескрипторы и события
- + эффективен (лучше чем poll) при > 1К клиентов
- + возвращает только сокеты, на которых есть события
- + можно хранить “клиентов” прямо “внутри” epoll
- + нет ограничения на макс кол-во дескрипторов
- не портабелен - только linux после 2.5

Мультиплексирование: epoll, event loop



```
int epfd = epoll_create(max_clients);
struct epoll_event *events = malloc(max_clients*sizeof(struct epoll_event));
int epoll_ret = epoll_wait (epfd, events, max_clients, -1);
for (int i = 0; i < epoll_ret; ++i) {
    if (events[i].data.fd == listener()) {
        struct epoll_event cli_ev;
        cli_ev.data.ptr = new Client(::accept(listener()...));
        cli_ev.events = EPOLLIN;
        epoll_ctl(epfd, EPOLL_CTL_ADD, cli_sd, &cli_ev);
    }
    else if (events[i].events & EPOLLIN) {
        Client *cs = static_cast<Client*>(events[i].data.ptr);
        cs->doRead();
    }
}
```

Коннект с таймаутом (connect != recv) с помощью select



```
int sd = socket(/*Protocol Family*/PF_INET, SOCK_STREAM, IPPROTO_TCP);
set_non_blocked(sd, true);
int connected = ::connect(sd, (struct sockaddr*)&addr, sizeof(addr));
// errno == EINPROGRESS

fd_set write_fds;
FD_ZERO(&write_fds);
FD_SET(sd, &write_fds);
struct timeval tm {.tv_sec = timeout, .tv_usec = 0};

int sel = select(sd + 1, NULL, &write_fds, /*except*/NULL, &tm);
if (sel != 1)
{
    throw std::runtime_error("connect timeout");
}
```

Пример: коннект с таймаутом



1. Коннект на обычный хост:

```
./client_timeout google.com 80           # OK!
```

*Можно использовать эти адреса для тестов NOROUTE:

192.168.0.0, 10.255.255.*, 192.168.255.255...

2. Коннект на “медленный” хост без таймаута:

```
telnet google.com 81  # long hang
```

Connection refused

3. Коннект на “медленный” хост с таймаутом:

```
./client_timeout google.com 81           # hang 5 sec
```

connect timeout

Чтение с таймаутом с помощью select (just in case)



```
std::string Client::recvTimed(int timeout) throw (std::exception) {  
    fd_set read_fds;  
    FD_ZERO(&read_fds);  
    FD_SET(m_Sd, &read_fds);  
  
    struct timeval tm;  
    tm.tv_sec = timeout;  
    tm.tv_usec = 0;  
  
    int sel = select(m_Sd+1, &read_fds, /*w*/NULL, /*except*/NULL, &tm);  
    if (sel != 1)  
        throw std::runtime_error("read timeout");  
  
    return recv();  
}
```


Мультиплексирование: библиотеки



Самые известные: libevent (<http://libevent.org/>), libev, libuv

- кроссплатформенные
- умеют select, poll, epoll, kqueue, /dev/poll и даже windows
- имеют “биндинги” к другим языкам (к примеру, Event::Lib для perl)

Простой echo-сервер, select, poll, epoll:

<https://github.com/o2qy84/misc/tree/master/technopark/libevent>

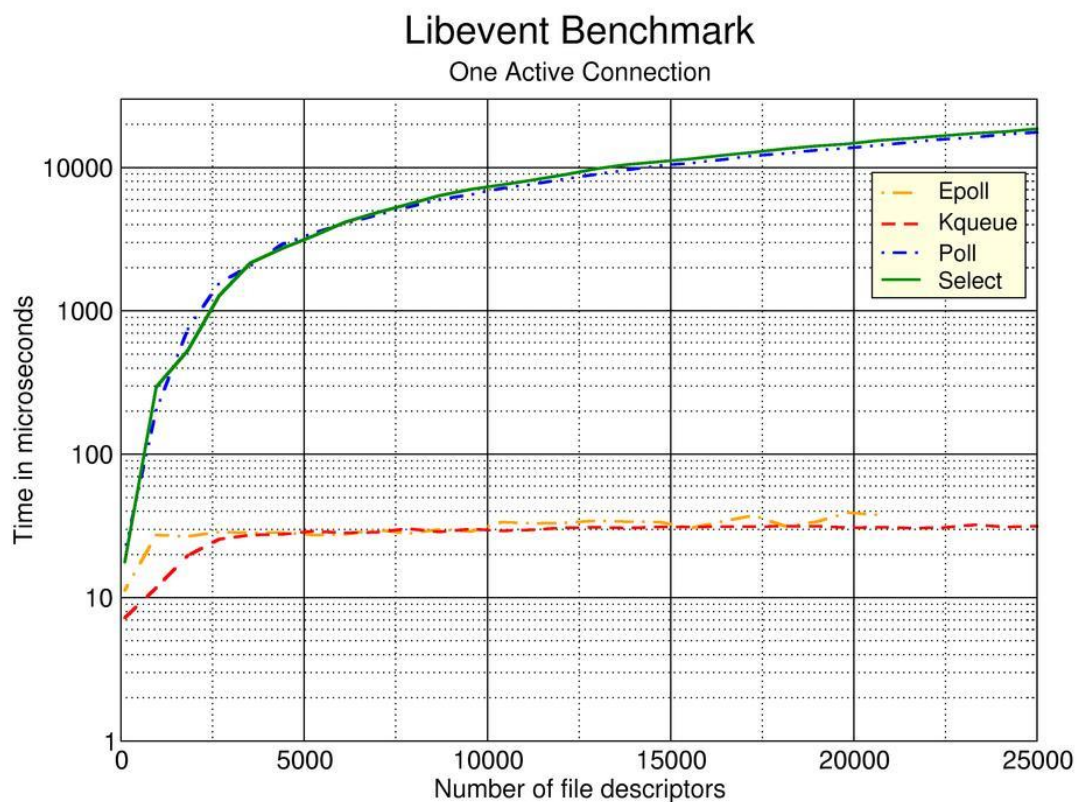
Http прокси-сервер:

<https://github.com/o2qy84/o2proxy>

Мультиплексирование.



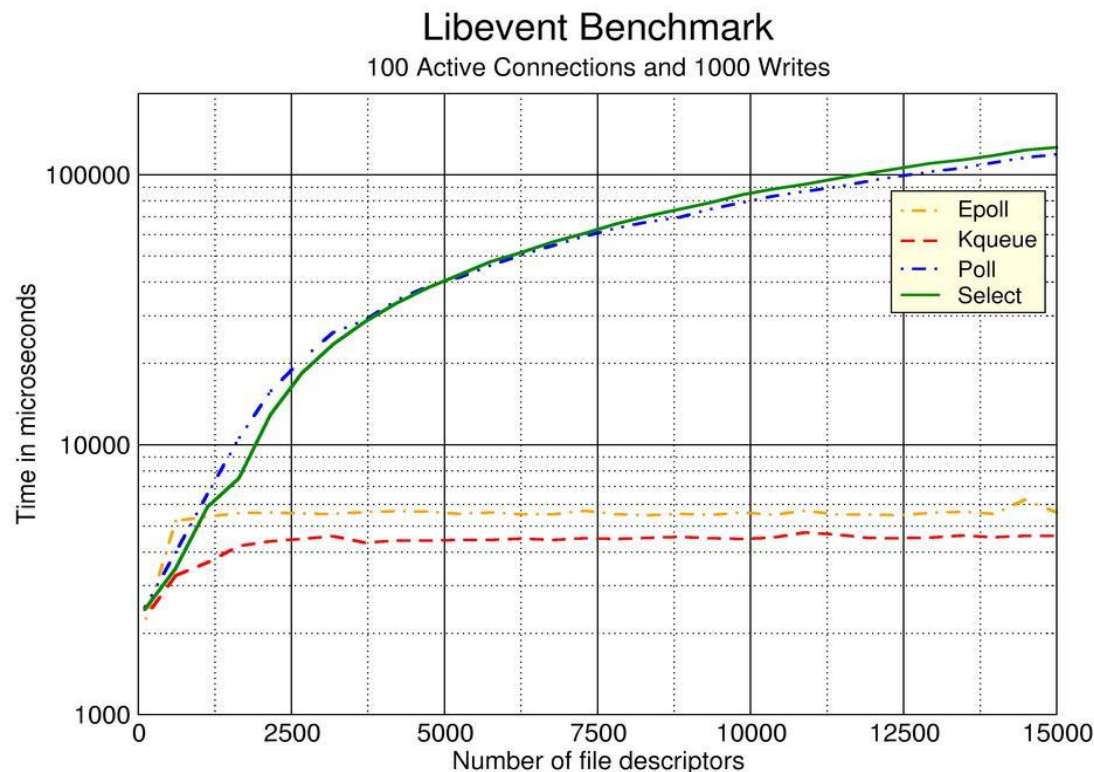
Время на обработку события в зависимости от числа соединений



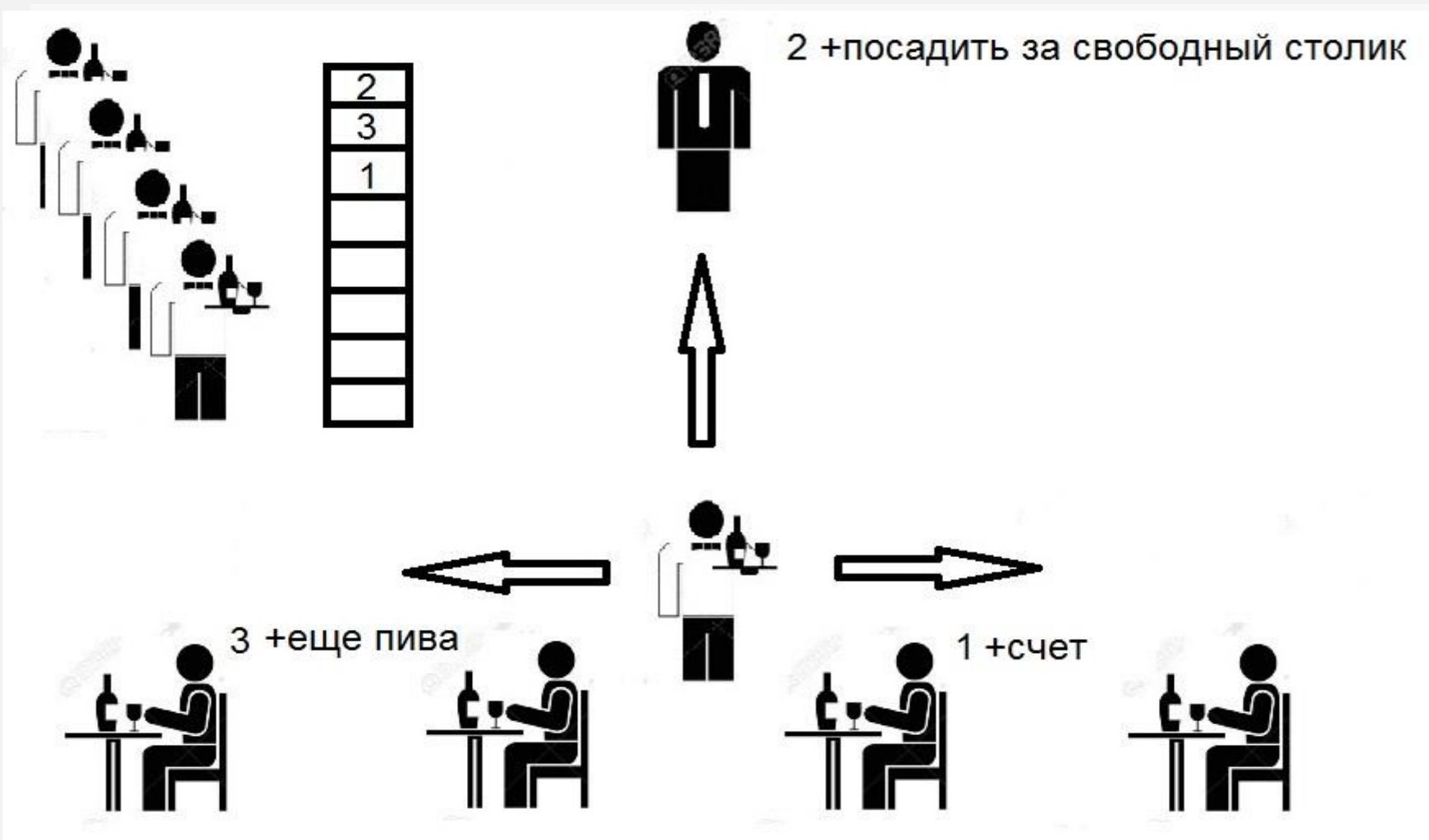
Мультиплексирование. Время на обработку события



Время на обработку 1000 операций записи/чтения в 100 активных соединений в зависимости от общего числа соединений (дескрипторов).



Асинхронная очередь



Асинхронная модель: мультиплексирование + очередь



Плюс и минус:

- мультиплексирует, но за счет очереди чуть менее эффективный
- не “складывается” из-за нескольких “тяжелых” операций

Для асинхронной модели нам понадобятся:

- очередь ← **где** сохраняем события?
- `bind + shared_ptr` ← **как** храним обработчики события?
- пулл потоков ← **чем** вызываем обработчики?

Асинхронная модель: bind + shared_ptr - объект живи!



```
#include <functional>
class Class : public std::enable_shared_from_this<Class> {
public:
    void funcB(int arg1, const std::string &arg2) {}
};

std::vector<std::function<void (int)>> Queue;

{    // с будет уничтожен при выходе за scope
    Class c;

    Queue.push_back(bind(&Class::funcB, &c, _1, _2));    // f(4,
"str")
}

{    //c2 будет жить!
    std::shared_ptr<Class> c2 = std::make_shared<Class>();
    Queue.push_back(bind(&Class::funcB, c2->shared_from_this(), _1, _2));
}
```

Асинхронная модель: объекты



```
struct Event {
    Event(Client c, std::string &data): _client(c), _data(data) {}
    Client _client;

    // WANT_READ, WANT_WRITE

    std::function<void(int)> _callback;
    std::reference_wrapper<std::string> _data; // read or write buffer
};

class EventLoop { // SINGLETON
    std::vector<Event> m_Clients; // все клиенты
    std::queue<Event> m_ClientsHaveWork; // клиенты с событиями
    // Очереди, мьютексы и т.п.
public:
    void run();
    void asyncRead(int sd, std::string &str, std::function<void(int)> cb);
};
```

Асинхронная модель: объекты



```
// Connection - это объект, представляющий соединение

class Connection: public std::enable_shared_from_this<Connection> {
public:
    Connection(int sd, EventLoop *loop) : m_Sd(sd), m_EventLoop(loop) {}
    void read();
private:
    void readHandler(int error);    // Обработчик чтения
    int m_Sd;
    EventLoop *m_EventLoop;
    std::string m_ReadBuf;        // Сюда помещаем результат чтения
};
```


Асинхронная модель: read()



```
void EventLoop::asyncRead(int sd, std::string &str,
                          std::function<void(int)> cb)
{
    Event e (Client(sd, client_state_t::WANT_READ), str);
    e._callback = cb;
    std::unique_lock<std::mutex> lock(m_WantWorkQueueMutex);
    m_Clients.emplace_back(e);
}

void Connection::read()
{
    m_EventLoop->asyncRead(m_Sd, m_ReadBuf,
                          std::bind(&Connection::readHandler, shared_from_this(), _1)
    );
}
```

Асинхронная модель: поток - producer, наполнение очереди



```
poll(fds, m_Clients.size(), /* timeout in msec */ 10);

for (size_t i = 0; i < m_Clients.size(); ++i) {
    if (fds[i].revents & POLLIN) {           // DON'T READ, JUST PUSH QUEUE!
        m_ClientsHaveWork.push(m_Clients[i]);
        remove_from_poll(m_Clients[i]);
    }
    else if (fds[i].revents & POLLOUT) { // DON'T WRITE, JUST PUSH QUEUE!
        m_ClientsHaveWork.push(m_Clients[i]);
    }
    ...
}
```

- `m_Clients[i]` - уже содержит state: READ или WRITE, в зависимости от того, был ли вызов `Connection::read()` или `Connection::write()`

Асинхронная модель: run - обработка очереди



```
void EventLoop::run() {
    while (true) {
        // критическая секция!
        Event event = m_ClientsHaveWork.front();
        m_ClientsHaveWork.pop();

        if (event._client.state == client_state_t::WANT_READ) {
            event._data.get() = recv(event._client.sd, ...); // READ
            event._callback(errno);
        }
        ...
    }

    thread_group group(N, &EventLoop::run);
    group.join();
}
```

Асинхронная модель: примерный main



```
EventLoop &ev = EventLoop::eventLoop(); // SINGLETON
std::vector<std::thread> event_loop_threads;

for (int i = 0; i < 4; ++i) {
    event_loop_threads.push_back(thread(bind(&EventLoop::run, &ev)));
    usleep(1000);
}

while (true) {
    std::shared_ptr<Connection> conn = accept_work(listener(), &ev);
    if (conn) {
        conn->read();
    }
}
```

Асинхронная модель: проблемы реализации



- Синхронизация - два потока выполняют один и тот же handler или разные handler'ы одного объекта (этого надо избегать)
- Эффективная постановка в очередь (mutex? spinlock? lockfree?)
- Эффективный забор из очереди
- Сборка мусора (зависшие, отвалившиеся клиенты)
- Программирование callback-ами, непоследовательный код
- Обеспечение “живучести” объектов (классический пример - асинхронная запись):

```
{  
    std::string s = "hello!";    // s будет “мертв”!  
    async_write(s);  
}
```

boost::asio: общие сведения



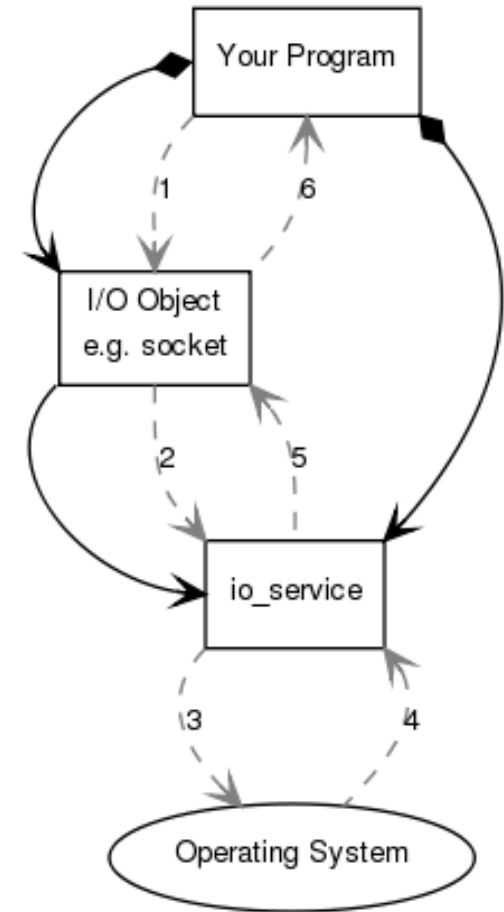
- хорошая реализация асинхронной модели;
- открытый исходный код;
- кроссплатформенность;
- максимальная производительность для каждой платформы;
- позволяет писать синхронный и асинхронный код;
- обработка ошибок;
- легко интегрируется с ssl;
- хорошо документирован.

<http://www.boost.org/> → Documentation → asio

boost::asio: анатомия синхронных операций



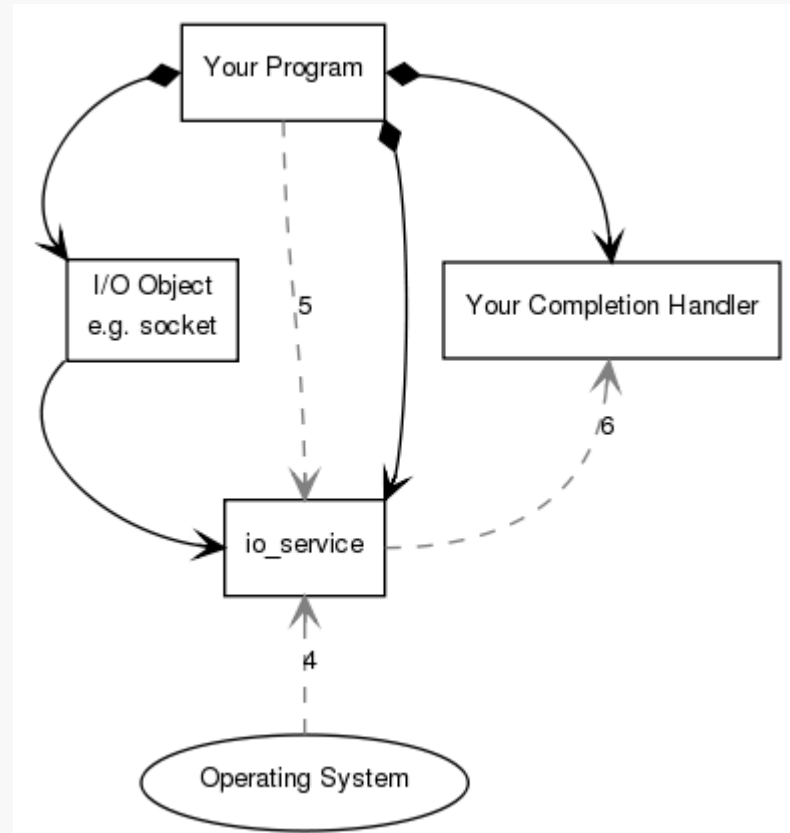
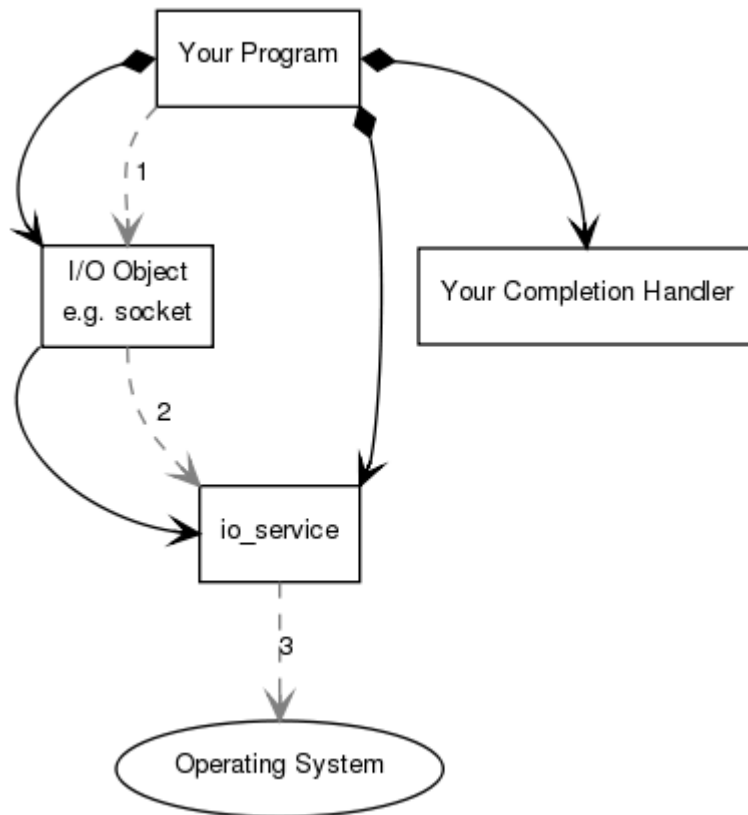
```
typedef io_context io_service;  
  
// io_service как минимум один в программе  
boost::asio::io_service io_service;  
boost::asio::ip::tcp::socket socket(io_service);  
boost::system::error_code ec;  
socket.connect(server_endpoint, ec);  
if(ec)  
    throw std::runtime_error(ec.message());
```



boost::asio: анатомия асинхронных операций



- `socket.async_connect(server_endpoint, your_completion_handler);`
- `void your_completion_handler(const boost::system::error_code& ec);`
- `io_service::run()`



boost::asio: общие сведения



boost::asio: - это namespace, содержащий все классы, функции и т.п.
boost::asio::io_service io_service; - должен быть в каждой программе

Есть аналоги для всех “нативных” функций:

connect() => boost::asio::ip::tcp::socket::connect()
read() => boost::asio::ip::tcp::socket::read_some()
и т.д.

Есть асинхронные аналоги для всех “нативных” функций:

async_connect()/async_read() => отправить в очередь + callback

Есть таймеры, ICMP, RS232 и т.д.

“Не умеет” работать с std::string, в целях эффективности использует:

- boost::asio::streambuf (async_read_until(m_socket, stream, “\r\n”))
- boost::asio::buffer (async_read_some(asio::buffer(data, length)))

boost::asio: зависимости



Пример: <https://github.com/o2gy84/misc/tree/master/tecnopark/asio>

В коде:

```
#include <boost/asio.hpp>  
#include <boost/bind.hpp>
```

При сборке:

```
g++ -I/usr/boost156 -L/usr/boost156 -o app -std=c++14 main.cpp -lboost_system -lpthread
```

boost::asio: пример синхронной работы, клиент



```
boost::asio::io_service io_service;
boost::asio::ip::tcp::socket s(io_service);
boost::asio::ip::tcp::resolver resolver(io_service);
boost::asio::ip::tcp::resolver::query query("localhost", "7789");

boost::system::error_code ec;
auto endpoint = resolver.resolve(query, ec);
if (ec) throw std::runtime_error(ec.message());

boost::asio::connect(s, endpoint, ec);
if (ec) throw std::runtime_error(ec.message());

boost::asio::write(s, boost::asio::buffer("LOGIN\r\n", 7), ec);
if (ec) throw std::runtime_error(ec.message());

char reply[max_len];
s.read_some(boost::asio::buffer(reply, max_len), ec);
```

boost::asio: пример синхронной работы, нужно больше C++!



```
ip::tcp::iostream stream;
stream.expires_from_now(boost::asio::chrono::seconds(60));
stream.connect("www.boost.org", "http");
stream << "GET /LICENSE_1_0.txt HTTP/1.0\r\n";
stream << "Host: www.boost.org\r\n";
stream << "Accept: */*\r\n";
stream << "Connection: close\r\n\r\n";
stream.flush();
std::cout << stream.rdbuf();
```

boost::asio: можно считать очередью



```
void func() { std::cerr << "func" << std::endl; }
struct Struct { void operator()() {cerr << "operator()" << std::endl; }};

boost::asio::io_service io;
io.post(func);
io.post(Struct());
io.post([]()->void { std::cerr << "lambda" << std::endl; });

std::shared_ptr<Class> c = std::make_shared<Class>();
io.post(std::bind(&Class::funcC, c->shared_from_this()));

io.run_one(); // func
io.run_one(); // lambda
io.run_one(); // Struct::operator()
io.run_one(); // Class...

io.run(); // ИЛИ ТАК: func + lambda + Struct::operator() + Class + ...
```

boost::asio: пример многопоточного асинхронного echo-сервера



```
// Класс, представляющий клиентское соединение

class Client: public std::enable_shared_from_this<Client> {
public:
    Client(boost::asio::io_service &io) : m_Sock(io) {}
    boost::asio::ip::tcp::socket& sock() { return m_Sock; }
    void read();
    void handleRead(const boost::system::error_code& e, size_t bytes);

private:
    boost::asio::ip::tcp::socket m_Sock;
    char m_Buf[1024];
};
```

boost::asio: пример многопоточного асинхронного echo-сервера



```
void Client::read() {
    m_Sock.async_read_some(boost::asio::buffer(m_Buf),
                           bind(&Client::handleRead, shared_from_this(),
                               boost::asio::placeholders::error,
                               ::placeholders::bytes_transferred));
}

void Client::handleRead(const boost::system::error_code& e, size_t bytes) {
    if (e) return;
    m_Sock.async_write_some(boost::asio::buffer(m_Buf),
                            [self = shared_from_this()](const error_code& e, size_t bytes) {
                                // После того, как запишем ответ, можно снова читать
                                self->read();
                            });
}

};
```

boost::asio: пример многопоточного асинхронного echo-сервера



```
class Server {
    boost::asio::io_service m_Service;
    boost::asio::ip::tcp::acceptor m_Acceptor;
    void onAccept(std::shared_ptr<Client> c, const error_code& e) {
        if (e) return;
        c->read();
        startAccept();
    }
    void startAccept() {
        std::shared_ptr<Client> c(new Client(m_Service));
        m_Acceptor.async_accept(c->sock(),
            bind(&Server::onAccept, this, c, asio::placeholders::error));
    }
public:
    Server() : m_Acceptor(m_Service) {}
    void startServer();
};
```


boost::asio: пример многопоточного асинхронного echo-сервера



```
void Server::startServer() {
    boost::asio::ip::tcp::endpoint endpoint(asio::ip::tcp::v4(), 5001);
    m_Acceptor.open(endpoint.protocol());
    m_Acceptor.bind(endpoint);
    m_Acceptor.listen(1024);
    startAccept();

    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i)
        threads.push_back(thread(bind(&io_service::run, &m_Service)));
    for (auto &thread: threads)
        thread.join();
}

int main(int argc, char *argv[]) {
    Server().startServer();
    return 0;
}
```

Итоги: какая архитектура лучше?



1. fork/thread per connection - “домашние” проекты, прототипы
2. prefork - если не критично количество ресурсов, или заранее известно, что будет небольшое количество клиентов
3. multiplexing - мало бизнес логики (прокси, балансеры и т.п.);
4. prefork + multiplexing - годится для “большого” прода, но может быть дорого (один httpd может “кушать” и гигабайт), а делать prefork на тредах - уменьшается надежность, уж лучше тогда asio;
5. asio - годится для “большого” прода, требует меньше памяти, но сильно сложнее в реализации.

Примеры кода



<https://github.com/o2gy84/misc/tree/master/texnopark>

- `common` - общий класс для работы с сетью
- `simple-server` - работа только с одним клиентом
- `client_connect_timeout` - клиент, умеющий таймауты на соединение
- `client_read_timeout` - клиент, умеющий таймауты на чтение
- `http_client` - пример выполнения http-запроса
- `multiplexing` - "примитивное" мультиплексирование
- `prefork` - форкающийся сервер

- `libevent` - echo-сервер на libevent
- `asio` - echo-сервер на boost::asio

- `libtpevent` - сервер, в котором можно выбирать несколько вариантов `event_loop`'ов, в том числе реализован асинхронный движок на `poll`