Сетевое взаимодействие 2

АКОС, МФТИ

21 ноября, 2024

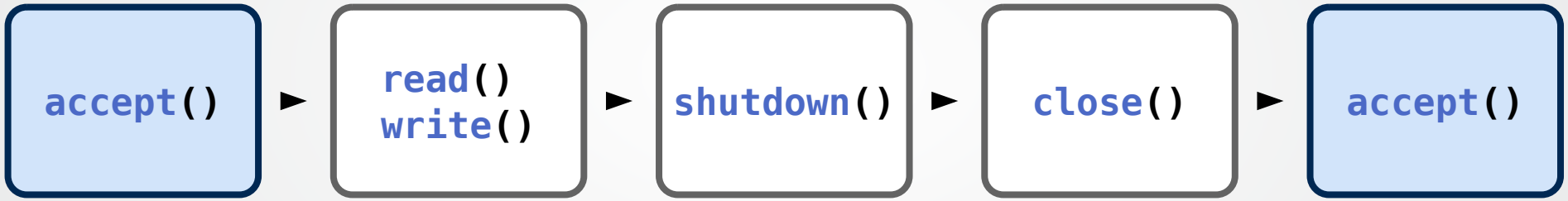
Как выглядел наш TCP-сервер

- Создаём сокет, связываем его с адресом через `bind(...)`
- Переводим сокет в режим сервера через `listen(...)`
- В цикле принимаем подключения через `accept(...)`, обрабатываем их и закрываем через `shutdown(...)` и `close(...)`.
- **В чем проблема такого сервера?**

```
1  sock = socket(AF_INET, SOCK_STREAM, 0);
2
3  // Какой адрес слушать
4  bind(sock,(...*)&addr, sizeof(addr));
5
6  // Перейти в режим сервера
7  listen(sock, CONNECTION_QUEUE_LEN);
8
9  while(server_is_running) {
10   // Принять подключение
11   int conn = accept(sock);
12   // conn - сокет для общения с клиентом
13
14   shutdown(conn, O_RDWR);
15   close(conn);
16 }
17 close(sock);
```

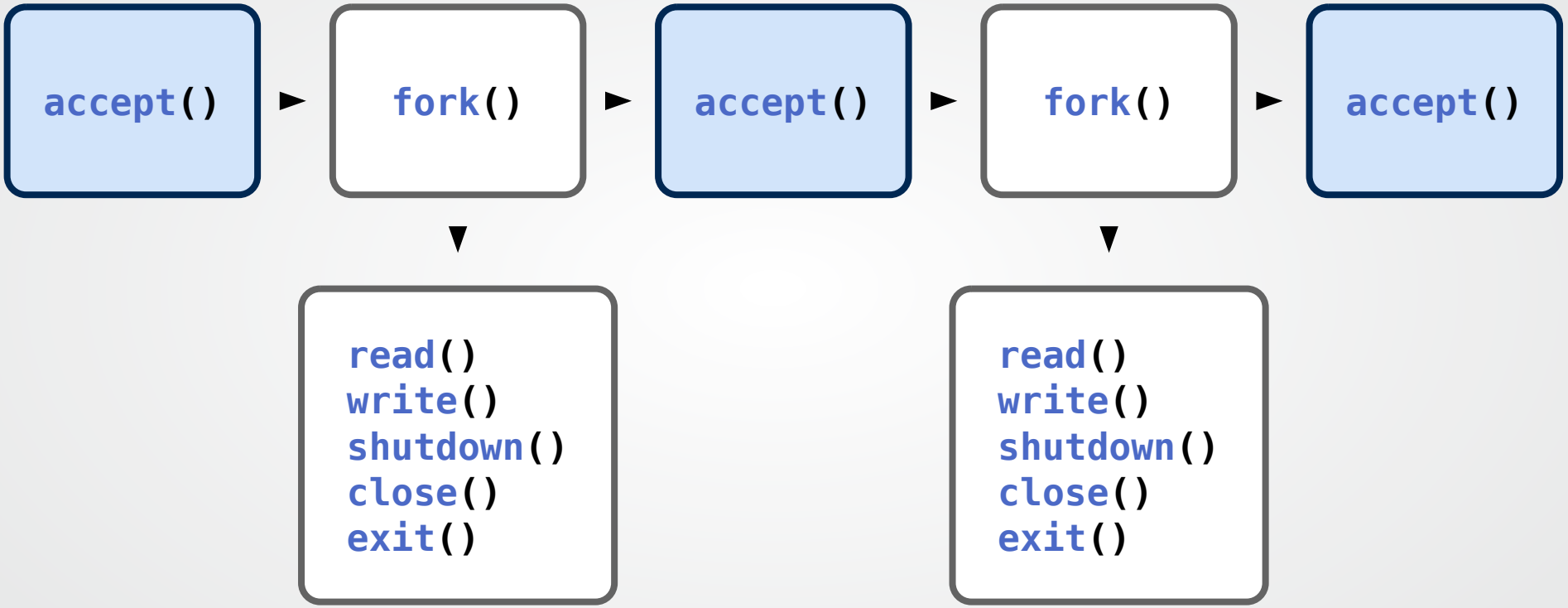
Недостаток линейной схемы

- `accept()` вызывается в том же цикле, в котором происходит общение с клиентом.



- Такой сервер не сможет поддерживать больше одного активного соединения.

Обработка подключений с `fork()`



Простое решение – по процессу на клиента

```
1  // ...
2
3  while(server_is_running) {
4    // Принять подключение
5    int conn = accept(sock);
6
7    if(fork() == 0) {
8      // Работа с клиентом
9      // в дочернем процессе
10
11     shutdown(conn, O_RDWR);
12     close(conn);
13     exit(0);
14   }
15 }
```

- `fork()` -аемся каждое подключение, и работаем с клиентом в дочернем процессе. Следующий `accept()` не будет ждать

Плюсы:

- Теперь мы можем обрабатывать несколько клиентов одновременно;
- Очень простое решение.

Минусы:

- Превращается в форк-бомбу при большом количестве клиентов;

`fork()` → `pthread_create()`

Решение посложнее - создавать потоки вместо дочерних процессов.

\+ Потоки создаются быстрее, чем процессы;

\+ Потоки занимают меньше памяти;

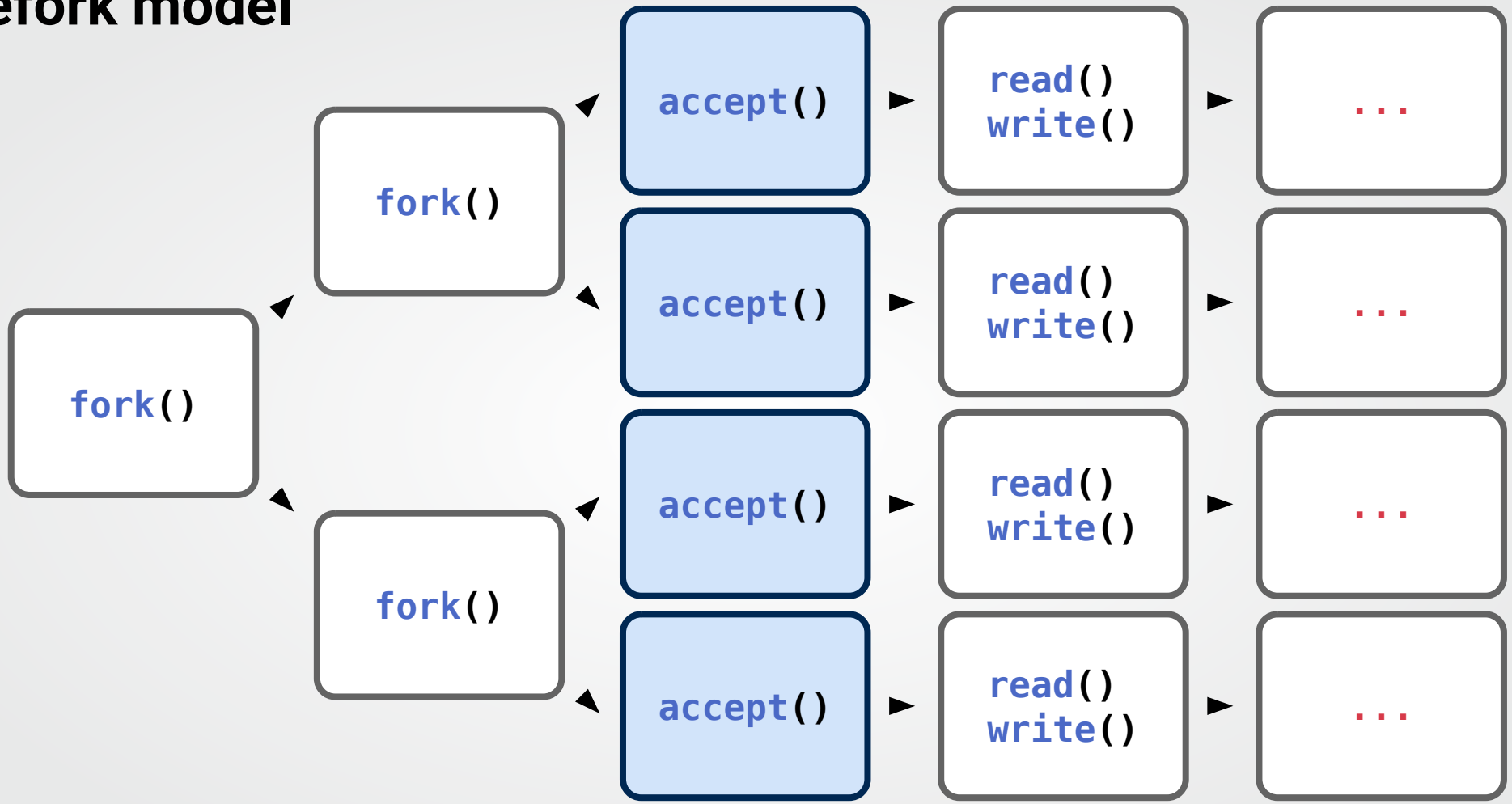
Но придётся мириться со следующим:

\- Здесь уже не обойтись одним `if` -ом.

\- С потоками приходят все проблемы многопоточности;

\- Потоки не дают такой изоляции, как процессы. Уязвимость в логике работы с одним подключением может привести к утечке данных между клиентами.

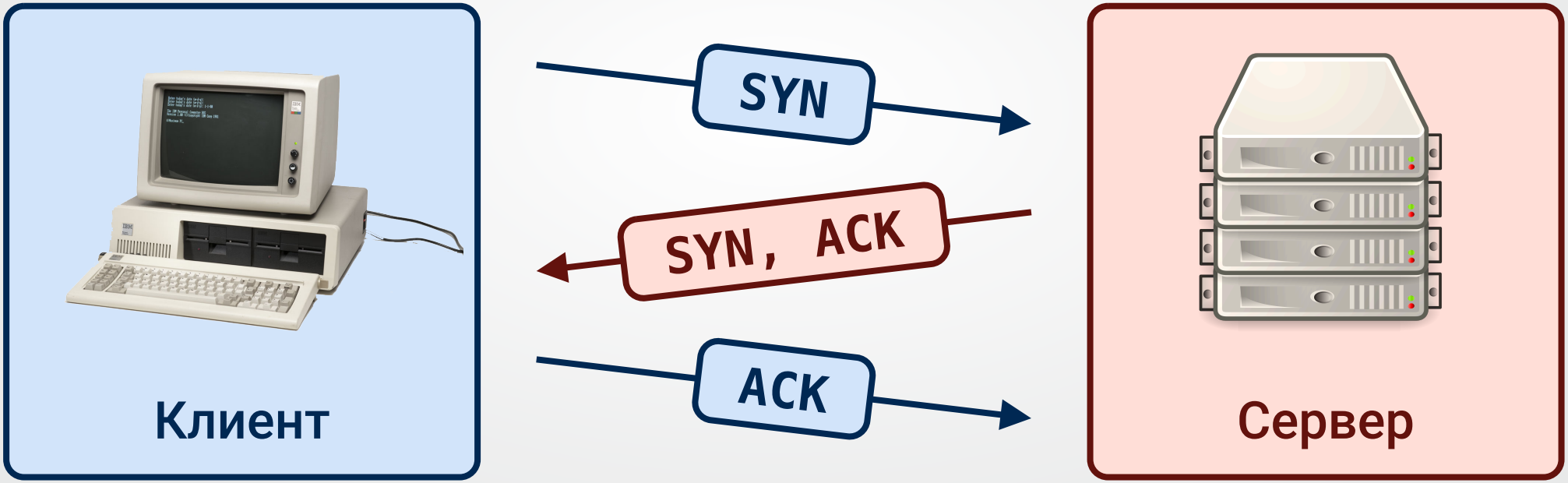
Prefork model



Идея - создать все процессы заранее. Каждый процесс работает по линейной схеме.

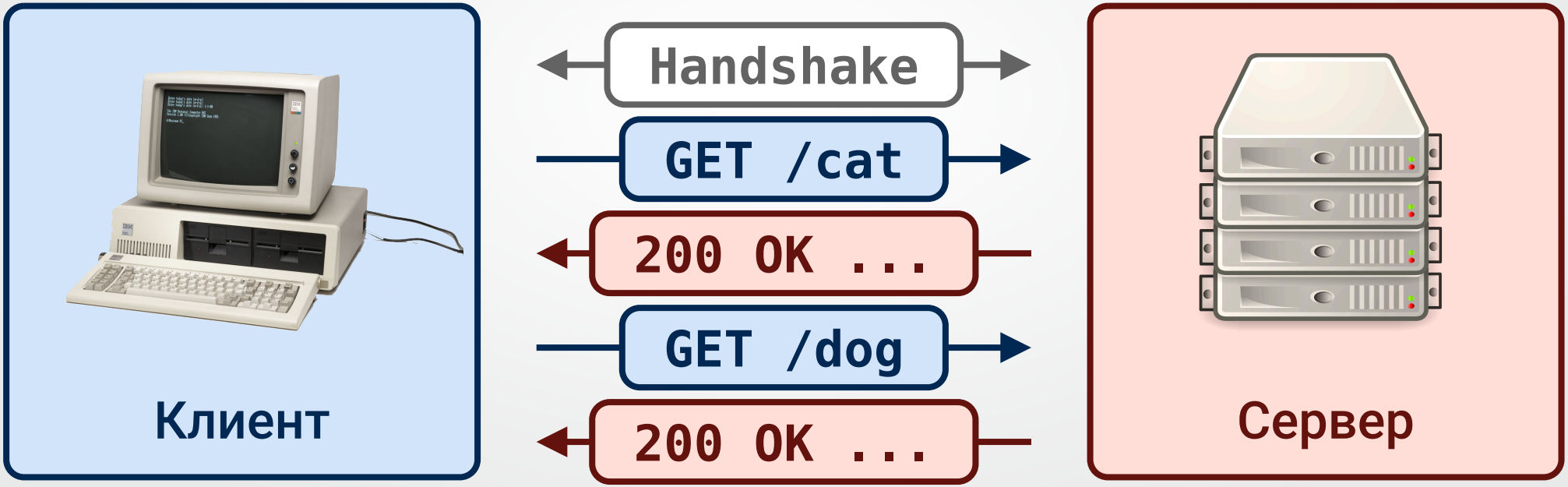
TCP Handshake

- Создание TCP-соединения происходит в три этапа.
- Такая схема называется “трёхсторонним рукопожатием”.
- Этот процесс занимает время, поэтому придумали keepalive-соединения.

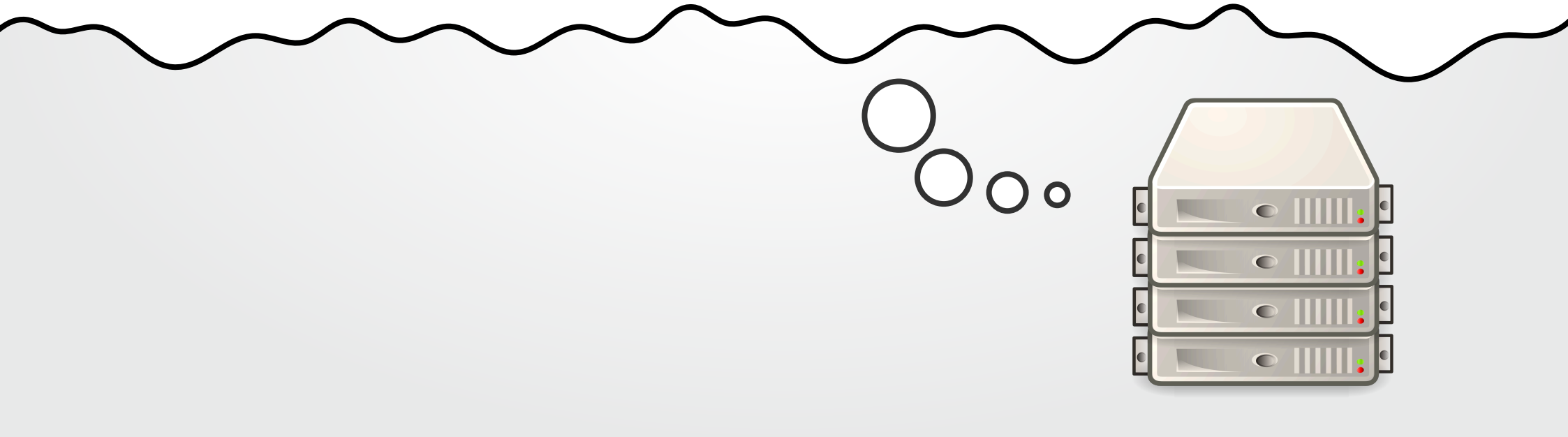


HTTP: Keepalive

- В старом HTTP каждый запрос - новое соединение. Сайты стали состоять из большого количества мелких файлов, и это стало проблемой.
- HTTP/1.1 даёт отправлять несколько запросов через одно соединение. Это называется keepalive-соединением. (HTTP-заголовок – **`Connection: keepalive`**)



- Keepalive-соединение может **долго не делать ничего полезного**.
- Выделять под каждое такое соединение поток или процесс – дорого.
- В линейной или prefork-модели это жестко ограничивает количество клиентов.
- **Как быть?**



- Keepalive-соединение может **долго не делать ничего полезного**.
- Выделять под каждое такое соединение поток или процесс – дорого.
- В линейной или prefork-модели это жестко ограничивает количество клиентов.
- **Как быть?**

Решение – мультиплексоры!

Мультиплексор

- Интерфейс, позволяющий получать события от нескольких дескрипторов.
- Помогает, если у вас много сокетов, и запрос может прилететь от любого из них.

Мультиплексоры в Linux

`select()`

- Самый старый мультиплексор в Linux;
- Поддерживает до 1024 дескрипторов;
- Требует перезаписывать маску наблюдаемых дескрипторов перед каждым вызовом.
- Тормозит, если дескрипторов много;

`poll(...)`

- Поддерживает больше дескрипторов;
- Позволяет отследить отключение клиента;
- Даёт переиспользовать маску дескрипторов;
- Всё ещё тормозит, если дескрипторов много.

`epoll`

- То же самое, что и `poll()`, но позволяет получать список готовых к чтению/записи дескрипторов за $O(1)$;
- Работает хорошо, но не кросс-платформенный.

```
select(int n, fd_set *r, fd_set *w, fd_set *e, timeval *t)
```

Блокируется до события на дескрипторах или до таймаута.

`int n`

- Максимальное численное значение дескриптора + 1.
- Ядро будет проверять дескрипторы от 0 до $n - 1$.
- Можно указать 1024, но вы замучаете ядро.

`fd_set *readfds, *writefds, *exceptfds` =

- Маски отслеживаемых дескрипторов: на чтение, на запись, и на ошибку.

`timeval *timeout`

- Максимальное время ожидания события.

`fd_set` - это структура, хранящая битовую маску. В ней есть место для 1024 битов, и это ограничение нельзя увеличить. Поэтому `select()` не поддерживает дескрипторы с номером больше 1023.

Как использовать `select()`

```
1  fd_set readfds;          // readfds - маска дескрипторов для чтения
2  FD_ZERO(&readfds);
3  FD_SET(sock1, &readfds);
4  FD_SET(sock2, &readfds);
5  // ... - Добавление остальных дескрипторов
6
7  int result = select(MAX(sock1, sock2, ...) + 1, &readfds, NULL, NULL, NULL);
8  if(result == -1) {
9    // Ошибка
10 } else if(result == 0) {
11   // Таймаут
12 } else {
13   if(FD_ISSET(sock1, &readfds)) handle_data(sock1);
14   if(FD_ISSET(sock2, &readfds)) handle_data(sock2);
15   // ... - Проверка остальных дескрипторов
16 }
```

(Но лучше – никак...)

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout)
```

Блокируется до события на дескрипторах или до таймаута.

`struct pollfd *fds`

- Указатель на массив структур `pollfd`, описывающих дескрипторы и события, которые нужно отслеживать.

`nfds_t nfds`

- Количество элементов в массиве `fds`.

`int timeout`

- Максимальное время ожидания события в миллисекундах.

`struct pollfd`

Структура, описывающая дескриптор и события, которые нужно отслеживать.

`int fd`

- Дескриптор.

`short events` =

|=	**POLLIN**	Событие “есть данные для чтения”;
|=	**POLLOUT**	Событие “можно записывать данные”;
|=	**POLLERR**	Событие “ошибка на дескрипторе”;
|=	**POLLHUP**	Событие “положили трубку”.

- Битовая маска событий, которые нужно отслеживать.

`short revents`

- В этом поле ядро указывает события, которые на самом деле произошли.

Как использовать `poll()`

```
1  struct pollfd fds[2] = {};   // Массив структур pollfd
2  fds[0].fd = sock1;
3  fds[0].events = POLLIN;
4  fds[1].fd = sock2;
5  fds[1].events = POLLIN;
6  // ... - Добавление остальных дескрипторов
7
8  int result = poll(fds, 2, -1);
9  if(result == -1) {
10   // Ошибка
11 } else if(result == 0) {
12   // Таймаут
13 } else {
14   if(fds[0].revents & POLLIN) handle_data(sock1);
15   if(fds[1].revents & POLLIN) handle_data(sock2);
16   // ... - Проверка остальных дескрипторов
17 }
```

Осторожно!

Сейчас будет куча скучных слайдов про `epoll`

`int epoll_create(int size)`

Создаёт новый epoll-дескриптор.

`int size`

- Устаревший параметр, который игнорируется в современных версиях ядра.
- Рекомендуется использовать `epoll_create1()` вместо этого.

`int epoll_create1(int flags)`

Создаёт новый epoll-дескриптор с флагами

`int flags`

- Флаги для создания epoll-дескриптора.
- Например, `EPOLL_CLOEXEC` для автоматического закрытия при `exec()`.

```
int epoll_ctl(int epfd, int op, int fd, epoll_event *event)
```

Управляет событиями, отслеживаемыми epoll-дескриптором.

`int epfd`

- epoll-дескриптор, созданный с помощью `epoll_create()` или `epoll_create1()`.

`int op`

- Операция, которую нужно выполнить.
- Может быть `EPOLL_CTL_ADD`, `EPOLL_CTL_MOD`, или `EPOLL_CTL_DEL`.

`int fd`

- Дескриптор файла, для которого нужно управлять событиями.

`struct epoll_event *event`

- Указатель на структуру, описывающую события, которые нужно отслеживать.

`struct epoll_event`

Структура, описывающая события, которые нужно отслеживать.

`uint32_t events`

- Битовая маска событий, которые нужно отслеживать.

|=	**EPOLLIN**	Событие “есть данные для чтения”;
|=	**EPOLLOUT**	Событие “можно записывать данные”;
|=	**EPOLLERR**	Событие “ошибка на дескрипторе”;
|=	**EPOLLHUP**	Событие “положили трубку”;
|=	**EPOLLET**	Edge-Triggered режим. (Сработать при изменении состояния);

`epoll_data_t data`

- Пользовательские данные, связанные с дескриптором.

```
int epoll_wait(int epfd, epoll_event *events, int m, int t)
```

Ожидает события на epoll-дескрипторе.

`int epfd`

- epoll-дескриптор, созданный с помощью `epoll_create()` или `epoll_create1()`.

`struct epoll_event *events`

- Указатель на массив, в который будут записаны произошедшие события.

`int maxevents`

- Максимальное количество событий, которые могут быть записаны в массив.

`int timeout`

- Таймаут ожидания события в миллисекундах. Если -1, то без таймаута.

Как использовать `epoll`

```
1  int epfd = epoll_create1(0);
2  if (epfd == -1) { /* Ошибка */ }
3
4  struct epoll_event event { .events = EPOLLIN };
5  event.data.fd = sock1;
6  if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock1, &event) == -1) { /* Ошибка */ }
7
8  event.data.fd = sock2;
9  if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock2, &event) == -1) { /* Ошибка */ }
10 // ... - Добавление остальных дескрипторов
11
12 struct epoll_event events[MAX_EVENTS];
13 int result = epoll_wait(epfd, events, MAX_EVENTS, -1);
14
15 if (result == -1) { /* Ошибка */ }
16 else for (int i = 0; i < result; i++)
17     handle_data(events[i].data.fd);
```

Ждём подключений и данные одновременно

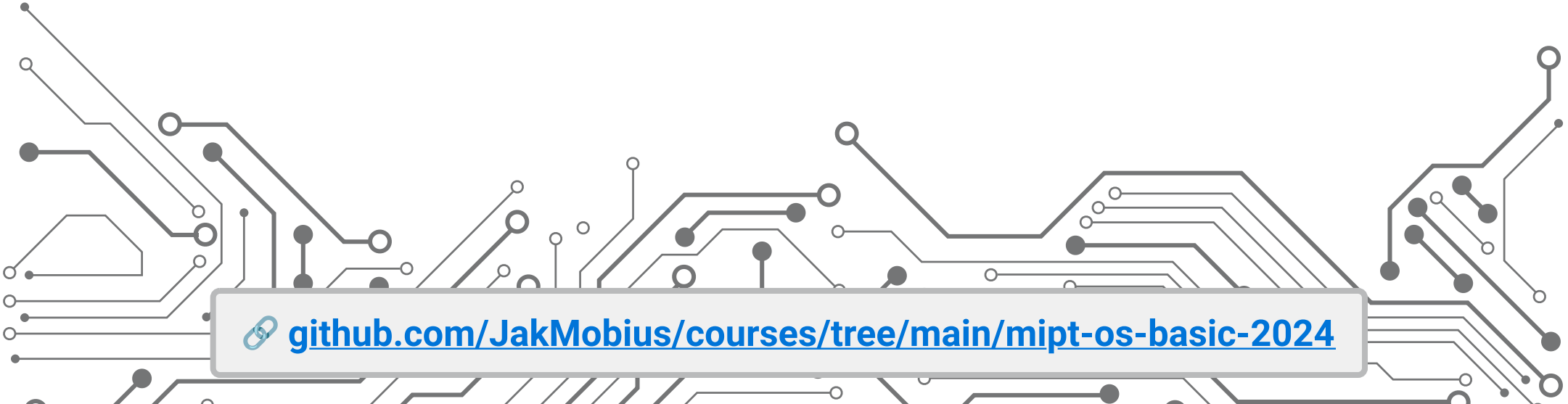
- Если к вашему серверу кто-то пытается подключиться, мультиплексор сообщит об этом так, как будто на сокете есть данные для чтения.
- Так можно одновременно ждать либо новые подключения, либо данные на существующих.

```
1  // В случае с select():
2  select(MAX_FD, &readfds, NULL, NULL, NULL);
3
4  if(FD_ISSET(server_socket, &readfds)) {
5    int new_connection = accept(server_socket);
6    // ...
7  }
```

Как написать TCP-сервер с мультиплексингом

- Создать сокет, привязать его к порту. (`socket()` , `bind()`)
- Перевести сокет в неблокирующий режим. (`fcntl()` с флагом **`O_NONBLOCK`**)
- Создать мультиплексор и добавить в него дескриптор сокета.
- Вызвать `listen()` на сокете, и начать слушать подключения.
- `while(true):`
  - Ожидать события на мультиплексоре.
  - Если событие на дескрипторе сокета, то доступно новое подключение. Нужно принять его (`accept()`) и добавить в мультиплексор.
  - Если событие на дескрипторе подключения, то на нём появились данные, либо клиент отключился.
  - Если клиент отключился, то нужно закрыть соединение и удалить дескриптор из мультиплексора.

Спасибо за внимание!



🔗 github.com/JakMobius/courses/tree/main/mipt-os-basic-2024