

# Задание первое

В первом задании необходимо реализовать примитивный вариант утилиты `cat`, копирующий символы из `stdin` на `stdout`.

## Первая часть

Требуется реализовать динамическую библиотеку, содержащую функции-хелперы `read_` и `write_`, делающие то же, что и `read` и `write`, но для буфера целиком (либо до EOF). Сигнатуры хелперов должны совпадать с сигнатурами оригинальных функций.

### Файлы в репозитории

- `/lib/helpers.h`
- `/lib/helpers.c`
- `/lib/Makefile`

### Скриптом сборки генерируется

- `/lib/libhelpers.so`

### Hints

- `man 2 read`
- `man 2 write`

## Вторая часть

Используя функции-хелперы из первой части, реализовать утилиту `cat`. В качестве аргумента `fd` хелперам `read_` и `write_` необходимо передавать `STDIN_FILENO` и `STDOUT_FILENO` соответственно.

### Файлы в репозитории

- `/cat/cat.c`
- `/cat/Makefile`

### Скриптом сборки генерируется

- `/cat/cat`

### Пример работы

- `./cat < cat.c > cat2.c && diff cat.c cat2.c && echo OK`

## Дедлайн

- 11 марта, 06:00 (GMT+3)

# Задание второе

Требуется реализовать утилиту `revwords`, читающую слова из `stdin` и выводящую в `stdout` эти же слова развёрнутыми. Гарантируется, что каждое слово имеет длину не более 4096 байт.

## Первая часть

Добавить в библиотеку функцию `read_until`, имеющую следующую сигнатуру:

```
ssize_t read_until(int fd, void * buf, size_t count, char delimiter);
```

Функция имеет ту же семантику, что и `read_` из первого задания, с одним отличием: она прекращает считывание из `fd`

не только при заполнении буфера, но и при наличии символа `delimiter` в уже заполненной части буфера.

## Файлы в репозитории

- `/lib/helpers.h`
- `/lib/helpers.c`
- `/lib/Makefile`

## Скриптом сборки генерируется

- `/lib/libhelpers.so`

## Вторая часть

Используя функции из библиотеки, реализовать требуемую утилиту. Если в буфере есть слово, которое может быть выведено на `stdout`, утилита не должна ожидать данных из `stdin`.

Слова отделяются друг от друга пробелом. Все прочие символы, в том числе `\n` и `\t`, считаются буквой.

## Файлы в репозитории

- `/revwords/revwords.c`
- `/revwords/Makefile`

## Скриптом сборки генерируется

- `/revwords/revwords`

## Пример использования

Команда (`echo -ne "abc def\ngh" ; sleep 3; echo -ne " qwer" | ./revwords`) должна вести себя следующим образом: вывести `cba_`, три секунды ничего не делать, вывести `hg\nfed_rewq`, где знаком `_` обозначается пробел.

## Дедлайн

- 18 марта, 06:00 (GMT+3)

## Задание третье

Реализовать утилиту `filter`, читающую строки из `stdin`, передающую эти строки как последний аргумент команде, указанной в `argv`, и выводящую только те из них, на которых команда завершилась с нулевым кодом возврата.

## Часть первая

Добавить в библиотеку следующую функцию:

```
int spawn(const char * file, char * const argv [])
```

Функция должна запускать исполняемый файл `file`, выбираемый в соответствии с переменной окружения [PATH](#), с аргументами, задаваемыми в `argv`, дожидаться её завершения и возвращать её код возврата.

Например, если специально не предпринимать против этого мер, следующий код

```
char* args[] = {"ls", "/bin", NULL};
int res = spawn("ls", args);
```

должен показывать содержимое директории `/bin` и возвращать `res = 0`.

## Файлы в репозитории

- `/lib/helpers.h`
- `/lib/helpers.c`

- /lib/Makefile

## Скриптом сборки генерируется

- /lib/libhelpers.so

## Hints

- man 2 fork
- man 3 exec
- man 2 wait

## Запрещается использовать

- man 3 system

## Часть вторая

Используя библиотеку, реализовать требуемую утилиту.

Строки отделяются друг от друга `\n`.

## Файлы в репозитории

- /filter/filter.h
- /filter/filter.c
- /filter/Makefile

## Скриптом сборки генерируется

- /filter/filter

## Пример использования

- Вызов `echo -ne "/bin/sh\n/blablabla\n/bin/cat\n" | ./filter tar cf /tmp/filter.tar` **должен вывести** `/bin/sh` и `/bin/cat`.

## Дедлайн

- 25 марта, 06:00 (GMT+3)

# Задание четвёртое

Требуется реализовать аналог буферизированного ввода-вывода из `stdio.h`, но без аналога типу `FILE` и с другим интерфейсом (и вообще не очень похоже, но ведь со знакомыми ключевыми словами спокойнее).

## Первая часть

Требуется реализовать динамическую библиотеку содержащую:

- `struct buf_t {...}` — буфер с максимальным размером хранимых данных далее именуемым `capacity` и заполненный на число байт далее именуемое `size`.
- `struct buf_t *buf_new(size_t capacity)` — конструктор пустого буфера, возвращает `NULL` если не удалось сделать `malloc`,
- `void buf_free(struct buf_t *)` — деструктор.
- `size_t buf_capacity(buf_t *)` — возвращает максимальный возможный размер,
- `size_t buf_size(buf_t *)` — возвращает текущую заполненность,
- `ssize_t buf_fill(fd_t fd, buf_t *buf, size_t required)` — заполняет буфер `read`ами до тех пор пока его `size` не станет как минимум `required` байт (если может больше — читает больше) или не наступит EOF. Возвращает:
  - `-1` при ошибке, ошибка в `errno`,
  - текущий `size` если не ошибка:

- `>= required` если всё удалось,
- `< required` если рановато наступил EOF.

Если в буфере не хватает `capacity`, то поведение не определено.

Заметьте, что `buf_fill(fd, buf, 1)` для пустого буфера должно пытаться заполнить весь буфер, `buf_fill(fd, buf, 1)` для буфера в котором есть хотя бы один байт должно делать ровно ничего, а `buf_fill(fd, buf, buf_capacity(buf))` должно заполнять весь буфер, если это вообще возможно.

- `ssize_t buf_flush(fd_t fd, buf_t *buf, size_t required)` — выписывает данные из буфера до тех пор, пока не будет записано как минимум `required` байт (больше — так больше) или буфер не опустеет. Возвращает:
  - `-1` при ошибке, ошибка в `errno`,
  - предыдущий `size` - текущий `size` если не ошибка:
    - `>= required` если всё удалось,
    - `< required` если в буфере столько нету.

Удачно записанные данные выбрасываются из буфера.

Заметьте, что `buf_flush(fd, buf, 1)` должно пытаться записать весь буфер, а `buf_flush(fd, buf, buf_size(buf))` должно записывать весь буфер если это вообще возможно.

Все функции имеют неопределённое поведение, когда `buf_t *` аргумент равен `NULL`.

При компиляции с `#define DEBUG` все неопределённые поведения должны вызывать завершение программы при помощи `abort`.

## Файлы в репозитории

- `/lib/bufio.h`
- `/lib/bufio.c`
- `/lib/Makefile`

## Скриптом сборки генерируется

- `/lib/libbufio.so`

## Hints

- `man 2 read`
- `man 2 write`

## Вторая часть

Используя `libbufio` из первой части, реализовать утилиту `cat`.

Даже если при чтении из `stdin` происходит ошибка, все уже прочитанные данные должны быть записаны в `stdout`, если это вообще возможно.

Если программа не помещается в 20 строк, то вы делаете что-то серьёзно не так.

## Файлы в репозитории

- `/bufcat/bufcat.c`
- `/bufcat/Makefile`

## Скриптом сборки генерируется

- `/bufcat/bufcat`

## Дедлайн

- ~~1 апреля, 06:00 (GMT+3)~~ (и это не шутка)
- Какой-то шутник уронил рейн. 3 апреля, 06:00 (GMT+3)

# Задание пятое

Требуется реализовать механизм, аналогичный | в терминале: запустить первую программу, передать её стандартный вывод на вход второй, стандартный вывод второй – на стандартный ввод третьей, ... стандартный вывод n-1-й на стандартный ввод n-й, стандартный вывод n-й оставить без изменений.

## Первая часть

Добавить в библиотеку (libhelpers) следующее:

- struct `execargs_t {...}` – информация о том, какую программу с какими аргументами нужно запустить.
- Какой-нибудь, на ваш выбор, способ конструирования `execargs_t`.
- int `exec(execargs_t* args)` – запустить указанную в `args` программу. Вернуть отрицательное число, если что-то пошло не так. Запуск производить аналогично запуску `spawn` из третьего задания: с учётом переменной окружения `PATH`.
- int `runpiped(execargs_t** programs, size_t n)` — запустить `n` программ, указанные в первых `n` элементах массива `programs`, связав стандартный вывод `programs[0]` со стандартным входом `programs[1]`, ..., стандартный вывод `programs[n-2]` со стандартным входом `programs[n-1]` и дождаться *завершения работы* этой цепочки. Соединение стандартного вывода одной программы со стандартным входом другой осуществляется с помощью пайпа. Завершение работы может произойти по следующим причинам:
  - прилетел `SIGINT` (например, пользователь нажал `Ctrl+C`)
  - один из запущенных процессов завершил работу
  - один из запущенных процессов закрыл пайп

Если что-то из этого произошло, нужно завершать всю цепочку процессов и вернуть 0. В случае ошибки, вернуть -1.

## Файлы в репозитории

- `/lib/libhelpers.h`
- `/lib/libhelpers.c`
- `/lib/Makefile`

## Скриптом сборки генерируется

- `/lib/libhelpers.so`

## Hints

- `man 2 pipe`
- `man 3 dup`, `man 3 dup2`
- `man 3 execlp`
- `man 3 sigaction`, `man 2 kill`, `man 7 signal`

## Вторая часть

Используя `libbufio` из первой части, реализовать утилиту `simplesh`.

Утилита должна вести себя следующим образом:

1. Выводит приглашение командной строки `$`.
2. Считывает строку со стандартного ввода.
3. Разбивает её по символу `|`. Получатся программы с аргументами, которые нужно будет запустить.
4. Полученные после разбивки по `|` строки разбивает по пробелу (или нескольким). Получается программа, которую нужно запустить, и аргументы к ней (как следствие, нельзя дать программе аргумент, содержащий пробел или `|`).
5. Используя `runpiped` запускает эту цепочку процессов.
6. Ждёт завершения, повторяет.

Ожидается, что пользователь будет работать с этой программой также, как с привычным эмулятором терминала. В

частности, нажатие Ctrl+C убивает запущенную цепочку процессов, но не убивает сам simplesh. Ctrl+D вместо задания строки должен приводить к выходу из simplesh.

## Файлы в репозитории

- /simplesh/simplesh.c
- /simplesh/Makefile

## Скриптом сборки генерируется

- /simplesh/simplesh

## Пример работы

```
$ls
<показано содержимое директории>
$find /home | grep \.c | head
<показаны какие-то 10 файлов, заканчивающихся на .c>
$find /home | head | cat
<показаны какие-то 10 файлов>
$cat /dev/urandom | grep 12345678
<работает. после нажатия Ctrl+C останавливается>
$<Ctrl+D приводит к завершению работы>
```

## Дедлайн

- 29 мая, 06:00 (GMT+3)

# Задание шестое

Требуется поработать с различными вариантами обработки множества запросов.

## Часть первая

Требуется написать TCP-сервер, который отправляет всем своим клиентам заданный файл, fork(2)аясь на каждый запрос.

## Формулировка

- Программа аргументами принимает:
  - порт, на котором будет слушать сервер;
  - название файла, который будет отправляться.
- Сервер создаёт сокет из указанного порта используя getaddrinfo(3) для localhost.
- Сервер ждёт подключений на этом порту.
- Как только кто-то ассепт(2)нулся, сервер fork(2)ется, родитель продолжает ассепт(2)ать новых клиентов, а ребёнок отправляет содержимое заданного файла клиенту.

- 
- Следует использовать ваш bufio.
  - Несколько клиентов могут получать файл независимо друг от друга.

## Файлы в репозитории

- /filesender/filesender.c
- /filesender/Makefile

## Скриптом сборки генерируется

- /filesender/filesender

## Hints

- man 2 fork

- ## Ещё больше подсказок

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <stdio.h>

int main()
{
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == -1)
        perror("socket");
    printf("sock = %d\n", sock);
    int one = 1;
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(int)) == -1)
        perror("setsockopt");
    struct sockaddr_in addr = {
        .sin_family = AF_INET,
        .sin_port = htons(1234),
        .sin_addr = {s_addr = INADDR_ANY}};
    if (bind(sock, (struct sockaddr*)&addr, sizeof(addr)) == -1)
        perror("bind");
    if (listen(sock, 1) == -1)
        perror("listen");
    struct sockaddr_in client;
    socklen_t sz = sizeof(client);
    int fd = accept(sock, (struct sockaddr*)&client, &sz);
    if (fd == -1)
        perror("accept");
    printf("accept = %d\n", fd);
    printf("from %s:%d\n", inet_ntoa(client.sin_addr), ntohs(client.sin_port));
    char hello[] = "hello\n";
    if (write(fd, hello, sizeof(hello)) == -1)
        perror("write");
    char reply[100];
    ssize_t r = read(fd, reply, sizeof(reply));
    if (r == -1)
        perror("read");
    reply[r] = 0;
    printf("reply: %s\n", reply);
    return 0;
}
```

```

+-----+-----+
| $ cat hello.txt | | | |
| Hello, world! | | |
| $ ./filesender 1234 hello.txt | | |
| | $ nc localhost 1234 | | |
| | Hello, world! | | |
| | $ socat STDIO TCP:localhost:1234 | | |
| | Hello, world! | | |
+-----+-----+

```

```

$ dd if=/dev/zero of=zeros.bin count=1000000
1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB) copied, 1.23916 s, 413 MB/s
$ md5sum zeros.bin

```

```

7583b57452ccb83f9d313ab76e3629b8 zeros.bin
$ ./filesender 1235 zeros.bin
$ nc localhost 1235 | (sleep 100; cat) | md5sum
7583b57452ccb83f9d313ab76e3629b8
# через полторы минуты $ nc localhost 1235 | md5sum
7583b57452ccb83f9d313ab76e3629b8
# быстрее

```

## Часть вторая

Аналогично первой части, но теперь вместо файла серверу указывается второй порт на котором надо слушать, а данные передаются между парами клиентов, подключённых к двум различным портам, в обе стороны.

### Формулировка

- Программа принимает два аргумента с номерами портов.
- Сервер ждёт подключения на первом порту.
- Как только кто-то ассерт(2)нулся, сервер начинает ждать подключения на втором порту.
- Как только кто-то ассерт(2)нулся на втором порту, сервер fork(2)ется дважды, родитель продолжает ассерт(2)ать, а дети начинают перенаправлять данные между двумя сокетами: один — в одну сторону, другой — в другую).

- 
- Следует использовать ваш bufio.
  - Несколько пар клиентов могут работать независимо друг от друга.

### Файлы в репозитории

- /bipiper/forking.c
- /bipiper/Makefile

### Скриптом сборки генерируется

- /bipiper/forking

### Hints

- Аналогично первой части.

### Пример работы

```

$ ./forking 1234 4321
$ nc localhost 1234 | $ socat STDIO TCP:localhost:1234
< Hi!
> Hi!
< Hi. How are you?
> Hi. How are you?

```

## Дедлайн первых двух частей

- 3 июня, 06:00 (GMT+3)

## Часть третья

Аналогично второй части, но без fork, а с poll.

Снаружи сервер выглядит абсолютно эквивалентно серверу из второй части, но внутри он никогда не fork(2)ается, а всё обрабатывается в одном цикле через один единственный poll.

Дальше тут строго предполагается, что вы действительно слушали лекцию и/или прочли man 2 poll. Если ещё нет, то сейчас самое время. Вот прямо сейчас. Я предупредил.



### Формулировка

- Программа принимает два аргумента с номерами портов.
- Сервер делает poll(2).
- После появления соединений на оба порта, сервер выделяет пару буферов и добавляет полученные два клиентских файловых дескриптора в массив для poll.
- Обработывает пары уже существующих клиентов.

- Следует использовать ваш bufio.
- До 127 пар клиентов могут работать независимо друг от друга. Если их становится 127, то сервер просто больше не делает ассептов до освобождения ресурсов.
- При ошибке или лаге между какой-то парой клиентов остальные пары клиентов не должны страдать.

Каждая пара клиентов для сервера выглядит так:

```
fd1 => | buffer1 | => fd2
fd2 => | buffer2 | => fd1
```

и этими `bufferami` между этими `fd` вы в этом задании делаете аналог `pipe(2)` ("с его обратной стороны") опрашивая `стейты` этих `fd` через `poll(2)`.

- Заметьте, что если, например, `buffer1` пуст, то про `fd2` нельзя спрашивать у `poll` разрешения на `POLLOUT` (а не то будете жрать 100% CPU), что в ядре было бы эквивалентно приостановке процесса `fd2`.
- Аналогично, если `buffer1` полон, то про `fd1` нельзя спрашивать у `poll` разрешения на `POLLIN`, что в ядре было бы эквивалентно приостановке процесса `fd1`.

Внутри у вашего сервера должно быть множество структур вида "пара файловых дескрипторов и пара буферов" для двунаправленных пайпов для максимум 127 клиентов, которые он poll(2)ит по мере надобности, используя один фиксированный массив pollfd[256] и один массив пар буферов buffs[127].

Первые два файловых дескриптора в первом из массивов — ассерт сокеты, остальные — пары дескрипторов из двунаправленных пайпов. Во втором массиве только пары буферов.

Картинкой:

```

+-----+-----+
| pollfd[256] | buffs[127] |
+-----+-----+
| acceptfd1 | | |
| | | |
| acceptfd2 | | |
| | | |
| fd1 | { buffer1 | x x |
| | | x 127 |
| fd2 | buffer2 } x x |
+-----+-----+

```

У сервера два состояния:

- делаем accept(2) на первом порту,
- делаем accept(2) на втором порту,

в каждом из которых он спрашивает POLLIN только у соответствующего нужному порту accept сокету и одновременно опрашивает нужных ему для работы живых клиентов.

Когда какой-то из пайпов завершается, нужно свопать эти элементы массивов с последними занятыми элементами, чтобы не спрашивать poll больше, чем надо.

Картинкой для pollfd:

| acceptfd1 | acceptfd2 | fd1<sub>1</sub> | fd2<sub>1</sub> | fd1<sub>2</sub> | fd2<sub>2</sub> | ... заняты ... | fd1<sub>n</sub> | fd2<sub>n</sub> | fd1\_empty | fd2\_empty | ... свободные

$\underbrace{\hspace{15em}}_{2n \text{ штук}}$

Вторая пара закрылась:

| acceptfd1 | acceptfd2 | fd1<sub>1</sub> | fd2<sub>1</sub> | fd1<sub>n</sub> | fd2<sub>n</sub> | ... занятые ... | fd1<sub>2</sub> | fd2<sub>2</sub> | fd1\_empty | fd2\_empty | ... свободные

И аналогично для `bufs`.

Между клиентами нужно делать прямо настоящие пайпы, для этого вам иногда придётся закрывать файловые дескрипторы только в одну сторону, что делается `shutdown(2)`.

## Файлы в репозитории

- `/bipiper/polling.c`
- `/bipiper/Makefile`

## Скриптом сборки генерируется

- `/bipiper/polling`

## Hints

- Аналогично второй части.
- `man 2 poll` обратите внимание на `ERRORS`. `poll` может прерваться с `EINTR` и это не ошибка.
- `man 2 shutdown`

## Пример работы

- Аналогично второй части.

## Дедлайн для третьей части

- 5 июня, 06:00 (GMT+3)