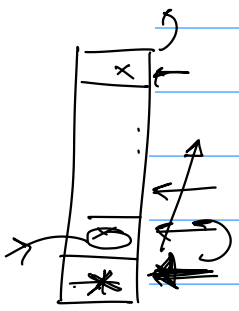
1. (10 poena)

U nekom sistemu implementira se keš blokova sa blokovskih uređaja kodom koji je dat u nastavku. Za svaki uređaj sa datim identifikatorom pravi se jedan objekat klase `BlockIOCache`, inicijalizovan tim identifikatorom, koji predstavlja keš blokova sa tog uređaja. Keš je kapaciteta `CACHESIZE` blokova veličine `BLKSIZE`. Keš je interno organizovan kao heš mapa `map` sa `MAPSIZE` ulaza. Svaki ulaz niza `map` sadrži glavu liste keširanih blokova koji se preslikavaju u taj ulaz. Funkcija `hash` je heš funkcija koja preslikava broj bloka u ulaz u nizu `map`. Glava liste, kao i pokazivač na sledeći element u listi čuvaju se kao indeksi elementa niza `entries` koji sadrži keširane blokove; vrednost -1 označava kraj (*null*). Svaki element niza `entries` je struktura tipa `CacheEntry` u kojoj je polje `blkNo` broj bloka koji je keširan u tom elementu, polje `next` ukazuje na sledeći element liste, a polje `buf` je sadržaj samog bloka.

Na početku složene operacije sa uređajem, kod koji koristi keš najpre zahteva da potrebni blok bude učitan pozivom funkcije `getBlock` koja vraća pokazivač na niz bajtova u baferu – učitanom bloku. Pošto više ovakvih složenih operacija može biti pokrenuto uporedo, blok iz keša može biti izbačen (zamenjen drugim) samo ako ga više niko ne koristi, što se realizuje brojanjem referenci u polju `refCounter` strukture `CacheEntry`. Prikazana je implementacija funkcije `getBlock` koja treba da obezbedi da je traženi blok u kešu, odnosno učita ga ako nije.

Potrebno je implementirati pomoćnu funkciju `getFreeEntry` koja treba da vrati indeks slobodnog ulaza u nizu `entries` u koji se može učitati traženi blok u keš. Inicijalno je keš prazan i svi ulazi u njemu su slobodni. Ova funkcija treba redom da zauzima elemente niza `entries`, sve dok ima slobodnih. Kada slobodnih ulaza više nema, ona treba da izbaci blok (snimi ga na disk) u prvom ulazu koji je na redu po FIFO redosledu (najstariji učitan), ali samo pod uslovom da se blok u tom ulazu ne koristi (tj. njegov `refCounter` je nula). Ako to nije zadovoljeno, treba da proba sa sledećim i tako u krug. Ako nema mesta u kešu jer nijedan blok ne može da se izbaci, treba vratiti -1. Ukoliko je potrebno dodati ili izmeniti članove ove klase, precizno navesti kako to treba uraditi. Na raspolaganju je i funkcija koja učitava blok, odnosno upisuje blok na dati uređaj:

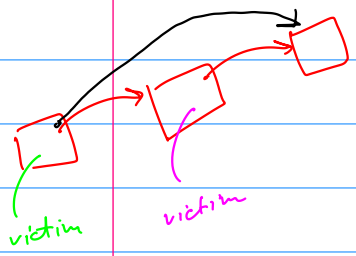
```
void ioRead (int device, BlkNo blk, Byte* buffer);
void ioWrite(int device, BlkNo blk, Byte* buffer);

typedef unsigned char Byte; // Unit of memory
typedef long BlkNo; // Device block number
const unsigned BLKSIZE = ...; // Block size in Bytes

class BlockIOCache {
public:
  BlockIOCache (int device); device(device) { init(); }.
  Byte* getBlock (BlkNo blk);
  ...
protected:
  static int hash (BlkNo);
  int getFreeBlock ();
private:
  static const unsigned CACHESIZE = ...; // Cache size in blocks
  static const unsigned MAPSIZE = ...; // Hash map size in entries

  struct CacheEntry { BlkNo blkNo; int next; int refCounter; Byte buf[BLKSIZE]; };

  int dev;
  int map[MAPSIZE];  // Hash map
  CacheEntry entries[CACHESIZE]; // Cache
  int cursor = 0;
```



```
Byte* BlockIOCache::getBlock (BlkNo blk) {
  // Find the requested block in the cache and return it if present:
  int entry = hash(blk);
  for (int i=map[entry]; i!=-1; i=entries[i].next)
    if (entries[i].blkNo==blk) {
      entries[i].refCounter++;
      return entries[i].buf;
    }
  // The block is not in the cache, find a free slot to load it:
  int free = getFreeEntry();
  if (free==-1) return 0;  // Error: cannot find space
  // Load the requested block:
  entries[free].blkNo = blk;
  entries[free].refCounter = 1;
  entries[free].next = map[entry];
  map[entry] = free;
  ioRead(dev,blk,entries[free].buf);
  return entries[free].buf;
}
```

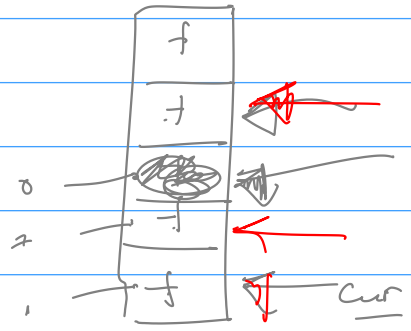
```
void BlockIOCache::init() {
    for (int i=0; i< CASHSIZE; i++) {
        entries[i].blkNo = -1;
    }
}
```

```
int BlockIOCache::getFreeEntry() {
    if (entries[cursor].blkNo == -1) {
        int free = cursor;
        cursor = (cursor+1) % CASHSIZE;
        return free;
    } else {
        int victim = -1;
        for (int i=0; i< CASHSIZE; i++) {
            int idx = (cursor+i) % CASHSIZE;
            if (entries[idx].refCounter == 0) {
                victim = idx;
                break;
            }
        }
```




```
        if (victim == -1)  return -1;
        int vlist = hash(entries[victim].blkNo);
        if (map[vlist] == victim) {
            map[vlist] = entries[victim].next;
        } else {
            int cur = map[vlist];
            while (entries[cur].next ≠ victim) {
                cur = entries[cur].next;
            }
```

```
        entries[cur].next = entries[victim].next;
    }

    cursor = (victim + 1) % CACHE_SIZE
    io_write(dev, entries[victim].blkNo, entries[victim].buf);
    entries[victim].blkNo = -1;
    entries[victim].next = -1;
    return victim;
}
```



2. (10 poena)

Dat je podsetnik na neke osnovne Unix komande:

- `cat`: iz svakog fajla koji je naveden kao argument ove komande, redom kojim su oni navedeni, učitava znakove i ispisuje ih na standardni izlaz; ukoliko nema argumenata, znakove učitava sa standardnog ulaza (dok ne naiđe na znak EOF koji se na konzoli signalizira pritiskom tastera Ctrl+D).
- `ln` *`src_file dst_file`*: pravi novu tvrdu vezu (ulaz u direktorijumu) za postojeći fajl *src_file* u odredištu definisanom sa *dst_file*; opcija -s radi isto to, samo što pravi meku (simboličku) vezu.
- `rm`: briše fajl ili direktorijum ili vezu zadat kao argument.

Izvršavaju se sledeće komande po datom redosledu; napisati te komande ili odgovoriti na postavljeno pitanje:

1. U postojećem poddirektorijumu `a` tekućeg direktorijuma napraviti nov tekstualni fajl `doc` čiji će sadržaj biti učitan sa konzole:

   cat > a/doc

2. U postojećem poddirektorijumu `b` tekućeg direktorijuma napraviti tvrdu vezu pod nazivom `hdoc` na fajl napravljen u koraku 1:

   ln a/doc b/hdoc

3. U postojećem poddirektorijumu `c` tekućeg direktorijuma napraviti meku vezu pod nazivom `sdoc` na fajl napravljen u koraku 2:

   ln -s b/hdoc c/sdoc

4. Obrisati `doc` u poddirektorijumu a:

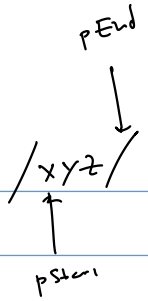
   rm a/doc

5. Šta će nakon ovoga ispisati komanda `cat b/hdoc`?

   Neće ništa ispisati doc. fajla.

6. Nakon izdate komande `rm b/hdoc`, šta će ispisati komanda `cat c/sdoc`?

   c/sdoc je dangling, pokaže na ništa.

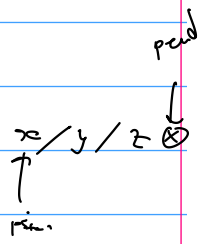


3. (10 poena)

U implementaciji nekog fajl sistema svaki čvor u hijerarhijskoj strukturi direktorijuma i fajlova predstavljen je objektom klase `Node`. Operacija te klase:

```
Node* Node::getSubnode(const char* pStart, const char* pEnd);
```

vraća podčvor datog čvora `this` koji ima simboličko ime zadato nizom znakova koji počinje znakom na koga ukazuje `pStart`, a završava znakom ispred znaka na koga ukazuje `pEnd` (`pEnd` može ukazivati na `'\0'` ili `'/'`). Ukoliko dati čvor `this` nije direktorijum, ili u njemu ne postoji podčvor sa datim simboličkim imenom, ova funkcija vraća 0. Koreni direktorijum cele hijerarhije dostupan je kao statički pokazivač `Node::root` tipa `Node*`. Znak za razdvajanje (`delimiter`) u stazama, kao i znak za koreni dorektorijum je kosa crta `'/'`.

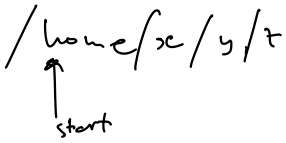


Data je implementacija funkcije `getNodeRel` koja za (sintaksno ispravnu) relativnu putanju datu nizom znakova koji počinje znakom na koga ukazuje `pStart` i završava se znakom ispred znaka na koji ukazuje `pEnd` vraća čvor definisan tom putanjom u odnosu na dati čvor.

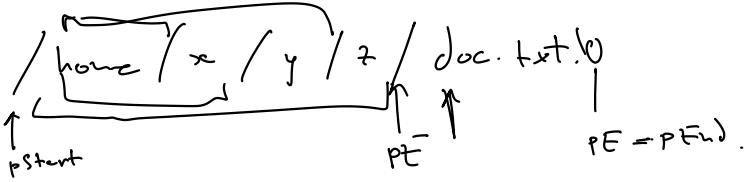
Korišćenjem ove funkcije, implementirati funkciju `Node::getNodeAbs` koja za apsolutnu putanju (garantovano je sintaksno ispravna i počinje znakom `'/'`) vraća čvor određen tom putanjom, uz korišćenje keša preslikavanja apsolutnih putanja u čvorove `DentryCache`. Data funkcija `DentryCache::getNode` vraća ranije zapamćen čvor za apsolutnu putanju datu nizom znakova zadatog sa `pStart` i `pEnd`. Ukoliko traženi čvor postoji, treba ažurirati keš unosom ulaza koji preslikava datu apsolutnu putanju u dati čvor pozivom funkcije `DentryCache::storeNode`.

```
static const char delimiter = '/';

Node* Node::getNodeRel (const char* pStart, const char* pEnd, Node* node) {
  while (node && pStart<pEnd) {
    const char* pE = pStart+1;
    while (pE<pEnd && *pE!=delimiter) pE++;
    node = node->getSubnode(pStart,pE);
    pStart = (pE<pEnd)?(pE+1):pE;
  };
  return node;
}

Node* DentryCache::getNode(const char* pStart, const char* pEnd);
void DentryCache::storeNode(const char* pStart, const char* pEnd, Node*);

Node* Node::getNodeAbs (const char* pStart, const char* pEnd){
```

root.




```
Node * node = DentryCache::getNode (pStart, pEnd);
if (node ≠ null)  return node;
node = Node::getNodeRel (pStart+1, pEnd, Node::root);
if (node == null)  return null;
DentryCache (pstart, pEnd, node);
return node;
}
```



3. (10 poena)

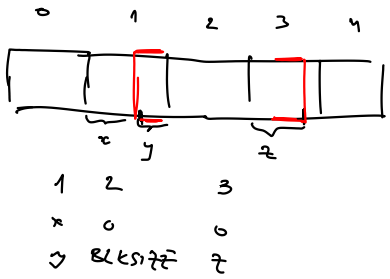
```
Node* Node::getNodeAbs (const char* pStart, const char* pEnd) {
  const char *pE = pEnd;
  while (pE>pStart) {
    Node* node = DentryCache::getNode(pStart,pE);
    if (node) {
      if (pE<pEnd) node = Node::getNodeRel(pE+1,pEnd,node);
      if (node) DentryCache::store(pStart,pEnd,node);
      return node;
    }
    do { pE--; } while (pE>pStart && *pE!=delimiter);
  }
  Node* node = Node::getNodeRel(pStart+1,pEnd,Node::root);
  if (node) DentryCache::store(pStart,pEnd,node);
  return node;
}
```

3. (10 poena) Fajl sistem

Sistemski pozivi za pristup sadržaju binarnog fajla uobičajeno omogućavaju čitanje ili upis *size* susednih bajtova počev od pozicije *offset* u odnosu na početak sadržaja fajla (prvi bajt je na poziciji 0). Da bi ostvario ovakav pristup, neki kernel koristi klasu `FLogicalAccess` čiji je interfejs dat dole. Ova klasa koristi se tako što se za svaki ovakav pristup instancira objekat ove klase, inicijalizuje se pozivom operacije `reset` sa zadatim parametrima, a onda se iterira sve dok operacija `end` ne vrati 1. U svakoj iteraciji se pristupa odgovarajućim bajtovima jednog logičkog bloka fajla, tako što se može dobiti sledeća informacija:

- `getBlock`: vraća redni broj logičkog bloka u kom se pristupa u toj iteraciji;
- `getRelOffset`: vraća redni broj bajta u tekućem bloku počev od kog se pristupa;
- `getRelSize`: broj bajtova u tekućem bloku kojima se pristupa u toj iteraciji.

```
class FLogicalAccess {
public:

 FLogicalAccess ();
 void reset (size_t offset, size_t size);
 int  end() const;
 void next();

 size_t getBlock() const;
 size_t getRelOffset() const;
 size_t getRelSize() const;
};
```



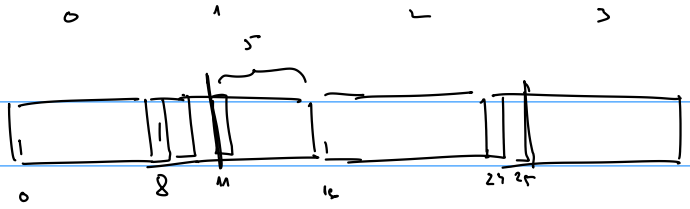
Primer upotrebe ove klase je sledeći:

```
FLogicalAccess fla;
for (fla.reset(offset,size); !fla.end(); fla.next()) {
  // Access fla.getRelSize() bytes
  // starting from the offset fla.getRelOffset()
  // in the logical block fla.getBlock()
}
```

Na primer, ako je veličina bloka 8 bajtova, a inicijalno je zadato: *offset* = 11 i *size* = 15, onda će u prvoj iteraciji biti: `getBlock()`=1, `getRelOffset()`=3, `getRelSize()`=5, u drugoj će ove vrednosti redom biti 2, 0, 8, a u trećoj 3, 0, 2; posle toga će se izaći iz petlje zbog `end()`=1.

Implementirati u potpunosti klasu `FLogicalAccess`. Veličina bloka je `BLK_SIZE`.

Rešenje:

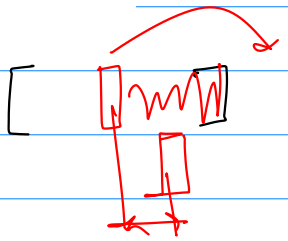


```
class FLogicalAccess {
public:

 FLogicalAccess ();
 void reset (size_t offset, size_t size);
 int  end() const;
 void next();

 size_t getBlock() const;
 size_t getRelOffset() const;
 size_t getRelSize() const;
};
```

Block
rel_offset
rel_size

Primer upotrebe ove klase je sledeći:

```
FLogicalAccess fla;
for (fla.reset(offset,size); !fla.end(); fla.next()) {
  //Access fla.getRelSize() bytes
  //starting from the offset fla.getRelOffset()
  //in the logical block fla.getBlock()
}
```



```
class FLogicalAccess {
 public:
        int block, rel_offset, rel_size; done = 0;
        int rem_size; offset
        void reset (size_t offset, size_t size) {
                rem_size = size;
                this->offset = offset;
                next();
        }
        void next() {
                block = offset / BLKSIZE;
                rel_offset = offset % BLKSIZE;
                if (rem_size > BLK_SIZE - rel_offset) {
                        rel_size = BLK_SIZE - rel_offset;
                        offset += rel_size;
                        rem_size -= rel_size;
                } else {
                        rel_size = rem_size;
                        rem_size = 0; offset += rel_size;
                        done = 1;
                }
        }
};
```

```
int eof() const {
    return done;
}

int getBlock() const { return block; }
int getRelSize() const { return rel_size; }
int getRelOffset() const { return rel_offset; }
};
```

K L D

2. (10 poena) Fajl sistem

Neprazno binarno stablo čiji su čvorovi tipa `Node` zapisano je u binarni fajl na sledeći način (prikazan je pojednostavljen kod bez obrade grešaka): za svaki čvor najpre je zapisan njegov sadržaj tipa `NodeData`, zatim jedan `int` indikator koji kaže da li taj čvor ima svoje levo podstablo, zatim jedan `int` indikator koji kaže da li taj čvor ima svoje desno podstablo, a onda isto tako redom celo levo podstablo ako ga ima, pa celo desno podstablo ako ga ima. Korišćenjem istog POSIX API za fajlove napisati kod funkcije `readTree` za učitavanje i izgradnju stabla iz takvog fajla. Prvi parametar je staza do fajla, a drugi parametar je adresa pokazivača u koji treba upisati adresu korenog čvora formiranog i učitanog stabla. Obraditi greške vraćanjem negativne vrednosti. POSIX API funkcija

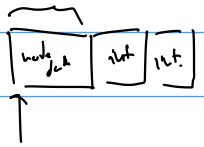
```
ssize_t read(int fd, void *buf, size_t count);
```

vraća broj stvarno učitanih bajtova (može biti manji od zahtevanog ako se stigne do kraja fajla) ili negativnu vrednost u slučaju greške.

```
#include <fcntl.h>

struct Node {
  NodeData data;
  Node *left, *right;
};

void writeSubtree (int fd, Node* node) {
  write(fd,node->data,sizeof(NodeData));
  int ind = (node->left!=0);
  write(fd,ind,sizeof(int));
  ind = (node->right!=0);
  write(fd,ind,sizeof(int));

  if (node->left) writeSubtree(fd, node->left);
  if (node->right) writeSubtree(fd, node->right);
}

void writeTree (const char* pathname, Node* root) {
 int fd = open(*pathname,O_WRONLY|O_CREATE|O_TRUNC);
 int r = writeSubtree(fd,root);
 close(fd);
}

int readTree (const char* pathname, Node** root);
```

Rešenje:

```
int readTree (const char * path, Node ** root) {
    int fd = open(path, O_RDONLY);
    if (fd < 0) return -1;
    int ret = readSubTree (fd, root);
    close (fd);
    return ret;
}
```




```
int readSubTree (int fd, Node ** root) {
    *root = new Node();
    int ret = read (fd, &(*root -> data), sizeof (NodeData));
    if (ret ≠ sizeof(NodeData)) return -1;
    int has_left, has_right;
    ret = read (fd, &has_left, sizeof(int));
    if (ret ≠ sizeof(int)) return -1;
    ret = read (fd, &has_right, sizeof(int));
    if (ret ≠ sizeof(int)) return -1;
    if (has_left) {
        ret = readSubTree (fd, &((*root) -> left));
        if (ret < 0) return -1;
    }
    if (has_right) {
        ret = readSubTree (fd, &((*root) -> right));
        if (ret < 0) return -1;
    }
    return 0;
}
```

0 stdin
1 stdout
2 stderr

dup(2) ⟶

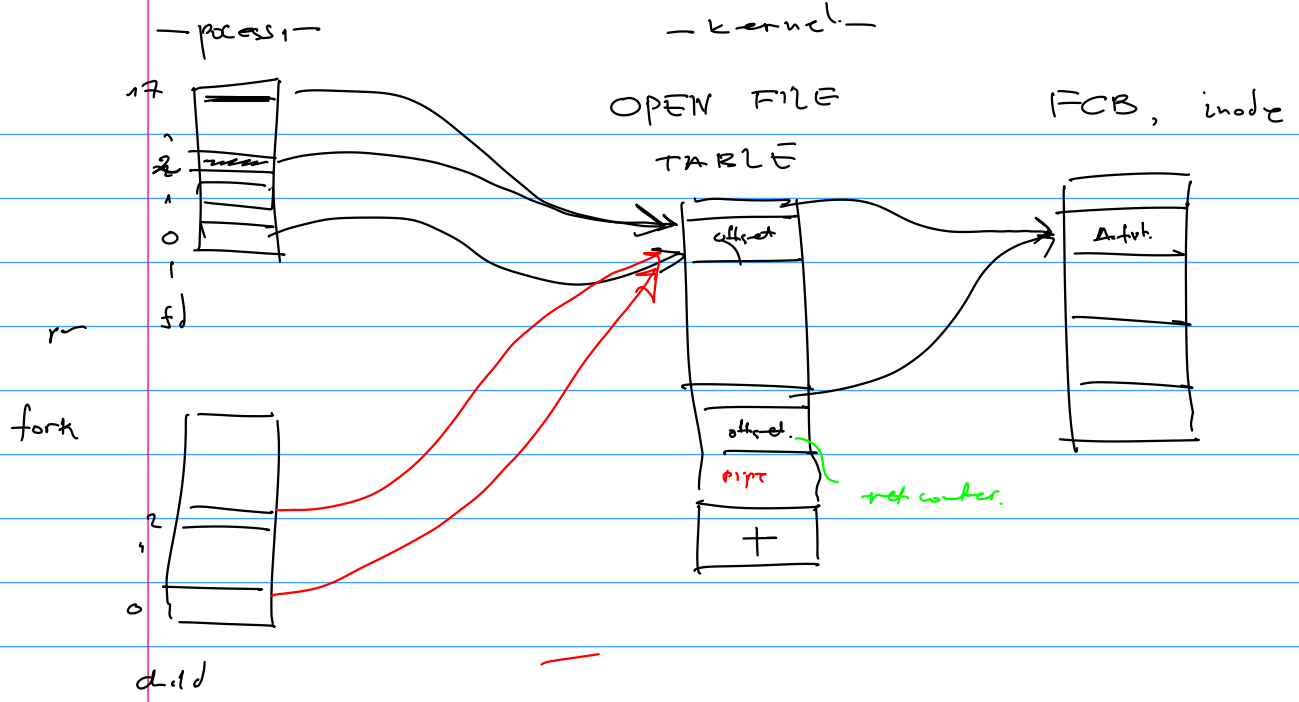
dup2 (old, new);

int fd = open("bla.txt", ..);

printf, scanf.

dup2 (fd, 1);

⟶ printf("hey");



2. (10 poena) Fajl sistem

U sistemima nalik sistemu Unix cevovod (*pipe*) se formira sistemskim pozivom *pipe*:

```
#include <unistd.h>
int pipe (int pipefd[2]);
```

Ovaj sistemski poziv pravi cevovod i u `pipefd[0]` upisuje deskriptor fajla (*file descriptor*) odredišne strane otvorenog cevovoda (strana za čitanje), a u `pipefd[1]` upisuje desktriptor fajla izvorišne strane tog cevovoda (strana za upis). U slučaju greške, ovaj poziv vraća negativnu vrednost, a u slučaju uspeha vraća 0. Dati su i sistemski pozivi za čitanje i upis:

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

Ovi pozivi prenose najviše `count` znakova sa fajlom koga identifikuje `fd` iz bafera ili u bafer na koga ukazuje `buf`, a vraćaju stvarno prenesen broj znakova.

Korišćenjem ovih, kao i drugih potrebnih sistemskih poziva i bibliotečnih funkcija (*fork*, *exit*, *close*, *putchar*, *fprintf*), napisati program koji, kada se nad njim pokrene proces, formira jedan cevovod kojim komunicira sa procesom detetom koga pokreće. Nakon toga, proces roditelj kroz cevovod procesu detetu šalje niz znakova dat svojim argumentom komandne linije `argv[1]` i završen znakom `'\0'`, a proces dete dobijeni niz znakova ispisuje na svoj standardni izlaz. Nakon što uradi navedeno, svaki od procesa se gasi. Greške obraditi ispisom na `stderr` i gašenjem procesa.

Rešenje:

```
#define READ_END 1
#define WRITE_END 0
int main(int argc, char *argv[]) {
    int pipefds[2];
    if (pipe(pipefds) < 0) {
        fprintf(stderr, "pipe failed\n");
        exit(1);
    }
```

```
pid_t cpid = fork();
if (cpid < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
}

if (cpid > 0) {  // parent:
    int ret = write(pipefd[WRITE_END], argv[1], 1 + strlen(argv[1]));
    if (ret != (1 + strlen(argv[1]))) {
        fprintf(stderr, "write failed\n");
        exit(1);
    }
} else {
    char c;
    while (1) {
        int ret = read(pipefd[READ_END], &c, sizeof(char));
        if (ret != sizeof(char)) {
            printf(stderr, "read failed\n");
            exit(1);
        }
        if (c == '\0') break;
        putc(c);
    }
}
```