

# Мультиплексирование I/O в Linux

---

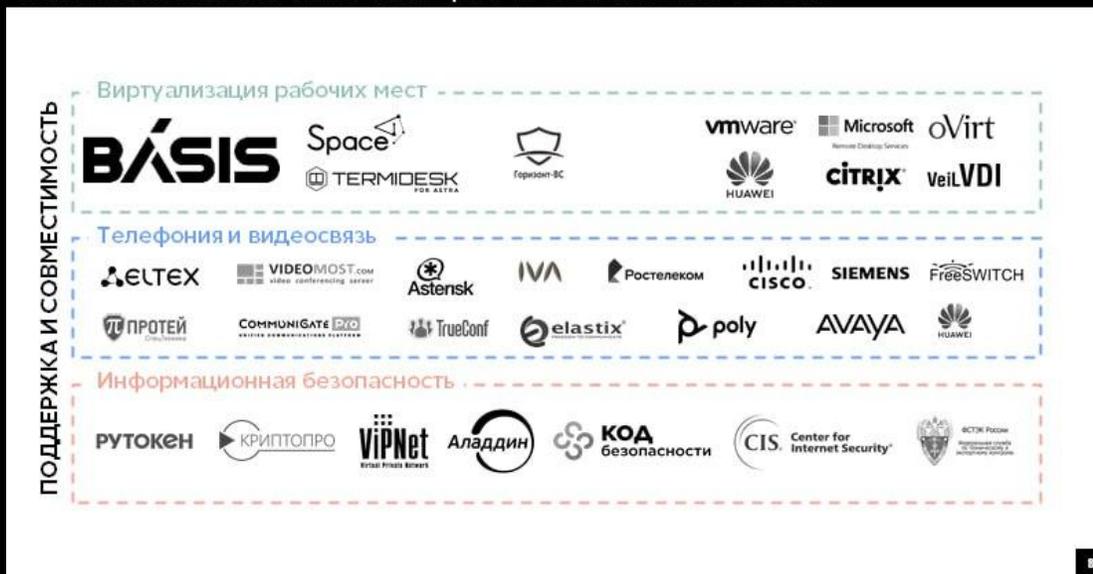
## Обо мне

- ❑ Старший разработчик и техлид направления системной разработки.
- ❑ Занимаемся автоматизацией рабочих мест и безопасным доступом к корпоративным и государственным информационным системам:
  - ❑ Виртуальным рабочим местам,
  - ❑ Видеоконференцсвязи,
  - ❑ Телефонии
  - ❑ прочие приложения.

# GETMOVIT

# Обо мне

## МЕЖСИСТЕМНАЯ ИНТЕГРАЦИЯ И СОВМЕСТИМОСТЬ

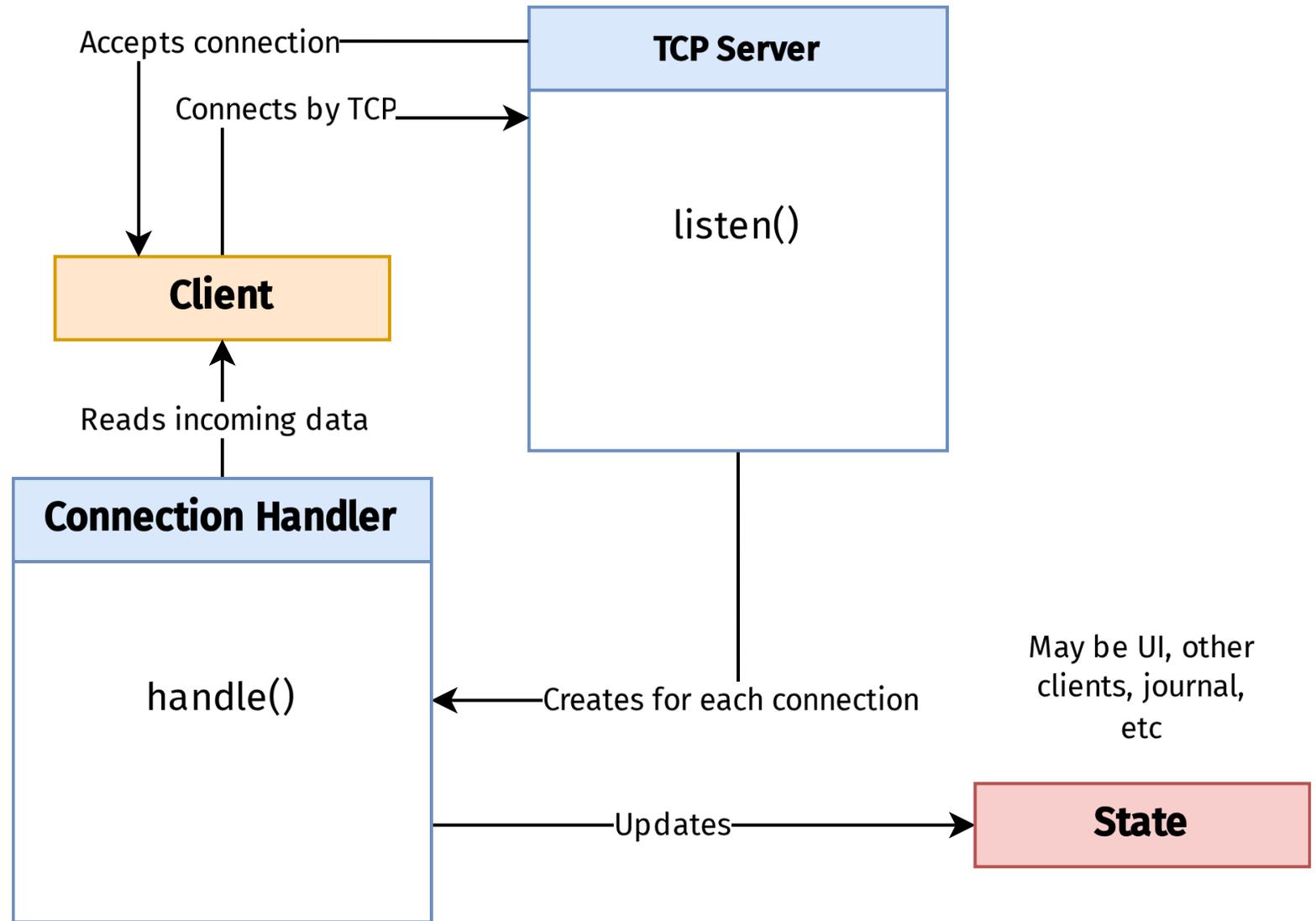


- ❑ Разрабатываем и производим универсальную гибридную док-станцию для рабочего места GM-BOX G1.
- ❑ Разрабатываем GM Smart System – защищенную цифровую платформу.
- ❑ Разрабатываем средства агрегирования информационных ресурсов в разнородных инфраструктурах.
- ❑ >50 клиентов и партнеров.

# План

- ❑ Сравним подходы работы с I/O.
- ❑ Посмотрим, что стоит за мультиплексированием внутри ядра Linux.
- ❑ Посмотрим, что нам предлагает Linux для работы с мультиплексированием:
  - `select` и `poll`;
  - `epoll`;
  - `io_uring`.

# TCP-сервер



# Как можем его реализовать?

1. **Умеем только пользоваться сокетами:** неблокирующий I/O и busy waiting.

# Busy waiting

- ❑ Используем неблокирующий I/O для наблюдения за состоянием на файловом дескрипторе.

```
1 int main() {
2     int server_fd, incoming_fd;
3     int client_fds[128];
4     char buffer[1024];
5     int bytes_read;
6     int n = 0;
7
8     /* initialize server_fd */
9
10    fcntl(server_fd, F_SETFL, O_NONBLOCK);
11    bind(server_fd, /* ... */);
12    listen(server_fd, 1);
13
14    while (1) {
15        incoming_fd = accept(server_fd, /* ... */);
16        if (incoming_fd > 0) {
17            fcntl(incoming_fd, F_SETFL, O_NONBLOCK);
18            client_fds[n++] = incoming_fd;
19        }
20
21        for (int i = 0; i < n; ++i) {
22            bytes_read = read(client_fds[i], buffer, 1024);
23            if (bytes_read ≤ 0)
24                continue;
25
26            handle(buffer, bytes_read);
27        }
28
29    }
30
31    return 0;
32 }
33
```

# Busy waiting

- ❑ Используем неблокирующий I/O для наблюдения за состоянием на файловом дескрипторе.
- ❑ Вхолостую тратим такты процессора.
- ❑ Может быть эффективно в низко-уровневом коде.

```
1 int main() {
2     int server_fd, incoming_fd;
3     int client_fds[128];
4     char buffer[1024];
5     int bytes_read;
6     int n = 0;
7
8     /* initialize server_fd */
9
10    fcntl(server_fd, F_SETFL, O_NONBLOCK);
11    bind(server_fd, /* ... */);
12    listen(server_fd, 1);
13
14    while (1) {
15        incoming_fd = accept(server_fd, /* ... */);
16        if (incoming_fd > 0) {
17            fcntl(incoming_fd, F_SETFL, O_NONBLOCK);
18            client_fds[n++] = incoming_fd;
19        }
20
21        for (int i = 0; i < n; ++i) {
22            bytes_read = read(client_fds[i], buffer, 1024);
23            if (bytes_read ≤ 0)
24                continue;
25
26            handle(buffer, bytes_read);
27        }
28
29    }
30
31    return 0;
32 }
33
```

# Busy waiting

- ❑ Используем неблокирующий I/O для наблюдения за состоянием на файловом дескрипторе.
- ❑ Вхолостую тратим такты процессора.
- ❑ Может быть эффективно в низко-уровневом коде.
- ❑ Можем снизить нагрузку на CPU.

```
1 int main() {
2     int server_fd, incoming_fd;
3     int client_fds[128];
4     char buffer[1024];
5     int bytes_read;
6     int n = 0;
7
8     /* initialize server_fd */
9
10    fcntl(server_fd, F_SETFL, O_NONBLOCK);
11    bind(server_fd, /* ... */);
12    listen(server_fd, 1);
13
14    while (1) {
15        incoming_fd = accept(server_fd, /* ... */);
16        if (incoming_fd > 0) {
17            fcntl(incoming_fd, F_SETFL, O_NONBLOCK);
18            client_fds[n++] = incoming_fd;
19        }
20
21        for (int i = 0; i < n; ++i) {
22            bytes_read = read(client_fds[i], buffer, 1024);
23            if (bytes_read ≤ 0)
24                continue;
25
26            handle(buffer, bytes_read);
27        }
28        usleep(100000 /* 100 ms */);
29    }
30
31    return 0;
32 }
33
```

# Как можем его реализовать?

1. **Умеем только пользоваться сокетами:** неблокирующий I/O и busy waiting.
2. **Thread-per-connection:** используем потоки и блокирующий I/O.

# Threads?

- ❑ Используем блокирующий I/O – убираем источник нагрузки на CPU.

```
1 void* communicate(void* data) {
2     int fd = *(int*) data;
3     free(data);
4
5     char buffer[1024];
6     int bytes_read = read(fd, buffer, 1024);
7     handle(buffer, bytes_read);
8
9     close(fd);
10    return NULL;
11 }
12
13 int main() {
14     int server_fd;
15     pthread_t t;
16
17     /* initialize server_fd */
18
19     bind(server_fd, /* ... */);
20     listen(server_fd, 1);
21
22     while (1) {
23         int* client_fd = malloc(sizeof(int));
24         *client_fd = accept(server_fd, /* ... */);
25
26         pthread_create(&t, NULL, communicate, client_fd);
27         pthread_detach(t);
28     }
29
30     return 0;
31 }
32
```

# Threads?

- ❑ Используем блокирующий I/O – убираем источник нагрузки на CPU.
- ❑ Создаём отдельный поток для каждого нового соединения.

```
1 void* communicate(void* data) {
2     int fd = *(int*) data;
3     free(data);
4
5     char buffer[1024];
6     int bytes_read = read(fd, buffer, 1024);
7     handle(buffer, bytes_read);
8
9     close(fd);
10    return NULL;
11 }
12
13 int main() {
14     int server_fd;
15     pthread_t t;
16
17     /* initialize server_fd */
18
19     bind(server_fd, /* ... */);
20     listen(server_fd, 1);
21
22     while (1) {
23         int* client_fd = malloc(sizeof(int));
24         *client_fd = accept(server_fd, /* ... */);
25
26         pthread_create(&t, NULL, communicate, client_fd);
27         pthread_detach(t);
28     }
29
30     return 0;
31 }
32
```

# Threads?

- Используем блокирующий I/O – убираем источник нагрузки на CPU.
- Создаём отдельный поток для каждого нового соединения.
- Может потребоваться синхронизация потоков.
- Больше потоков – больше времени мы тратим на переключение контекста.
- Потоки потребляют оперативную память.

```
1 void* communicate(void* data) {
2     int fd = *(int*) data;
3     free(data);
4
5     char buffer[1024];
6     int bytes_read = read(fd, buffer, 1024);
7     handle(buffer, bytes_read);
8
9     close(fd);
10    return NULL;
11 }
12
13 int main() {
14     int server_fd;
15     pthread_t t;
16
17     /* initialize server_fd */
18
19     bind(server_fd, /* ... */);
20     listen(server_fd, 1);
21
22     while (1) {
23         int* client_fd = malloc(sizeof(int));
24         *client_fd = accept(server_fd, /* ... */);
25
26         pthread_create(&t, NULL, communicate, client_fd);
27         pthread_detach(t);
28     }
29
30     return 0;
31 }
32
```

# Threads?

- Используем блокирующий I/O – убираем источник нагрузки на CPU.
- Создаём отдельный поток для каждого нового соединения.
- Может потребоваться синхронизация потоков.
- Больше потоков – больше времени мы тратим на переключение контекста.
- Потоки потребляют оперативную память.
- Альтернативы?

```
1 void* communicate(void* data) {
2     int fd = *(int*) data;
3     free(data);
4
5     char buffer[1024];
6     int bytes_read = read(fd, buffer, 1024);
7     handle(buffer, bytes_read);
8
9     close(fd);
10    return NULL;
11 }
12
13 int main() {
14     int server_fd;
15     pthread_t t;
16
17     /* initialize server_fd */
18
19     bind(server_fd, /* ... */);
20     listen(server_fd, 1);
21
22     while (1) {
23         int* client_fd = malloc(sizeof(int));
24         *client_fd = accept(server_fd, /* ... */);
25
26         pthread_create(&t, NULL, communicate, client_fd);
27         pthread_detach(t);
28     }
29
30     return 0;
31 }
32
```

# Как можем его реализовать?

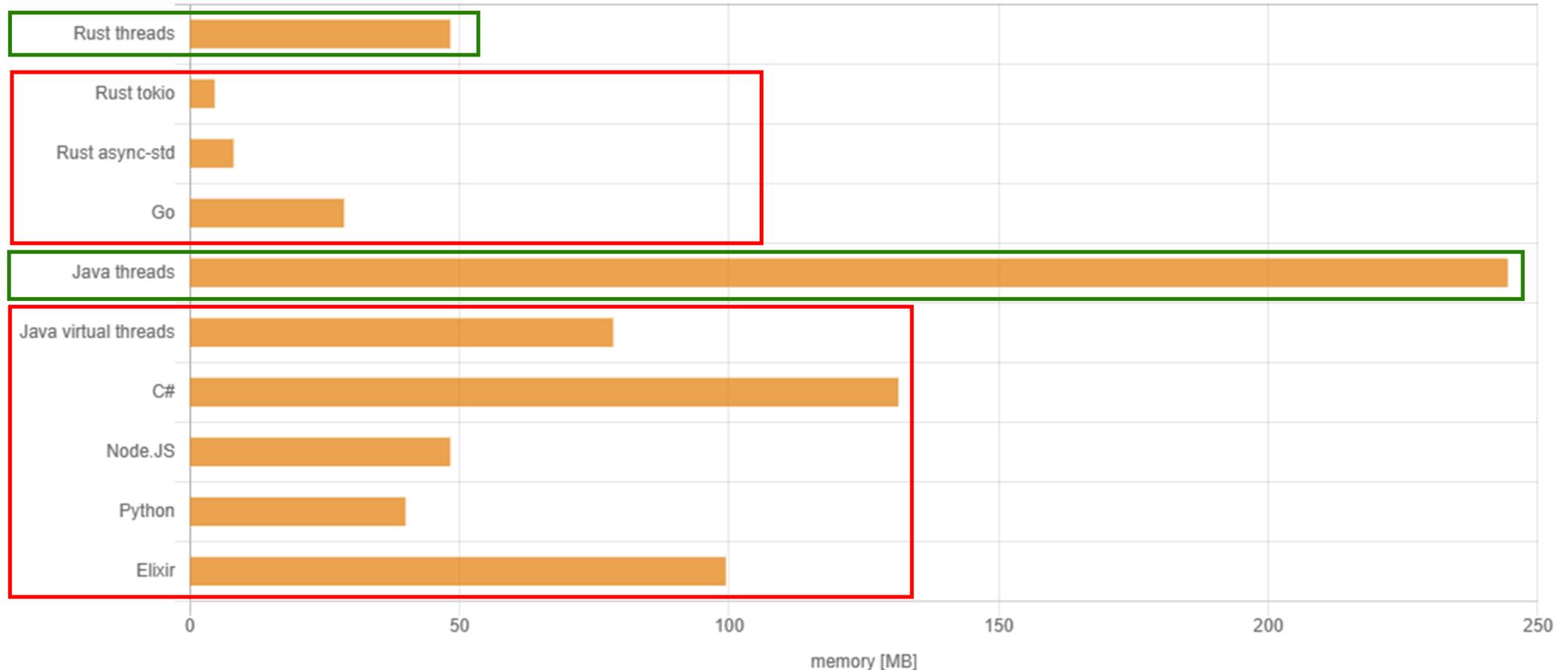
1. **Умеем только пользоваться сокетами:** неблокирующий I/O и busy waiting.
2. **Thread-per-connection:** используем потоки и блокирующий I/O.
3. **Reactor:** используем мультиплексирование I/O.

# Reactor

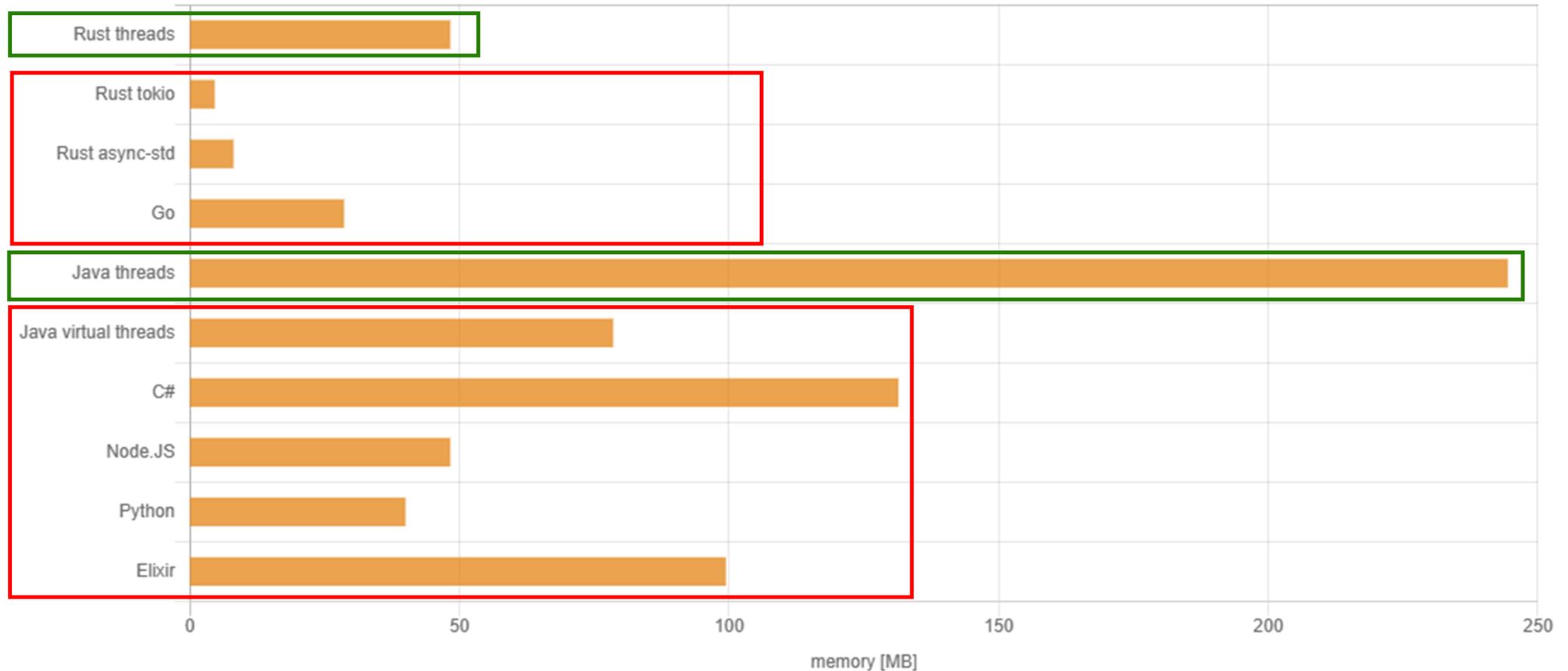
- ❑ Используем мультиплексирование I/O – можем эффективно обрабатывать множество одновременных соединений.
- ❑ Эффективно масштабируется, можем использовать thread-per-request модель.

```
1 int main() {
2     int nfd, server_fd, client_fd, n, epfd = epoll_create1(0);
3     struct epoll_event ev, cev, events[16];
4
5     ev.events = EPOLLIN;
6     ev.data.fd = server_fd;
7
8     epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);
9
10    while (1) {
11        nfd = epoll_wait(epfd, events, 16, -1);
12        for (int i = 0; i < nfd; ++i) {
13            if (events[i].data.fd == server_fd) {
14                client_fd = accept(server_fd, /* ... */);
15                cev = {
16                    .events = EPOLLIN,
17                    .data.fd = client_fd
18                };
19                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &cev);
20            } else {
21                char buffer[1024];
22                int bytes_read = read(client_fd, buffer, 1024);
23
24                handle(buffer, bytes_read);
25            }
26        }
27    }
28    return 0;
29 }
```

# RAM: 10k threads vs 10k coroutines in single thread



# RAM: 10k threads vs 10k coroutines in single thread



Больше: [How Much Memory Do You Need to Run 1 Million Concurrent Tasks?](#)

# Everything is a file!



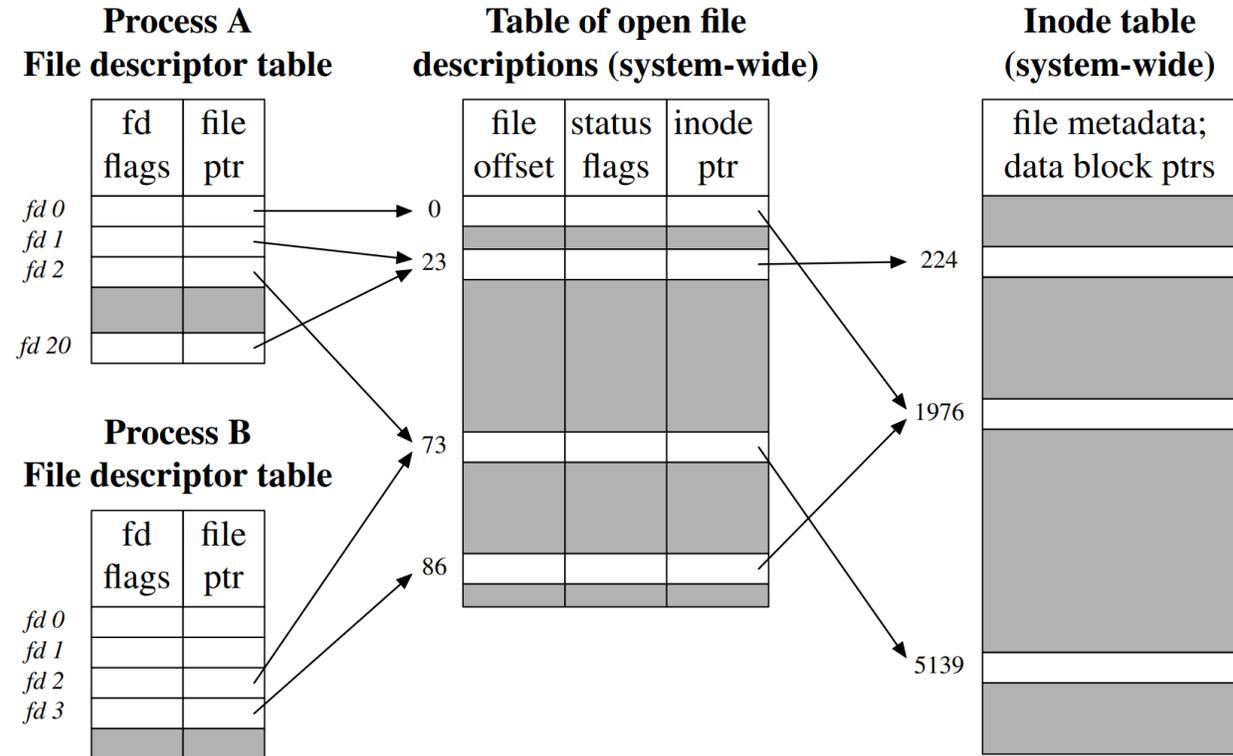
# Everything is a file!

- File descriptor – ключ в per-process таблице.

```
1 struct fdtable {
2     struct file __rcu **fd; /* current fd array */
3     unsigned long *open_fds;
4 };
```

# Everything is a file!

- ❑ File descriptor – ключ в per-process таблице.
- ❑ Значениями в таблице являются указатели на open file description.



# Everything is a file!

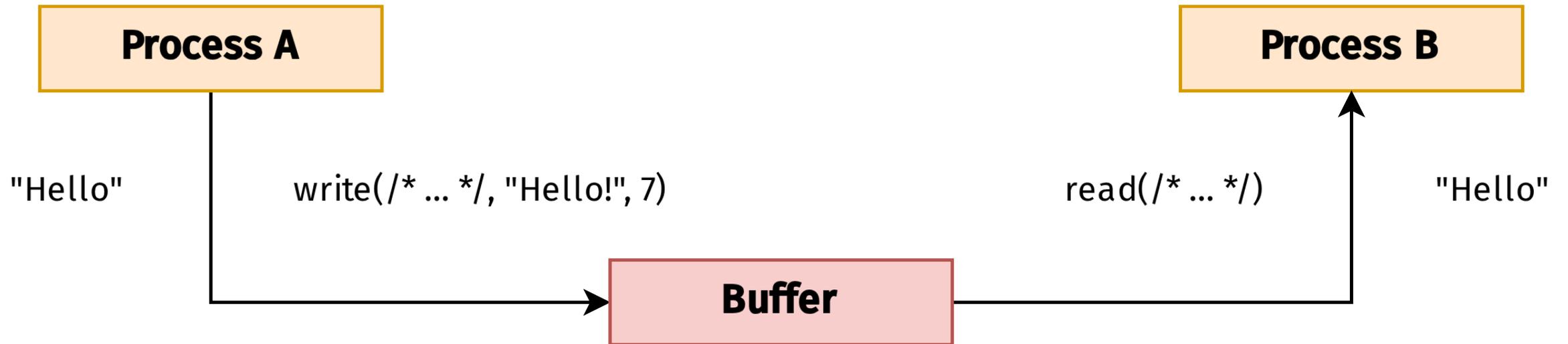
- ❑ Множество файловых дескрипторов могут указывать на один и тот же открытый файл.
- ❑ Набор допустимых для файла операций хранится в `struct file_operations`.

```
1 struct file {  
2     file_ref_t f_ref;  
3     const struct file_operations* f_op;  
4     struct inode *f_inode;  
5 };
```

# Everything is a file!

```
1 struct file_operations {
2     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
3     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
4     __poll_t (*poll) (struct file *, struct poll_table_struct *);
5 };
```

# pipe



# pipe\_poll

Реализация `__poll_t`  
`poll(struct file *,`  
`poll_table *)` для pipe.

```
1 static __poll_t
2 pipe_poll(struct file* filp, poll_table* wait) {
3     /* ... */
4     if (filp->f_mode & FMODE_READ)
5         poll_wait(filp, &pipe->rd_wait, wait);
6     /* ... */
7
8     mask = 0;
9     if (filp->f_mode & FMODE_READ) {
10         if (!pipe_empty(head, tail))
11             mask |= EPOLLIN | EPOLLRDNORM;
12         if (!pipe->writers && filp->f_pipe != pipe->w_counter)
13             mask |= EPOLLHUP;
14     }
15
16     return mask;
17 }
18
```

# poll\_wait

- ❑ Регистрирует через `_qproc` очередь ожидания файла как способную разбудить поток.
- ❑ Конкретная реализация `_qproc` отличается от используемого механизма: `epoll` или `select`.

```
1 static inline void
2 poll_wait(struct file* filp, wait_queue_head_t* wait_address, poll_table* p)
3 {
4     if (p && p->_qproc && wait_address)
5         p->_qproc(filp, wait_address, p);
6 }
```

# pipe\_poll

Реализация `__poll_t`  
`poll(struct file *,`  
`poll_table *)` для pipe.

```
1 static __poll_t
2 pipe_poll(struct file* filp, poll_table* wait) {
3     /* ... */
4     if (filp->f_mode & FMODE_READ)
5         poll_wait(filp, &pipe->rd_wait, wait);
6     /* ... */
7
8     mask = 0;
9     if (filp->f_mode & FMODE_READ) {
10         if (!pipe_empty(head, tail))
11             mask |= EPOLLIN | EPOLLRDNORM;
12         if (!pipe->writers && filp->f_pipe != pipe->w_counter)
13             mask |= EPOLLHUP;
14     }
15
16     return mask;
17 }
18
```

# pipe\_write

- ❑ Записывает данные в буфер.
- ❑ Пробуждает очередь ожидания на чтение.

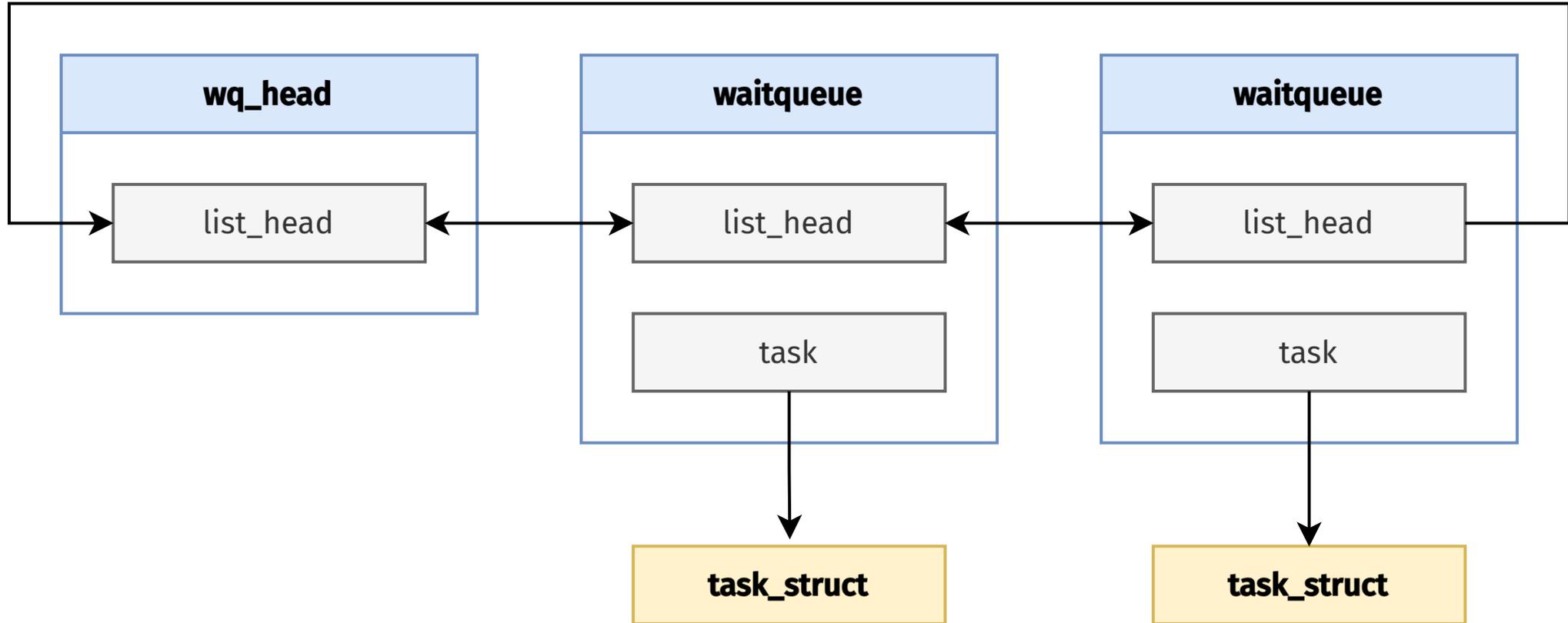
```
1 static ssize_t
2 pipe_write(struct kiocb* iocb, struct iov_iter* from) {
3     /* ...*/
4     if (was_empty || pipe->poll_usage)
5         wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN | EPOLLRDNORM);
6     kill_fasync(&pipe->fasync_readers, SIGIO, POLL_IN);
7     /* ... */
8 }
9
```

# poll\_wait

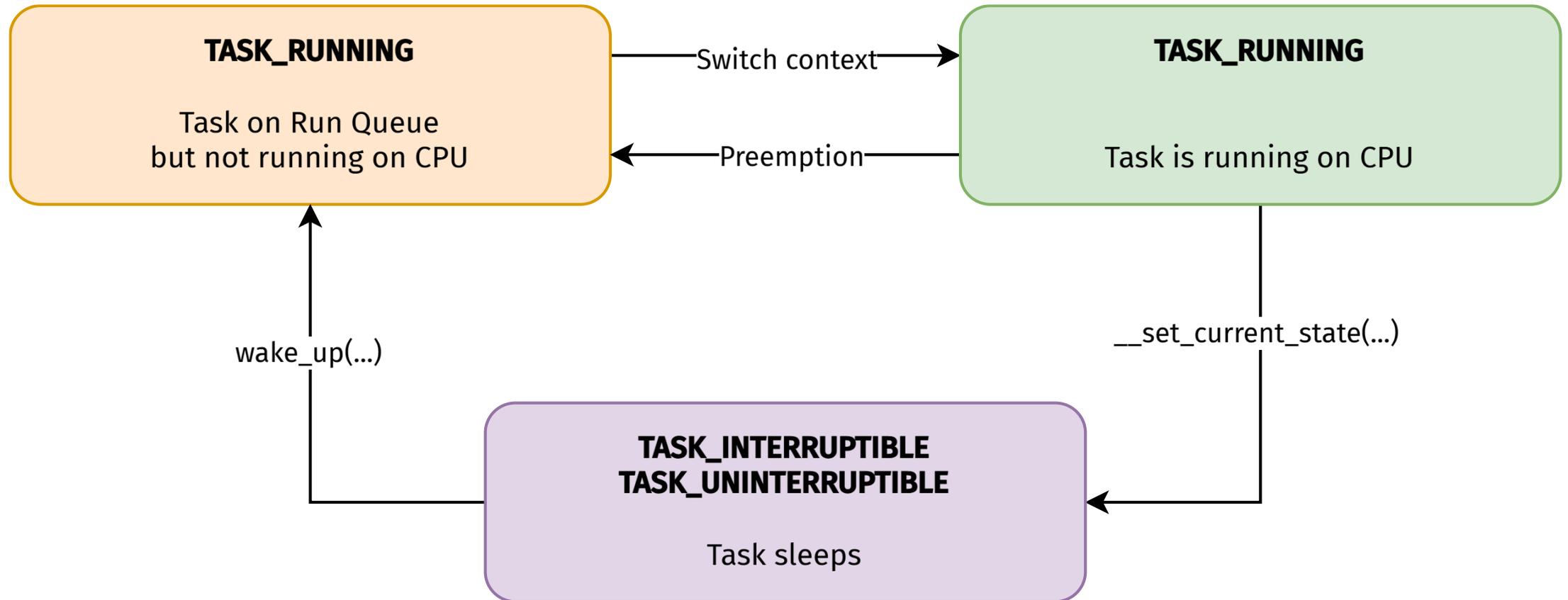
- ❑ Регистрирует через `_qproc` очередь ожидания файла как способную разбудить поток.
- ❑ Конкретная реализация `_qproc` отличается от используемого механизма: `epoll` или `select`.

```
1 static inline void
2 poll_wait(struct file* filp, wait_queue_head_t* wait_address, poll_table* p)
3 {
4     if (p && p->_qproc && wait_address)
5         p->_qproc(filp, wait_address, p);
6 }
```

# Wait queue



# Task state transition



# Способы мультиплексирования

- `select` и `poll`

# select and poll

- ❑ Часть POSIX.
- ❑ `select`: старше ядра Linux.
- ❑ `poll`: более user-friendly интерфейс.
- ❑ Одна и та же реализация в “select.c”.

```
1 int select(  
2     int n,  
3     fd_set *readfds,  
4     fd_set *writefds,  
5     fd_set *exceptfds,  
6     struct timeval* timeout  
7 );  
8  
9 int poll(  
10    struct pollfd *fds,  
11    nfds_t nfds,  
12    int timeout  
13 );
```

# select and poll

## Преимущества

- ❑ Часть Posix – гарантия портируемости.
- ❑ Могут быть эффективны на малом количестве дескрипторов.

## Недостатки

- ❑ `select`: невозможно передать >1024 file descriptors.
- ❑ Одна и та же реализация со сложностью  $O(n)$  от количества наблюдаемых файловых дескрипторов.
- ❑ Не умеет работать с обычными файлами?

# Способы мультиплексирования

- ❑ `select` и `poll`
- ❑ `epoll_*`

# epoll

- ❑ Linux-specific API, пришедший на замену `select()` и `poll()`.
- ❑ Эффективнее `poll()` при работе с большим количеством дескрипторов.
- ❑ Сложность `epoll_ctl` –  $O(\log n)$  от количества файловых наблюдаемых файловых дескрипторов.
- ❑ Сложность `epoll_wait` –  $O(n)$  от количества готовых файловых дескрипторов.

```
1 #include <sys/epoll.h>
2
3 int epoll_create(int size);
4 int epoll_create1(int flags);
5
6 int epoll_ctl(
7     int epfd,
8     int op,
9     int fd,
10    struct epoll_event *_Nullable event
11 );
12
13 int epoll_wait(
14     int epfd,
15     struct epoll_event events[.maxevents],
16     int maxevents,
17     int timeout
18 );
19
```

# epoll\_create

1. Инициализируем epoll и получаем его файловый дескриптор.

```
1 int main() {
2     int nfds, server_fd, client_fd, n, epfd = epoll_create1(0);
3     struct epoll_event ev, cev, events[16];
4
5     ev.events = EPOLLIN;
6     ev.data.fd = server_fd;
7
8     epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);
9
10    while (1) {
11        nfds = epoll_wait(epfd, events, 16, -1);
12        for (int i = 0; i < nfds; ++i) {
13            if (events[i].data.fd == server_fd) {
14                client_fd = accept(server_fd, /* ... */);
15                cev = {
16                    .events = EPOLLIN,
17                    .data.fd = client_fd
18                };
19                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &cev);
20            } else {
21                char buffer[1024];
22                int bytes_read = read(client_fd, buffer, 1024);
23
24                handle(buffer, bytes_read);
25            }
26        }
27    }
28    return 0;
29 }
```

# eventpoll

- ❑ Создаётся при вызове `epoll_create()`.
- ❑ Хранит наблюдаемые файловые дескрипторы в красно-черном дереве.

```
1 struct eventpoll {
2     /* ... */
3     wait_queue_head_t wq;
4     wait_queue_head_t poll_wait;
5     struct list_head rdllist;
6     struct rb_root_cached rbr;
7     /* ... */
8 };
```

# eventpoll

- ❑ Создаётся при вызове `epoll_create()`.
- ❑ Хранит наблюдаемые файловые дескрипторы в красно-черном дереве.
- ❑ Вдобавок к наблюдаемым файловым дескрипторам, хранит список готовых файловых дескрипторов.

```
1 struct eventpoll {
2     /* ... */
3     wait_queue_head_t wq;
4     wait_queue_head_t poll_wait;
5     struct list_head rdllist;
6     struct rb_root_cached rbr;
7     /* ... */
8 };
```

# epoll\_ctl

1. Инициализируем epoll и получаем его файловый дескриптор.
2. Добавляем интересующие нас файловые дескрипторы в epoll.

```
1 int main() {
2     int nfd, server_fd, client_fd, n, epfd = epoll_create1(0);
3     struct epoll_event ev, cev, events[16];
4
5     ev.events = EPOLLIN;
6     ev.data.fd = server_fd;
7
8     epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);
9
10    while (1) {
11        nfd = epoll_wait(epfd, events, 16, -1);
12        for (int i = 0; i < nfd; ++i) {
13            if (events[i].data.fd == server_fd) {
14                client_fd = accept(server_fd, /* ... */);
15                cev = {
16                    .events = EPOLLIN,
17                    .data.fd = client_fd
18                };
19                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &cev);
20            } else {
21                char buffer[1024];
22                int bytes_read = read(client_fd, buffer, 1024);
23
24                handle(buffer, bytes_read);
25            }
26        }
27    }
28    return 0;
29 }
```

# ep\_insert

- ❑ Вызывается внутри `epoll_ctl()` при добавлении нового файлового дескриптора.
- ❑ Создаёт экземпляр `struct epitem`, представляющий конкретный файл и добавляет его в `rbr`.

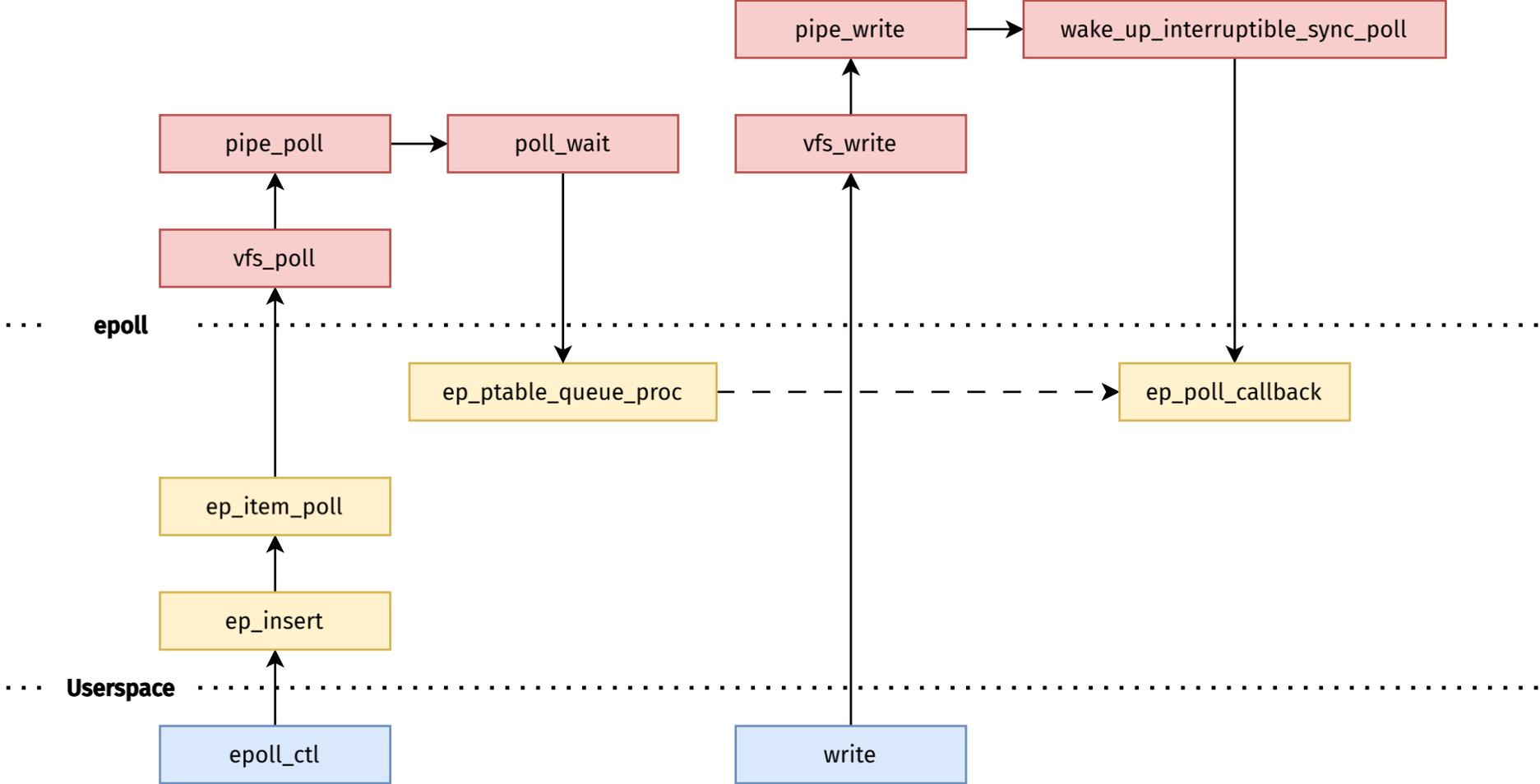
```
1 static int
2 ep_insert(struct eventpoll *ep,
3           const struct epoll_event *event,
4           struct file *tfile,
5           int fd,
6           int full_check) {
7     struct epitem *epi;
8     struct ep_queue epq;
9     struct eventpoll *tep = NULL;
10
11     /* ... */
12     if (!(epi = kmem_cache_zalloc(/* ... */))) {
13         return -ENOMEM;
14     }
15
16     INIT_LIST_HEAD(&epi->rdllink);
17     epi->ep = ep;
18     ep_set_ffd(&epi->ffd, tfile, fd);
19     epi->event = *event;
20     epi->next = EP_UNACTIVE_PTR;
21     epq.epi = epi;
22
23     init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);
24     revents = ep_item_poll(epi, &epq.pt, 1);
25     /* ... */
26 }
```

# ep\_insert

- ❑ Вызывается внутри `epoll_ctl()` при добавлении нового файлового дескриптора.
- ❑ Создаёт экземпляр `struct epitem`, представляющий конкретный файл и добавляет его в `rbr`.
- ❑ Вызывает `poll()` на файле.

```
1 static int
2 ep_insert(struct eventpoll *ep,
3           const struct epoll_event *event,
4           struct file *tfile,
5           int fd,
6           int full_check) {
7     struct epitem *epi;
8     struct ep_queue epq;
9     struct eventpoll *tep = NULL;
10
11     /* ... */
12     if (!(epi = kmem_cache_zalloc(/* ... */))) {
13         return -ENOMEM;
14     }
15
16     INIT_LIST_HEAD(&epi->rdllink);
17     epi->ep = ep;
18     ep_set_ffd(&epi->ffd, tfile, fd);
19     epi->event = *event;
20     epi->next = EP_UNACTIVE_PTR;
21     epq.epi = epi;
22
23     init_poll_funcptr(&epq.pt, ep_ptable_queue_proc);
24     revents = ep_item_poll(epi, &epq.pt, 1);
25     /* ... */
26 }
```

# Call Flow for ep\_item\_poll



# epoll\_wait

1. Инициализируем epoll и получаем его файловый дескриптор.
2. Добавляем интересующие нас файловые дескрипторы в epoll.
3. Блокируем текущий поток.

```
1 int main() {
2     int nfd, server_fd, client_fd, n, epfd = epoll_create1(0);
3     struct epoll_event ev, cev, events[16];
4
5     ev.events = EPOLLIN;
6     ev.data.fd = server_fd;
7
8     epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);
9
10    while (1) {
11        nfd = epoll_wait(epfd, events, 16, -1);
12        for (int i = 0; i < nfd; ++i) {
13            if (events[i].data.fd == server_fd) {
14                client_fd = accept(server_fd, /* ... */);
15                cev = {
16                    .events = EPOLLIN,
17                    .data.fd = client_fd
18                };
19                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &cev);
20            } else {
21                char buffer[1024];
22                int bytes_read = read(client_fd, buffer, 1024);
23
24                handle(buffer, bytes_read);
25            }
26        }
27    }
28    return 0;
29 }
```

# ep\_poll

- ❑ Вызывается через `epoll_wait()`.
- ❑ Копирует список готовых дескрипторов в User Space.

```
1 static int
2 ep_poll(struct eventpoll* ep,
3         struct epoll_event __user* events,
4         int maxevents,
5         struct timespec64* timeout) {
6     /* ... */
7     eavail = ep_events_available(ep);
8
9     while (1) {
10        if (eavail) {
11            res = ep_send_events(ep, events, maxevents);
12            if (res) {
13                /* .. */
14                return res;
15            }
16        }
17
18        /* ... */
19        eavail = ep_busy_loop(ep, timed_out);
20        if (eavail)
21            continue;
22
23        __set_current_state(TASK_INTERRUPTIBLE);
24        /* ... */
25        __set_current_state(TASK_RUNNING);
26
27        eavail = 1;
28        /* ... */
29    }
30 }
```

# ep\_poll

- Вызывается через `epoll_wait()`.
- Копирует список готовых дескрипторов в User Space.
- Может напрямую использовать NAPI на сокетах.
- В ином случае снимает поток с выполнения.

```
1 static int
2 ep_poll(struct eventpoll* ep,
3         struct epoll_event __user* events,
4         int maxevents,
5         struct timespec64* timeout) {
6     /* ... */
7     eavail = ep_events_available(ep);
8
9     while (1) {
10        if (eavail) {
11            res = ep_send_events(ep, events, maxevents);
12            if (res) {
13                /* .. */
14                return res;
15            }
16        }
17
18        /* ... */
19        eavail = ep_busy_loop(ep, timed_out);
20        if (eavail)
21            continue;
22
23        __set_current_state(TASK_INTERRUPTIBLE);
24        /* ... */
25        __set_current_state(TASK_RUNNING);
26
27        eavail = 1;
28        /* ... */
29    }
30 }
```

## ep\_busy\_loop

- ❑ Должен быть включен NAPI при сборке ядра.
- ❑ Вызывает napi\_poll.
- ❑ Позволяет снизить latency с помощью interrupt coalescing.

```
1 static bool
2 ep_busy_loop(struct eventpoll *ep, int nonblock) {
3     unsigned int napi_id = /* ... */;
4     u16 budget = /* ... */;
5
6     if (!budget)
7         budget = BUSY_POLL_BUDGET;
8
9     if (napi_id ≥ MIN_NAPI_ID && ep_busy_loop_on(ep)) {
10        napi_busy_loop(napi_id, nonblock ? NULL : ep_busy_loop_end,
11                       ep, prefer_busy_poll, budget);
12        if (ep_events_available(ep))
13            return true;
14
15        /* ... */
16    }
17    return false;
18 }
```

# epoll

## Преимущества

- ❑ Эффективнее чем `select` и `poll` на большом количестве дескрипторов.
- ❑ Особенно эффективен при большом количестве дескрипторов и малом количестве событий.
- ❑ С помощью файлового дескриптора можно группировать различные `epoll`'ы.

## Недостатки

- ❑ Высокая стоимость на поддержание внутреннего состояния.

# epoll

## Преимущества

- ❑ Эффективнее чем `select` и `poll` на большом количестве дескрипторов.
- ❑ Особенно эффективен при большом количестве дескрипторов и малом количестве событий.
- ❑ С помощью файлового дескриптора можно группировать различные `epoll`'ы.

## Недостатки

- ❑ Высокая стоимость на поддержание внутреннего состояния.
- ❑ Не умеет работать с обычными файлами?

# syscalls

Можем ли мы  
оптимизировать количество  
системных вызовов?

```
1 int main() {
2     int nfd, server_fd, client_fd, n, epfd = epoll_create1(0);
3     struct epoll_event ev, cev, events[16];
4
5     ev.events = EPOLLIN;
6     ev.data.fd = server_fd;
7
8     epoll_ctl(epfd, EPOLL_CTL_ADD, server_fd, &ev);
9
10    while (1) {
11        nfd = epoll_wait(epfd, events, 16, -1);
12        for (int i = 0; i < nfd; ++i) {
13            if (events[i].data.fd == server_fd) {
14                client_fd = accept(server_fd, /* ... */);
15                cev = {
16                    .events = EPOLLIN,
17                    .data.fd = client_fd
18                };
19                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &cev);
20            } else {
21                char buffer[1024];
22                int bytes_read = read(client_fd, buffer, 1024);
23
24                handle(buffer, bytes_read);
25            }
26        }
27    }
28    return 0;
29 }
```

# epoll

## Преимущества

- ❑ Эффективнее чем `select` и `poll` на большом количестве дескрипторов.
- ❑ Особенно эффективен при большом количестве дескрипторов и малом количестве событий.
- ❑ С помощью файлового дескриптора можно группировать различные `epoll`'ы.

## Недостатки

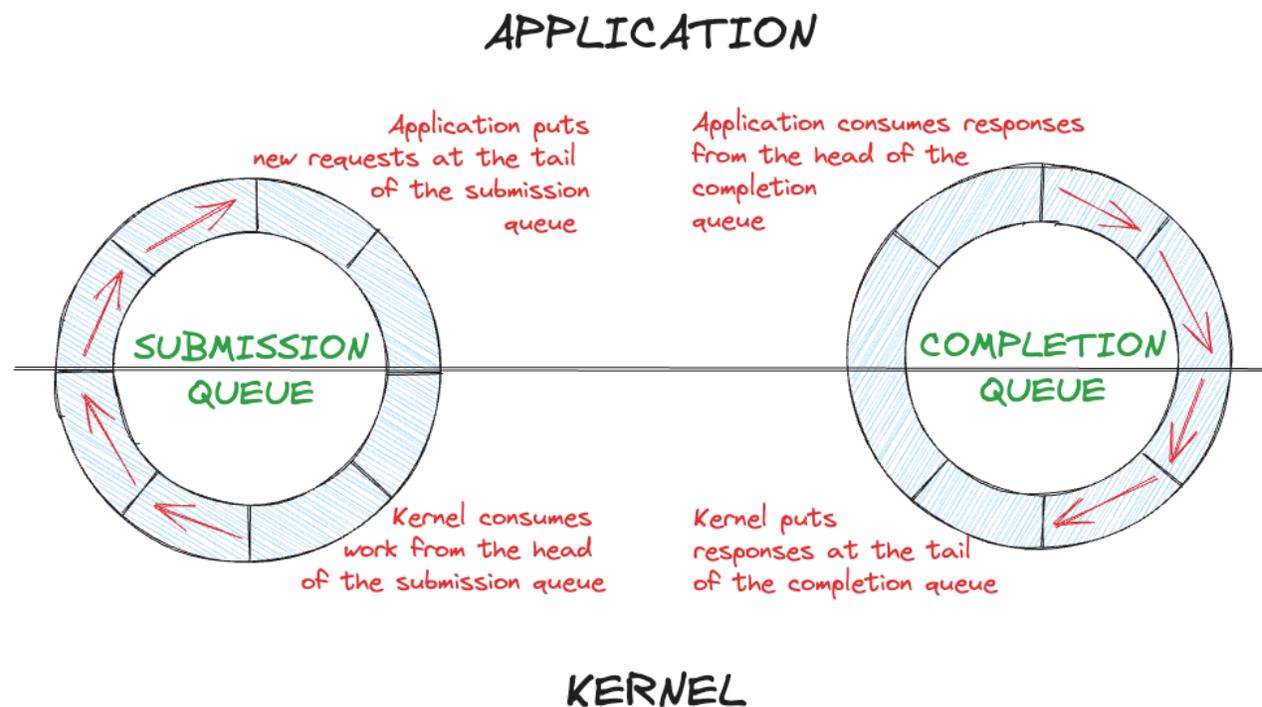
- ❑ Высокая стоимость на поддержание внутреннего состояния.
- ❑ Не умеет работать с обычными файлами?
- ❑ Слишком частое обращение к системным вызовам?

# Способы мультиплексирования

- ❑ `select` и `poll`
- ❑ `epoll_*`
- ❑ `io_uring`

# io\_uring

- ❑ Linux-specific API для асинхронного I/O.
- ❑ Прячем мультиплексирование внутри ядра.
- ❑ Избавляемся от системных вызовов в Userspace.
- ❑ Взаимодействуем с ядром через две очереди в shared memory.



# io\_uring

1. Инициализируем io\_uring и очереди.

```
1 int main() {
2     struct io_uring ring;
3     struct io_uring_sqe* sqe;
4     struct io_uring_cqe* cqe;
5     char buffer[1024];
6     int fd;
7
8     io_uring_queue_init(1, &ring, 0);
9     fd = open("example.txt", O_RDONLY);
10
11     sqe = io_uring_get_sqe(&ring);
12
13     io_uring_prep_read(sqe, fd, buffer, 1024, 0);
14     io_uring_submit(&ring);
15     io_uring_wait_cqe(&ring, &cqe);
16
17     printf("Read %d bytes: %.*s\n", cqe->res, cqe->res, buffer);
18
19     io_uring_cqe_seen(&ring, cqe);
20     io_uring_queue_exit(&ring);
21     close(fd);
22
23     return 0;
24 }
25
```

# io\_uring

1. Инициализируем io\_uring и очереди.
2. Подготавливаем параметры для операции и планируем её.

```
1 int main() {
2     struct io_uring ring;
3     struct io_uring_sqe* sqe;
4     struct io_uring_cqe* cqe;
5     char buffer[1024];
6     int fd;
7
8     io_uring_queue_init(1, &ring, 0);
9     fd = open("example.txt", O_RDONLY);
10
11     sqe = io_uring_get_sqe(&ring);
12
13     io_uring_prep_read(sqe, fd, buffer, 1024, 0);
14     io_uring_submit(&ring);
15     io_uring_wait_cqe(&ring, &cqe);
16
17     printf("Read %d bytes: %.*s\n", cqe->res, cqe->res, buffer);
18
19     io_uring_cqe_seen(&ring, cqe);
20     io_uring_queue_exit(&ring);
21     close(fd);
22
23     return 0;
24 }
25
```

# io\_uring

1. Инициализируем io\_uring и очереди.
2. Подготавливаем параметры для операции и планируем её.
3. Сообщаем ядру, что мы обновили очередь и блокируем текущий поток.

```
1 int main() {
2     struct io_uring ring;
3     struct io_uring_sqe* sqe;
4     struct io_uring_cqe* cqe;
5     char buffer[1024];
6     int fd;
7
8     io_uring_queue_init(1, &ring, 0);
9     fd = open("example.txt", O_RDONLY);
10
11     sqe = io_uring_get_sqe(&ring);
12
13     io_uring_prep_read(sqe, fd, buffer, 1024, 0);
14     io_uring_submit(&ring);
15     io_uring_wait_cqe(&ring, &cqe);
16
17     printf("Read %d bytes: %.*s\n", cqe->res, cqe->res, buffer);
18
19     io_uring_cqe_seen(&ring, cqe);
20     io_uring_queue_exit(&ring);
21     close(fd);
22
23     return 0;
24 }
25
```

# io\_uring

## Преимущества

- ❑ Даёт готовый фреймворк для работы с асинхронным I/O.
- ❑ Позволяет в одинаковой манере работать с обычными файлами и файлами устройств.
- ❑ Позволяет экономить на системных вызовах.

## Недостатки

- ❑ Находится всё ещё в состоянии активной разработки.

# Заключение

1. **Используйте `select` или `poll`, если:**
  - A. вам нужна гарантия портируемости;
  - B. количество файловых дескрипторов мало ( $<10000$ ).

# Заключение

- 1. Используйте select или poll, если:**
  - A. вам нужна гарантия портируемости;
  - B. количество файловых дескрипторов мало ( $<10000$ ).
- 2. Используйте epoll, если:**
  - A. количество файловых дескрипторов будет велико ( $>10000$ ).

# Заключение

## 1. Используйте `select` или `poll`, если:

- A. вам нужна гарантия портируемости;
- B. количество файловых дескрипторов мало ( $<10000$ ).

## 2. Используйте `epoll`, если:

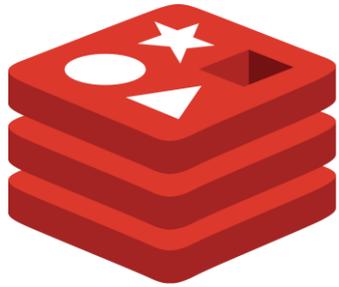
- A. количество файловых дескрипторов будет велико ( $>10000$ ).

## 3. Используйте `io_uring`, если:

- A. вы хотите использовать готовый фреймворк для асинхронного I/O;
- B. вы хотите асинхронно работать с файлами;
- C. вы не боитесь использовать код в состоянии активной разработки.



Где используется?



redis



**Спасибо за внимание!**