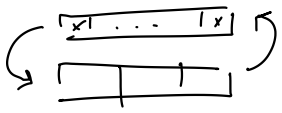
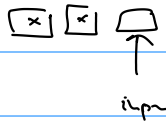
1. (10 poena) Ulaz/izlaz

Realizovati u potpunosti klasu `DoubleBuffer` čiji je interfejs dat. Ova klasa implementira dvostruki bafer. Proizvođač stavlja u bafer znak po znak pozivom operacije `put()`; znak se stavlja u trenutni „izlazni" bafer od dva interna bafera veličine `size` znakova. Potrošač uzima blokove veličine `chunkSize` znakova iz trenutnog „ulaznog" bafera pozivom operacije `get()`; znakovi se prepisuju u bafer pozivaoca na koji ukazuje argument `buffer`. Kada obojica završe sa svojim baferom, baferi zamenjuju uloge. Proizvođač i potrošač su uporedne niti (ne treba ih realizovati), dok je sva potrebna sinhronizacija unutar klase `DoubleBuffer`. Pretpostaviti da je zadata veličina bafera u znakovima (argument `size` konstruktora) celobrojan umnožak zadate veličine bloka (argument `chunkSize`). Za sinhronizaciju koristiti semafore.

```
class DoubleBuffer {
public:
  DoubleBuffer (int size, int chunkSize);
  void put (char);
  void get (char* buffer);
private:
  ...
};
```



```
class DoubleBuffer {
    char *buffers[2]
    int size, chunkSize;
    DoubleBuffer(int size, int chunkSize)
    : size(size), chunkSize(chunkSize) {
        buffers[0] = new char[size];
        buffers[1] = new char[size];
    }
    ~DoubleBuffer() {
        delete[] buffers[0];
        delete[] buffers[1];
    }
    int input = 0, input_idx = 0;
    int output = 1, output_idx = size;
    Semaphore input_full(0);
    Semaphor output_empty(1);
    void put(char c) {
        if (input_idx == size) {
            output_empty.wait();
            input = 1 - input;
            input_idx = 0
        }
```



swap_buffers();

```
        buffers[input][input_idx] = c;
        input_idx += 1;
        if (input_idx == size) {
            input_full.signal();
        }
    }

    void get(char *buffer) {
        if (output_idx == size) {
            input_full.wait();
            output = 1 - output;
            output_idx = 0;
        }

        for (int i=0; i<chunk_size; i++) {
            buffer[i] = buffers[output][output_idx];
            output_idx += 1;
        }

        if (output_idx == size) {
            output_empty.signal();
        }
    }

    void swap_buffers() {
        input_full.wait();
        output_empty.wait();
        input = 1 - input;
        output = 1 - output;
        input_idx = 0
        output_idx = 0;
    }
```

swap_buffers()

next

1. (10 poena) Ulaz/izlaz

Neki sistem organizuje keš blokova sa diska na sledeći način. Keš čuva najviše `CACHESIZE` blokova veličine `BLKSIZE` u nizu `diskCache`. Svaki ulaz *i* u nizu `diskCacheMap` sadrži broj bloka na disku koji se nalazi u ulazu *i* keša. Vrednost 0 u nekom ulazu niza `diskCacheMap` označava da je taj ulaz prazan (u njega nije učitan blok). Data je sledeća implementacija keša:

```
typedef ... Byte; // Unit of memory
typedef ... BlkNo; // Disk block number
const int BLKSIZE = ...; // Disk block size in Bytes
const int CACHESIZE = ...; // Disk cache size in blocks

BYTE diskCache [CACHESIZE][BLKSIZE]; // Disk cache
BlkNo diskCacheMap [CACHESIZE]; // The contents of disk cache. 0 for empty

int diskCacheCursor = 0; // FIFO cursor for eviction/loading

Byte* getDiskBlock (BlkNo blk) {
  // Search for the requested block in the cache and return it if found:
  for (int i=0; i<CACHESIZE; i++) {
    if (diskCacheMap[i]==blk) return diskCache[i];
    if (diskCacheMap[i]==0) break;
  }
  // Not found.
  // If there is a block to evict, write it to the disk:
  if (diskCacheMap[diskCacheCursor]!=0)
    diskWrite(diskCacheMap[diskCacheCursor],diskCache[diskCacheCursor]);
  // Load the requested block:
  diskCacheMap[diskCacheCursor] = blk;
  diskRead(blk,diskCache[diskCacheCursor]);
  Byte* ret = diskCache[diskCacheCursor];
  diskCacheCursor = (diskCacheCursor+1)%CACHESIZE;
  return ret;
}
```

Operacije `diskRead` i `diskWrite` vrše sinhrono čitanje, odnosno upis datog bloka sa diska:

```
void diskRead(BlkNo block, Byte* toBuffer);
void diskWrite(BlkNo block, Byte* fromBuffer);
```

Ukoliko je keš pun, a treba učitati novi blok, iz keša se izbacuje blok iz ulaza na koji ukazuje kurzor `diskCacheCursor` koji se pomera u krug, tako da je izbacivanje (engl. *eviction*) po FIFO (*first-in-first-out*) principu.

Funkcija `getDiskBlock` vraća pokazivač na deo memorije u kome se nalazi učitan traženi blok diska, pri čemu se taj blok učitava u keš (uz prethodno snimanje eventualno izbačenog bloka) ukoliko taj blok već nije u kešu. Ovu funkciju koristi ostatak sistema za pristup blokovima diska.

Ova implementacija keša je nedovoljno efikasna jer se neki blok sa diska može smestiti u bilo koji ulaz keša i zato pronalaženje bloka u punom kešu često zahteva obilaženje velikog dela niza `diskCacheMap`. Zbog toga treba promeniti implementaciju ovog keša tako da se `diskCacheMap` organizuje kao heš (*hash*) tabela. Disk blok broj *b* se smešta u ulaz *hash*(*b*)=*b* mod CACHESIZE, ukoliko je taj ulaz slobodan. U slučaju da nije (tj. u slučaju kolizije), taj blok se smešta u prvi naredni slobodan blok (u krug, po modulu `CACHESIZE`; rešavanje kolizije otvorenim adresiranjem).

Dati ovako izmenjenu funkciju `getDiskBlock`.

```
typedef ... Byte; // Unit of memory
typedef ... BlkNo; // Disk block number
const int BLKSIZE = ...; // Disk block size in Bytes
const int CACHESIZE = ...; // Disk cache size in blocks

BYTE diskCache [CACHESIZE][BLKSIZE]; // Disk cache
BlkNo diskCacheMap [CACHESIZE]; // The contents of disk cache. 0 for empty

int diskCacheCursor = 0; // FIFO cursor for eviction/loading

Byte* getDiskBlock (BlkNo blk) {
  // Search for the requested block in the cache and return it if found:
  for (int i=0; i<CACHESIZE; i++) {
    if (diskCacheMap[i]==blk) return diskCache[i];
    if (diskCacheMap[i]==0) break;
  }
  // Not found.
  // If there is a block to evict, write it to the disk:
  if (diskCacheMap[diskCacheCursor]!=0)
    diskWrite(diskCacheMap[diskCacheCursor],diskCache[diskCacheCursor]);
  // Load the requested block:
  diskCacheMap[diskCacheCursor] = blk;
  diskRead(blk,diskCache[diskCacheCursor]);
  Byte* ret = diskCache[diskCacheCursor];
  diskCacheCursor = (diskCacheCursor+1)%CACHESIZE;
  return ret;
}
```

```
Byte * getDiskBlock ( BlkNo blk ) {
    int idx = blk % CASHSIZE;
    for (int i = 0; i < CASHSIZE; i++) {
        if ( diskCashMap[idx] == blk ) {
            return diskCash[idx];
        } else if ( diskCashMap[idx] == 0 ) {
            break;
        }
        idx = (idx+1) % CASHSIZE;
    }

    if ( diskCashMap[idx] == 0 ) {
        diskCashMap[idx] = blk;
        diskRead ( blk, diskCash[idx] );
        return diskCash[idx];
    } else {
        diskWrite ( diskCashMap[idx], diskCash[idx] );
        diskCashMap[idx] = blk;
        diskRead ( blk, diskCash[idx] );
        return diskCash[idx];
    }
}
```

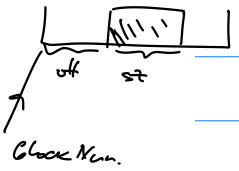
1. (10 poena) Ulaz/izlaz

U nekom sistemu implementira se keš blokova za ulazne, blokovski orijentisane uređaje sa direktnim pristupom; pošto su uređaji ulazni, blokovi se mogu samo čitati. Za svaki takav uređaj sa datim identifikatorom pravi se jedan objekat klase `BlockIOCache`, inicijalizovan tim identifikatorom, koji predstavlja keš blokova sa tog uređaja. Keš čuva najviše `CACHESIZE` blokova veličine `BLKSIZE` u nizu `cache`. Svaki ulaz *i* u nizu `cacheMap` sadrži broj bloka na uređaju koji se nalazi u ulazu *i* keša. Kada učitava blokove, keš najpre redom popunjava svoje ulaze, dok ima neiskorišćenih ulaza; podatak član `numOfBlocks` govori o tome koliko je ulaza zauzeto (redom, prvih, od ulaza broj 0). Kada popuni sve ulaze, traženi blok koji nije u kešu učitava se na mesto bloka koji je najdavnije učitan (zamenjuje se taj blok učitanim blokom).

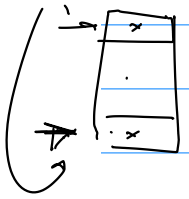
Na raspolaganju je operacija `ioRead` koja sa datog uređaja učitava blok sa zadatim brojem u bafer zadat poslednjim argumentom. Implementirati funkciju `BlockIOCache::read` koja treba da iz bloka za datim brojem `blk` niz bajtova počev od pozicije `offset` i dužine `sz` prepiše u bafer `buffer` koji je alocirao pozivalac. Pretpostaviti da su ovi argumenti ispravni i konzistentni (pozivalac je proverio njihovu ispravnost) i da se eventualne greške u operaciji `ioRead` obrađuju u njoj ili negde drugde (ignorisati ih). (Definicija klase `BlockIOCache` nije kompletna i može se dopunjavati po potrebi.)

```
typedef char byte; // Unit of memory
typedef long long BlkNo; // Device block number
void ioRead (int device, BlkNo blk, byte* buffer);

class BlockIOCache {
public:
  BlockIOCache (int device) : dev(device), numOfBlocks(0) {}
  void read (BlkNo blk, byte* buffer, size_t offset, size_t sz);

private:
  static const unsigned BLKSIZE = ...; // Block size in Bytes
  static const unsigned CACHESIZE = ...; // Cache size in blocks

  int dev;
  byte cache [CACHESIZE][BLKSIZE]; // Cache
  BlkNo cacheMap [CACHESIZE]; // Contents of the cache (block numbers)
  int numOfBlocks; // Number of used entries (blocks in the cache)
};
```

int cursor = 0;



cash_cursor = 0



Rešenje:

```
void read ( BlkNo blk, byte *buffer, size_t offset, size_t sz ) {
    int blk_idx = get_blk_idx ( blk );
    for ( int i=0; i < sz; i++ ) {
        buffer[i] = cash[ blk_idx ][ offset + i ];
    }
}

int get_blk_idx ( BlkNo blk ) {
    for ( int i=0; i < CASHSIZE; i++ ) {
        if ( cashMap[i] == blk ) return i;
        if ( cashMap[i] == 0 ) break;
    }
    ioRead ( dev, blk, cash[cursor] );
    int ret = cursor;
    cashMap[cursor] = blk;
```

```
        cursor = (cursor + 1) % CASH_SIZE;
        return ret;
    }
}
```

2. (10 poena) Interfejs fajl sistema

U nekom fajl sistemu postoje sledeći sistemski pozivi za osnovne operacije sa fajlom:

```
int open  (const char *pathname, int flags, mode_t mode);
int close (int fhandle);
int read  (int fhandle, byte* buffer, unsigned long size);
int write (int fhandle, byte* buffer, unsigned long size);
```

Sve ove operacije u slučaju greške vraćaju negativnu vrednost sa kodom greške. U slučaju uspeha, operacija otvaranja fajla vraća „ručku“ fajla (engl. *file handle*), a ostale vraćaju 0.

Realizovati objektno orijentisani „omotač“ ovog interfejsa, odnosno implementirati apstrakciju fajla kao klasu sa sledećim interfejsom:

```
class File {
public:
  File (const char *pathname, int flags, mode_t mode) throw Exception;
 ~File () throw Exception;

  void read  (byte* buffer, unsigned long size) throw Exception;
  void write (byte* buffer, unsigned long size) throw Exception;
};
```

Prilikom kreiranja objekta ove klase treba implicitno otvoriti fajl, a prilikom uništavanja objekta treba implicitno zatvoriti fajl. U slučaju greške, sve ove operacije treba da podignu izuzetak definisanog tipa `Exception`. Instance ovog tipa (klase) mogu se inicijalizovati celobrojnim kodom greške koju vraćaju sistemski pozivi.

Rešenje:

```
class File {
    private:
        int handle;
    public:
        File (const char *pathname, int flags, mode_t mode) {
            handle = open (pathname, flags, mode);
            if (handle < 0) {
                throw Exception (handle);
            }
        }
        ~File () {
            if (handle > 0) {
                int ret = close (handle);
                if (ret < 0) {
                    throw Exception (ret);
                }
            }
        }
```

```
void read (byte * buffer, unsigned long size) {
    int ret = read (handle, buffer, size);
    if (ret < 0) {
        throw Exception (ret);
    }
}.

void write (byte * buffer, unsigned long size) {
    int ret = write (handle, buffer, size);
    if (ret < 0) {
        throw Exception (ret);
    }
}
}.
```

2. (10 poena) Interfejs fajl sistema

U nekom fajl sistemu postoje, između ostalih, i sledeći sistemski pozivi za osnovne operacije sa fajlom:

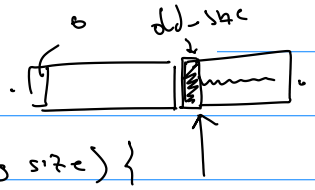
```
int fgetsize  (int fhandle, unsigned long& size);
int fresize   (int fhandle, unsigned long newsize);
int fmoveto   (int fhandle, unsigned long offset);
int fwrite    (int fhandle, byte* buffer, unsigned long size);
```

Operacija `fgetsize` u izlazni argument `size` upisuje trenutnu veličinu sadržaja fajla, a operacija `fresize` menja veličinu sadržaja fajla na novu zadatu veličinu, koja može biti i veća i manja od trenutne; ukoliko je manja, sadržaj na kraju se odseca, a ukoliko je veća, sadržaj koji se dodaje je nedefinisan. Operacija `fmoveto` pomera kurzor na zadatu poziciju bajta (numeracija bajtova sadržaja fajla počinje od 0). Sve ove operacije u slučaju greške vraćaju negativnu vrednost sa kodom greške.

Realizovati operaciju koja proširuje sadržaj fajla datim sadržajem (dodaje ga na kraj):

```
int append (int fhandle, byte* buffer, unsigned long size);
```

Rešenje:




```
int append (int fhandle, byte * buffer, unsigned long size) {
    unsigned long old_size;
    int ret = fgetsize (fhandle, old_size);
    if (ret < 0) return ret;
    ret = fresize (fhandle, old_size + size);
    if (ret < 0) return ret;
    ret = fmoveto (fhandle, old_size)
    if (ret < 0) return ret;
    ret = fwrite (fhandle, buffer, size);
    if (ret < 0) return ret;
    return 0;
}
```

2. (10 poena) Fajl sistem

Dole je dat izvod iz dokumentacije za API za fajlove u GNU sistemima. Date deklaracije su u `<unistd.h>` i `<fctl.h>`. Korišćenjem samo dole datih funkcija, realizovati sledeću funkciju koja prepisuje ceo sadržaj datog ulaznog fajla proizvoljne veličine u dati izlazni fajl, korišćenjem svog bafera određene veličine. U slučaju bilo kakve greške, funkcija treba da vrati negativnu vrednost, u suprotnom treba da vrati 0.

```
int fcopy (const char *filenamefrom, const char *filenameto);
```

`int open (const char *filename, int flags)`
Creates and returns a new file descriptor for the file named by `filename`. Initially, the file position indicator for the file is at the beginning of the file. The `flags` argument controls how the file is to be opened. This is a bit mask; you create the value by the bitwise OR of the appropriate parameters (using the '|' operator in C) – the following macros:

`O_RDONLY`	*Open the file for read access.*
`O_WRONLY`	*Open the file for write access.*
`O_RDWR`	*Open the file for both reading and writing.*
`O_CREAT`	*If set, the file will be created if it doesn't already exist.*
`O_APPEND`	*The bit that enables append mode for the file. If set, then all write operations write the data at the end of the file, extending it, regardless of the current file position.*
`O_TRUNC`	*Truncate the file to zero length.*

The normal return value from `open` is a non-negative integer file descriptor. In the case of an error, a value of -1 is returned instead.

`int close (int filedes);`
Closes the file descriptor `filedes`. The normal return value is 0. In the case of an error, a value of -1 is returned.

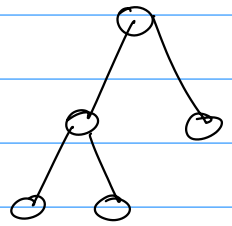
`ssize_t`
This data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to `size_t`, but must be a signed type.

`ssize_t read (int filedes, void *buffer, size_t size);`
Reads up to `size` bytes from the file with descriptor `filedes`, storing the results in the `buffer`. (This is not necessarily a character string, and no terminating null character is added.)
The return value is the number of bytes actually read. This might be less than `size`; for example, if there aren't that many bytes left in the file or if there aren't that many bytes immediately available. The exact behavior depends on what kind of file it is. Note that reading less than size bytes is not an error.
A value of zero indicates end-of-file (except if the value of the `size` argument is also zero). This is not considered an error. If you keep calling read while at end-of-file, it will keep returning zero and doing nothing else.
If `read` returns at least one character, there is no way you can tell whether end-of-file was reached. But if you did reach the end, the next read will return zero. In case of an error, `read` returns -1.

`ssize_t write (int filedes, const void *buffer, size_t size);`
Writes up to `size` bytes from `buffer` to the file with descriptor `filedes`. The data in `buffer` is not necessarily a character string and a null character is output like any other character.
The return value is the number of bytes actually written. This may be `size`, but can always be smaller. Your program should always call write in a loop, iterating until all the data is written. In the case of an error, `write` returns -1.

```
#define BSIZE 4096

int fcopy ( const char * src , const char * dest ) {
    int src_fd = open( src , O_RDONLY ) ;
    if ( src_fd < 0 ) return src_fd ;
    int dest_fd = open ( dest , O_WRONLY | O_CREAT | O_TRUNC ) ;
    if ( dest_fd ) return dest_fd ;
    char buffer [BSIZE] ;
    while (1) {
        ssize_t num_read = read ( src_fd , buffer , BSIZE ) ;
        if ( num_read < 0 ) return num_read ;
        if ( num_read == 0 ) break ;
        char * write_buf = buffer ;
        size_t to_write = num_read ;
```

```
        while ( to_write ≠ 0 ) {
            int written = write ( dest_fd, write_buf, to_write);
            if (written < 0) return written;
            to_write -= written;
            write_buf += written;
        }
    }

    return 0;
}
```



```
int len = 1 << n - 1;
```

≥ len → return 0;

if (x > v) go right

if (x < v) go left.

if (x == v) return 1;

2. (10 poena) Fajl sistem

U nekom binarnom fajlu zapisano je ogromno binarno stablo; stablo je potpuno balansirano, ima n nivoa i tačno 2^n-1 čvorova, tj. svaki čvor osim lista ima tačno dva deteta. U svakom čvoru nalazi se jedna jedinstvena vrednost tipa `int`, a stablo je sortirano (levi potomci su manji, a desni veći od svakog čvora). Stablo je zapisano u sadržaju fajla kao niz, tako da svaki element niza sadrži samo vrednost u odgovarajućem čvoru, dok su deca čvora koji odgovara elementu i implicitno određena i nalaze se u elementima $2i+1$ i $2i+2$ tog niza; koren je u elementu 0.

Korišćenjem dole datih sistemskih poziva standardnog C fajl interfejsa, implementirati funkciju

```
int binary_search (const char* filename, unsigned n, int x);
```

koja u binarnom fajlu sa datim imenom, u kome je zapisano stablo sa n nivoa u opisanom formatu, binarnom pretragom traži vrednost datu x i vraća 1 ako je pronađe, odnosno 0 ako je ne pronađe. Ignorisati sve potencijalne greške u sistemskim pozivima. Sledeće funkcije deklarisane su u zaglavlju `cstdio`:

- `std::FILE* std::fopen(const char* filename, const char* mode);`
  otvara fajl sa zadatim imenom u zadatom modalitetu; za čitanje je modalitet „r“, za upis „w“, a za otvaranje fajla u binarnom režimu treba dodati sufiks „b“ na modalitet;
- `int std::fclose(FILE*);`
  zatvara dati fajl;
- `std::size_t fread(void* buffer, std::size_t size, std::size_t count, std::FILE* stream);`
  iz datog fajla, počev od tekuće pozicije kurzora, u zadati bafer učitava najviše `count` objekata, svaki veličine `size` i pomera kurzor fajla iza pročitanog sadržaja; vraća broj stvarno pročitanih objekata (manje od traženog u slučaju greške ili nailaska na kraj fajla);
- `int fseek(std::FILE* stream, long offset, int origin);`
  pomera kurzor datog fajla na pomeraj (poziciju) zadatu parametrom `offset` (u bajtovima, odnosno jedinicama koje vraća operator `sizeof`), u odnosu na položaj zadat parametrom `origin`; za pomeraj u odnosu na početak fajla, ovaj parametar treba da bude jednak konstanti `SEEK_SET`.

Rešenje:

```
int binary_search (const char *filename, unsigned n, int x) {
    FILE *f = fopen(filename, "rb");
    int node_idx = 0;
    int len = 1 << n - 1;
    while (node_idx < len) {
        int val;
        fseek(f, node_idx * sizeof(int), SEEK_SET);
        fread(&val, sizeof(int), 1, f);
        if (val == x) {
            fclose(f);
            return 1;
        }
        if (val > x) node_idx = 2 * node_idx + 1;
        else node_idx = 2 * node_idx + 2;
    }
```

fclose(f);

return 0;

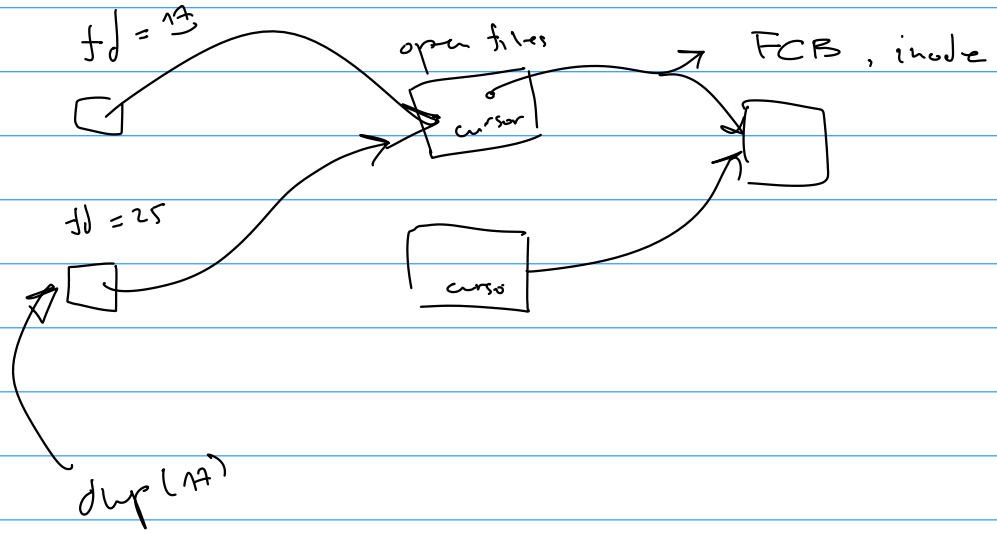
}.

1. (10 poena) Ulaz/izlaz

Korišćenjem biblitečnih funkcija *popen*, *dup2* i *fileno* implementirati funkciju *redirect* čiji je potpis dat i koja pokreće proces dete nad programom u fajlu sa putanjom zadatom argumentom `exe`, a onda preusmerava standardni izlaz pozivajućeg (svog) procesa na standardni ulaz tog procesa deteta; u slučaju neuspeha vraća -1, u slučaju uspeha vraća 0. Biblitečna funkcija *fileno* vraća celobrojni deskriptor fajla (*fd*) koji odgovara zadatom znakovnom toku `stream`.

```
int fileno (FILE* stream);
int redirect (const char* exe);
```

Rešenje:




```
int redirect (const char *exe) {
    FILE *f = popen (exe, "w");
    int fd = fileno (f);
    dup2 (fd, STDOUT_FILENO);
}

:
printf ("hello\n");
```