Мультиплексирование I/O

“Медленные” системные вызовы

ssize_t **read**(int fd, void *buf, size_t count);

- когда нечего читать

ssize_t **write**(int fd, const void *buf, size_t count);

- когда нет места в канале

открытие FIFO на запись, пока не появится второй процесс, готовый читать

...

Решение на основе процессов или нитей

+ легко реализуются
- ресурсоемки
- сложны для межпроцессного взаимодействия
- потоки лучше чем процессы, но переключение контекста и стек имеют цену

Неблокирующий ввод-вывод

O_NONBLOCK при создании, если дескриптор уже открыт, то:

```
flags = fcntl(fd, GET_FL);
fcntl(fd, SET_FL, flags|O_NONBLOCK )
```

read, write, accept возвращают EAGAIN вместо блокировки

Проблемы неблокирующего решения

- редкое опрашивание дескрипторов ведет к задержкам
- частое опрашивание тратит ресурсы процессора

Требуется поддержка со стороны ядра, процесс спит и не потребляет ресурсы, как только дескрипторы готовы ядро будит процесс

Мултиплексирование

Мультиплексирование - процесс сразу отслеживает много дескрипторов, и определяет можно ли в них писать

- **select** - работает с множеством файловых дескрипторов
- **poll** - работает с массивами файловых дескприпторов

Стандартизованы, нагрузка на ядро растет практически линейно с увеличением числа отслеживаемых дескрипторов

- **epoll** (Lin 2.6), kqueue (*BSD, Mac), /dev/poll (Solaris)

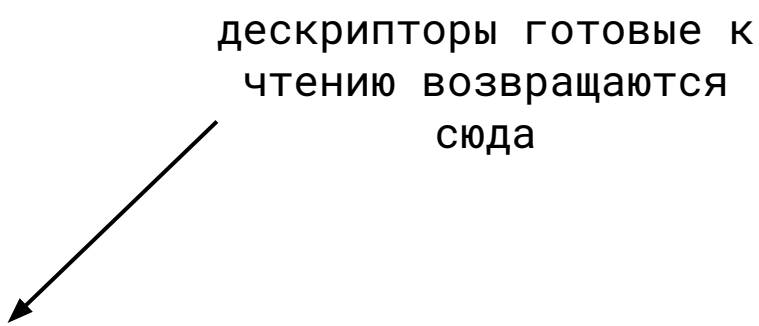
Могут отслеживать множество дескрипторов, нагрузка на ядро зависит от количества событий ввода-вывода

SELECT

```
int select(int nfds, fd_set *readfds,   //чтение
                     fd_set *writefds,  //запись
                     fd_set *exceptfds, //искл.ситуация
                     struct timeval *timeout);

FD_SETSIZE в линукс по умолчанию равен 1024
fd_set *set -- хранения набора ф.д. (1 бит на дескриптор)

void FD_ZERO(fd_set *set);          // обнулить набор
void FD_SET(int fd, fd_set *set);   // установить
void FD_CLR(int fd, fd_set *set);   // обнулить деск. в
наборе
int  FD_ISSET(int fd, fd_set *set); // проверить готовность

при успехе возвращает сумму готовых дескрипторов всех
наборов
```

SELECT (пример)

```
fd_set readfds;

while(1) {
    FD_ZERO(&readfds)
    FD_SET(ourfd, &readfds)
    ...
    res = select(MAX_FDS+1, &readfds, NULL, NULL, NULL,
NULL);

    for(fd) {
        if (FD_ISSET(*fd, &readfds)) { … }
    }
    ...
}
```



Проблема: нужно заново инициализировать после каждой итерации

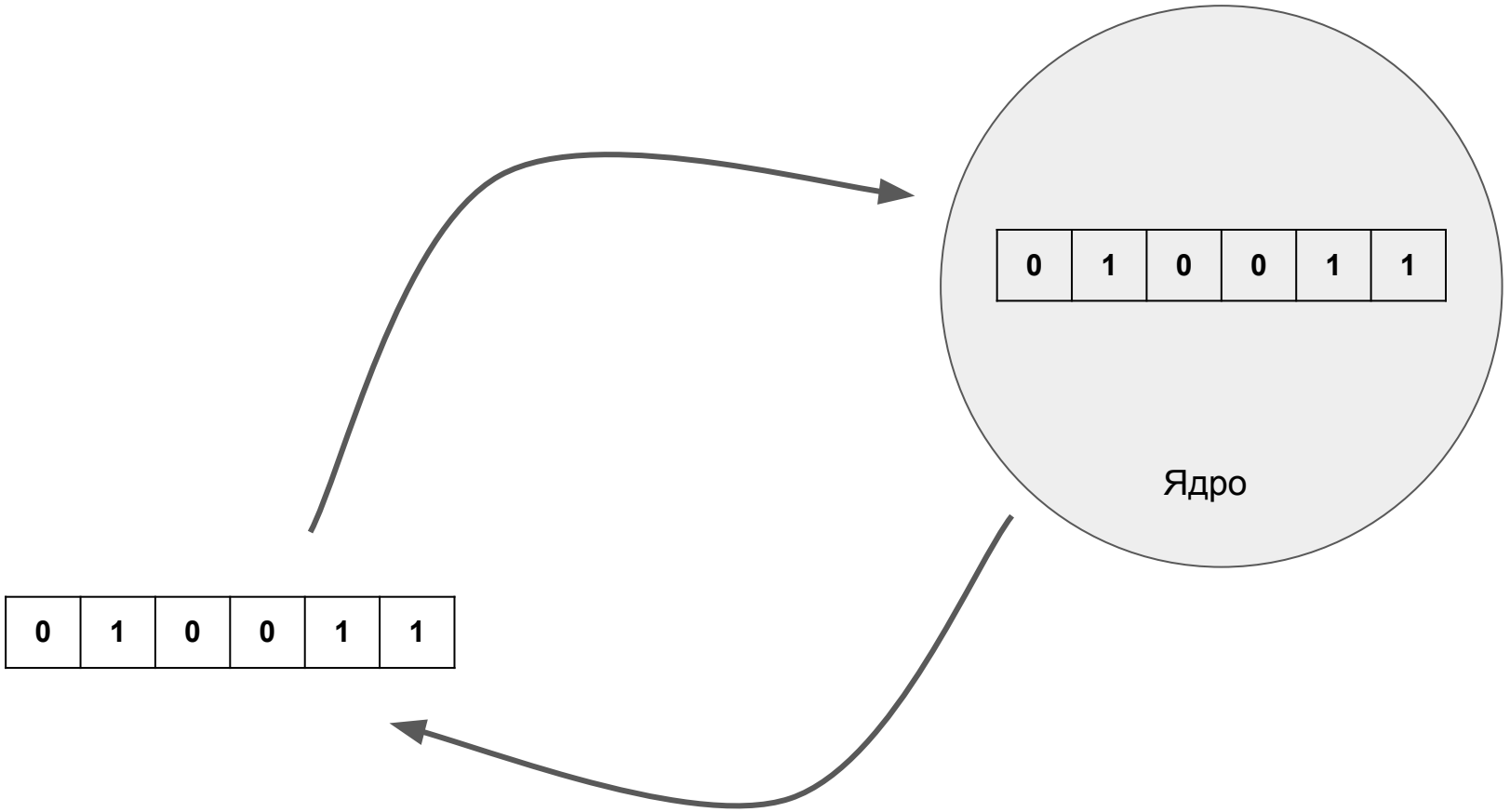
POLL

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
 -  количество дескрипторов в отличии от select по умолчанию
    не ограничено, но объем данных пересылаемых в ядро
    больше.
 -  Массив структур, а не множество:
struct pollfd {
    int   fd;       /* файловый дескриптор */
    short events;   /* запрашиваемые события */
    short revents; /* возвращённые события */
};

POLLIN  Есть данные для чтения.
POLLOUT Есть для записи, но может привести к блокировке,
если дескриптор не O_NONBLOCK
...
```

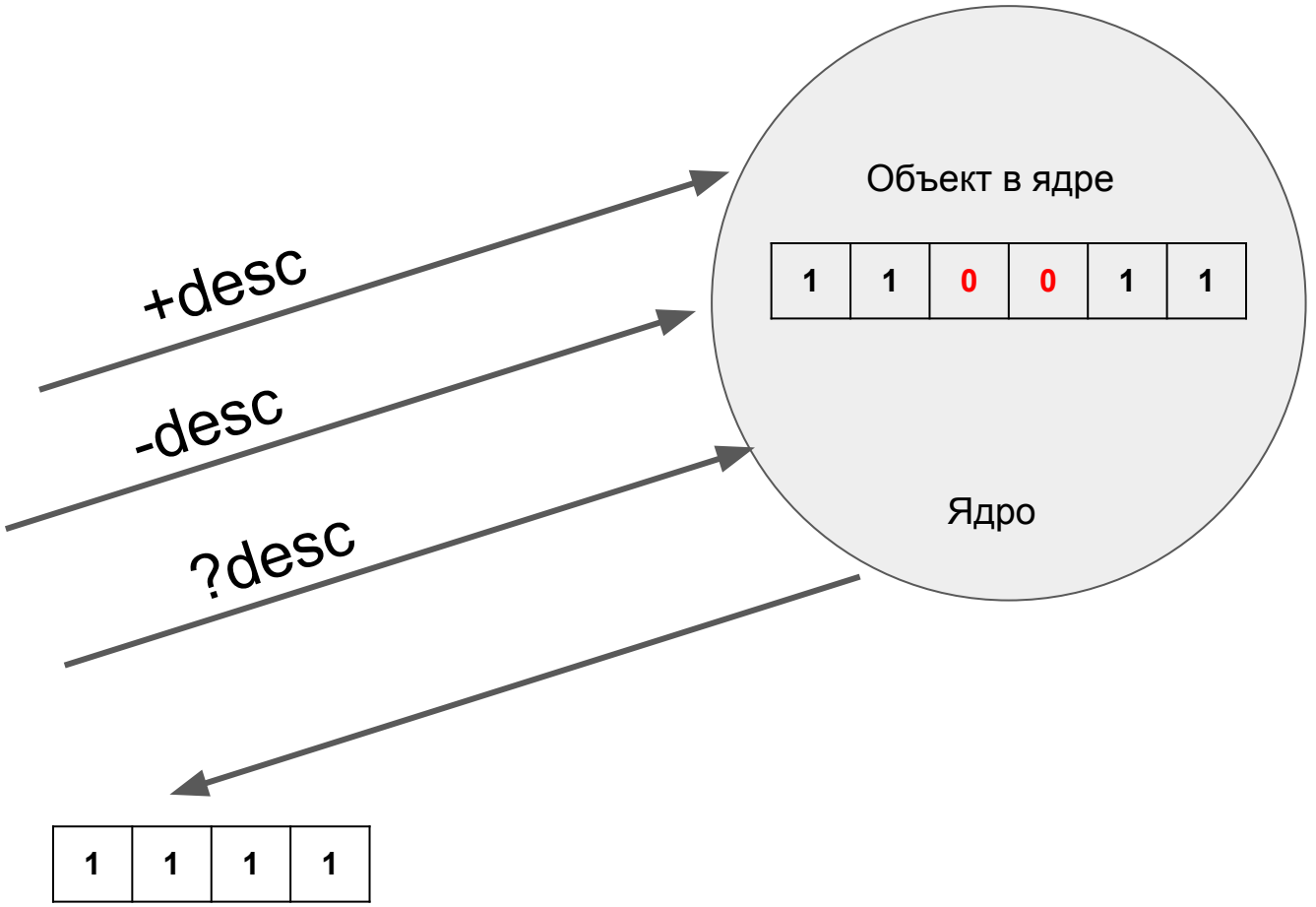
POLL (пример)

```
struct pollfd pollDesc[10];
pollDesc[0].fd = ourfd;
pollDesc[0].events = POLLIN;
...
while(1) {
    res = poll(pollDesc, numDesc, -1);
    for (...) {
        if (pollDesc[i].revents & POLLIN) {
            ...
        }
    }
}
```

Недостатки SELECT и POLL



Чего хотелось бы



Только готовые дескрипторы

EPOLL

```
int epoll_create1(int flags); flags = {0, EPOLL_CLOEXEC}
 -  создает объект в ядре, возвращает дескриптор
int epoll_ctl(int epfd, int op, int fd,
                              struct epoll_event *event);
op ⇒ EPOLL_CTL_ADD, EPOLL_CTL_MOD, EPOLL_CTL_DEL

typedef union epoll_data {
    void        *ptr;
    int         fd;
    uint32_t    u32;
    uint64_t    u64;
} epoll_data_t;

struct epoll_event {
    uint32_t    events;     /* События epoll */
    epoll_data_t data;      /* Переменная для данных пользователя */
};
```

EPOLL

```
int epoll_wait(int epfd, struct epoll_event *events,
                                int maxevents, int timeout);
```

- возвращает:
  0 - сработал таймаут
  -1 - ошибка
  > 0 - количество событий

- **events** - массив событий, его размер не больше **maxevents**

+ Позволяет обрабатывать множество дескрипторов
+ Не нужно делать постоянные пересылки дескрипторов в ядро
+ Сильно быстрее чем SELECT или POLL
+ level-triggered & edge-triggered

Библиотеки

libevent

libev
- хорошо, но под Linux

libuv
- node.js, абстракция над libev (Linux) или IOCP (Windows)

Boost.Asio

...