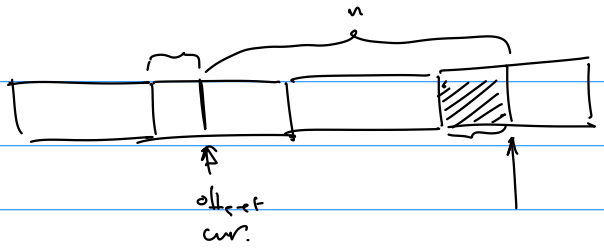
3. (10 poena) Implementacija fajl sistema

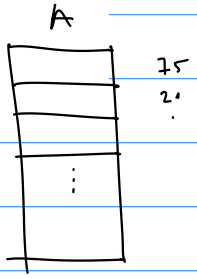
U implementaciji nekog fajl sistema definisani su celobrojni tip `Byte` koji predstavlja bajt, kao i celobrojni tip `BlkNo` koji predstavlja broj logičkog bloka sadržaja fajla (numeracija počev od 0). U strukturi FCB celobrojno polje `cur` predstavlja kurzor za čitanje i upis (u bajtovima, počev od 0), a polje `size` stvarnu veličinu u bajtovima. Konstanta `BLKSIZE` predstavlja veličinu bloka u bajtovima. Na raspolaganju je funkcija koja učitava logički blok datog fajla i vraća pokazivač na taj učitani blok u kešu (vraća 0 u slučaju greške):

```
Byte* readFileBlock (FCB* file, BlkNo blockNo);
```

Realizovati funkciju `fread()` koja za dati fajl učitava `n` bajtova u dati bafer, počev od kurzora, i vraća broj stvarno učitanih bajtova, a kurzor pomera na kraj učitane sekvence. U slučaju prekoračenja veličine sadržaja fajla ili druge greške treba vratiti broj stvarno učitanih bajtova:

```
int fread (FCB* file, Byte* buffer, int n);
```






```
int fread (FCB *file, Byte *buffer, int n) {
    int num_read = 0, to_read = n;
    while (num_read < n) {
        BlkNo blkno = file->cur / BLKSIZE;
        Byte * blk = readFile(file, blkno);
        if (blk == null) break;
        int rel_offset = file->cur % BLKSIZE;
        if (to_read >= (BLKSIZE - rel_offset)) {
            for (int i = 0; i < BLKSIZE - rel_offset; i++) {
                buffer[num_read++] = blk[rel_offset+i];
            }
            file->cur += BLKSIZE - rel_offset;
            to_read -= BLKSIZE - rel_offset;
        } else {
            for (int i=0; i < to_read; i++) {
                buffer[num_read++] = blk[rel_offset+i];
            }
            to_read = 0;
            file->cur += to_read;
        }
    }
    return num_read;
}
```

if (file->size - file->cur < n) {
    n = file->size - file->cur;
}

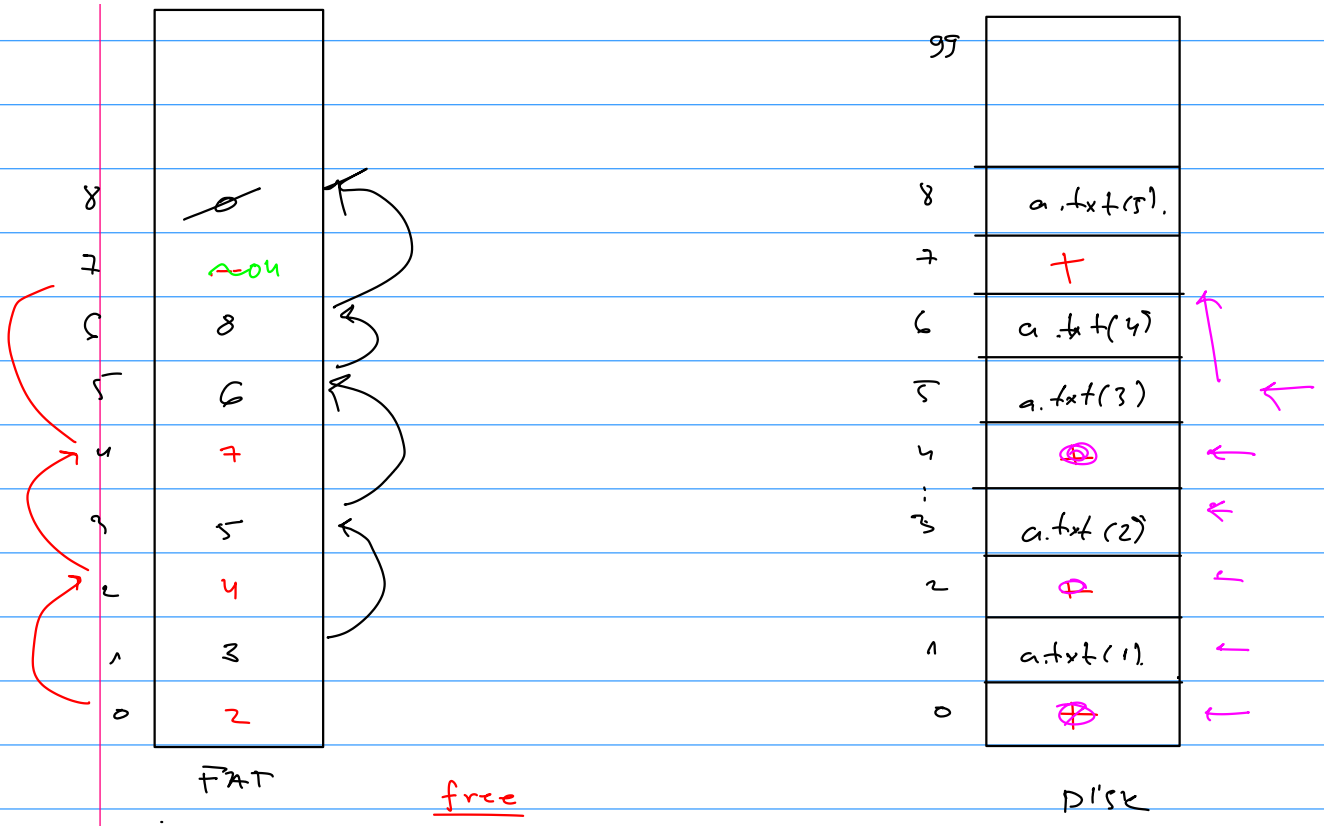
3. (10 poena) Implementacija fajl sistema

Neki fajl sistem primenjuje FAT za alokaciju sadržaja fajla. FAT je cela keširana u memoriji, na nju ukazuje pokazivač `fat`, i ima `FATSIZE` ulaza tipa `unsigned`. Prilikom ulančavanja blokova sa sadržajem fajla, *null* vrednost se označava vrednošću 0 u odgovarajućem ulazu u FAT, dok se slobodni blokovi ne ulančavaju posebno, već su njima odgovarajući ulazi u FAT označeni vrednostima `~0U` (sve jedinice binarno); blokovi broj 0 i broj `~0U` se ne koriste u fajl sistemu. U FCB polje `head` tipa `unsigned` sadrži broj prvog bloka sa sadržajem fajla (0 ako je sadržaj prazan).

Realizovati funkciju `extendFile()` datu dole, koja se koristi u implementaciji fajl sistema i koja treba da proširi sadržaj fajla za `by` blokova (da ih alocira i doda na kraj sadržaja fajla). Ova funkcija treba da vrati broj blokova kojim je stvarno proširen sadržaj fajla i koji može biti jednak `by`, ukoliko je u fajl sistemu bilo dovoljno slobodnog mesta, odnosno manji od te vrednosti (uključujući i 0), ukoliko nije bilo dovoljno slobodnih blokova.

```
unsigned extendFile (FCB* fcb, unsigned by);
```

Rešenje:



FCB a.txt 1

5 blokova.

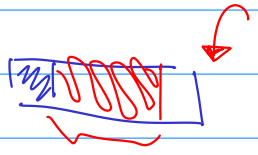
f->head == 0;

getfree.

```
int extendFile (FCB *f, unsigned by){
    int tail = f->head;
    while (tail != 0 && fat[tail] != 0){
        tail = fat[tail];
    }

    int blks = 0; int cur = 1;
    while (blks < by){
        int free = -1;
        while (cur < FATSIZE){
            if (fat[cur] == ~0u){
                free = cur
                cur += 1;
                break;
            }
            cur += 1;
        }
        if (free == -1) break;
        fat[tail] = free;
        fat[free] = 0;
        tail = free;
        blks ++;
    }
    return blks;
}
```

```
if (tail == 0){
    f->head = free;
} else {
    fat[tail] = free;
}
```



ext.

15 15.

FCB blks[15]

60KB

7 → blks[7].

3. (10 poena) Fajl sistem

Neki fajl sistem koristi indeksiranu alokaciju sadržaja fajla sa jednostepenim indeksom u jednom bloku. Struktura FCB keširana je u memoriji, kao i indeksni blok. U strukturi FCB, pored ostalih, postoje sledeća polja:

- `unsigned long size`: veličina sadržaja fajla u bajtovima;
- `unsigned long* index`: pokazivač na (keširan) indeks (ulazi su tipa `unsigned long`).

Osim toga, definisane su i sledeće konstante i funkcija:

- `unsigned long BlockSize`: veličina bloka na disku u bajtovima;
- `unsigned long MaxFileSize`: maksimalna dozvoljena veličina sadržaja fajla u bajtovima; jednaka je maksimalnom broju ulaza u indeksu pomnoženom veličinom bloka `BlockSize`;
- `unsigned long allocateBlock()`: alocira jedan slobodan blok na disku i vraća njegov broj; ukoliko slobodnog bloka nema, vraća 0.

Realizovati funkciju

```
unsigned long extend (FCB* fcb, unsigned extension);
```

koja proširuje sadržaj fajla na čiji FCB ukazuje prvi argument za broj bajtova dat drugim argumentom. Ukoliko traženo proširenje premašuje maksimalnu veličinu fajla ili na disku nema dovoljno slobodnog prostora, sadržaj fajla treba proširiti koliko je moguće. Ova funkcija vraća broj bajtova sa koliko je stvarno uspela da proširi sadržaj (jednako ili manje od traženog).

Rešenje:

```
unsigned long extend (FCB *f, unsigned extension){
    int old_size = f->size;
    int old_blks = f->size/BLKSIZE + (f->size%BLKSIZE ? 1 : 0);
    long new_size = f->size + extension;
    if (new_size > MaxFileSize) new_size = MaxFileSize;
    int new_blks = new_size/BLKSIZE + (new_size%BLKSIZE ? 1 : 0);

    int i;
    for (i = old_blks; i < new_blks; i++) {
        f->index[i] = allocateBlock();
        if (f->index[i] == 0) break;
    }

    if (i == new_blks) {
        f->size = new_size;
    } else {
        f->size = i*BLKSIZE;
    }

    return f->size - old_size;
}
```

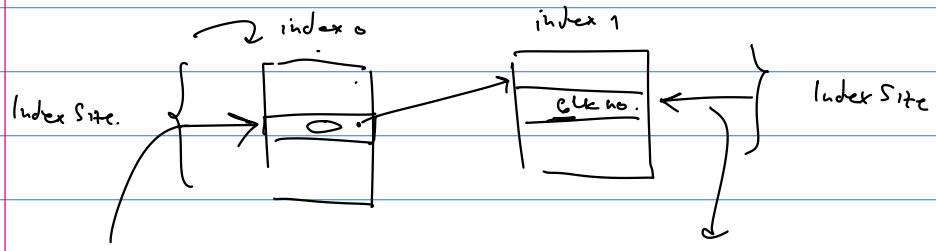
3. (10 poena) Implementacija fajl sistema

Neki fajl sistem primenjuje indeksiranu alokaciju fajlova, sa indeksima u dva nivoa. U FCB fajla nalazi se polje `index` tipa `unsigned long`, koje predstavlja broj bloka na disku u kome se nalazi indeks prvog nivoa. Indeksi oba nivoa su iste veličine `IndexSize` ulaza tipa `unsigned long`. Nepopunjeni ulazi u indeksima imaju vrednost 0. Operacija `getPBlock` po potrebi u keš učitava blok sa diska sa zadatim brojem i vraća pokazivač na mesto u kešu u koje je taj blok učitan; u slučaju greške vraća 0.

```
typedef unsigned long ulong;
extern const ulong IndexSize;
void* getPBlock(ulong pBlockNo);
```

Realizovati funkciju `getFileBlock()` datu dole, koja se koristi u implementaciji fajl sistema i koja treba da vrati adresu na učitan logički blok fajla sa zadatim (logičkim) brojem; u slučaju greške treba da vrati 0.

```
void* getFileBlock (FCB* fcb, ulong lBlockNo);
```



lBlockNo / IndexSize lBlockNo % IndexSize.

```
void * getFileBlock (FCB *fcb, ulong lBlockNo) {
    ulong * index = getPBlock (fcb->index);
    if (index == 0) return 0;
    ulong index1_blk = index [lBlockNo / IndexSize];
    ulong *index1 = getPBlock (index1_blk);
    if (index1 == 0) return 0;
    ulong fblk = index1 [lBlockNo % IndexSize];
    return getPBlock (fblk);
}
```

3. (10 poena) Implementacija fajl sistema

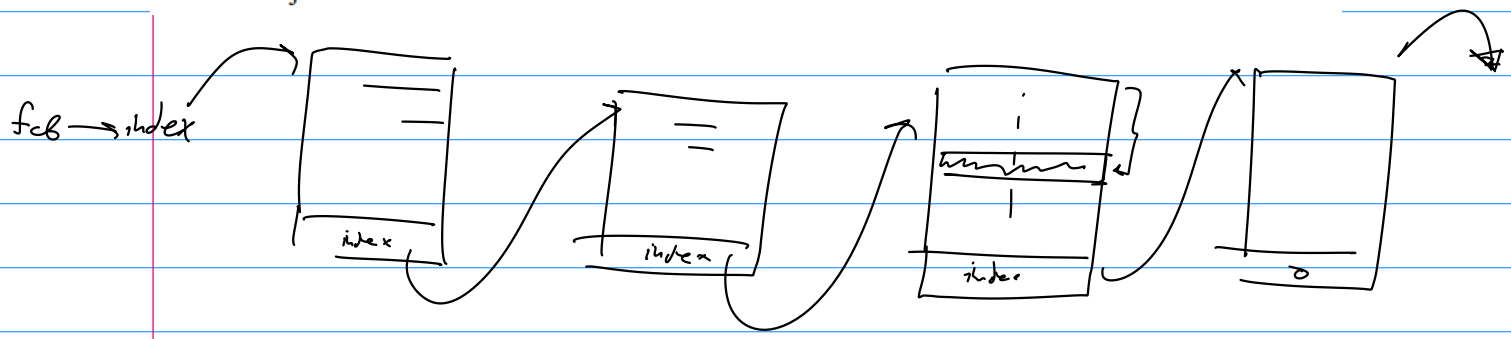
Neki fajl sistem primenjuje indeksirani pristup alokaciji prostora za sadržaj fajla, s tim da je indeks neograničen i organizovan kao jednostruko ulančana lista indeksnih blokova. Na prvi indeksni blok u listi ukazuje polje `index` u FCB. Svaki indeksni blok sadrži `NumOfEntries` ulaza tipa `BlkNo` koji ukazuju na blokove sa sadržajem fajla i još jedan ulaz (iza ovih) istog tipa koji ukazuje na sledeći indeksni blok u listi (vrednost 0 označava *null* ulaz). Na raspolaganju je funkcija za pristup blokovima diska kroz keš, koja vraća pokazivač na deo memorije u kome se nalazi traženi blok diska učitan u keš (vraća 0 u slučaju greške):

```
Byte* getDiskBlock (BlkNo block);
```

Realizovati funkciju `getFileBlock()` koja za dati fajl dohvata logički blok sa datim brojem. U slučaju prekoračenja veličine sadržaja fajla ili druge greške treba vratiti 0.

```
Byte* getFileBlock (FCB* file, unsigned int block);
```

Rešenje:



block / (NumOfEntries − 1) → index_num

block % (NumOfEntries − 1)

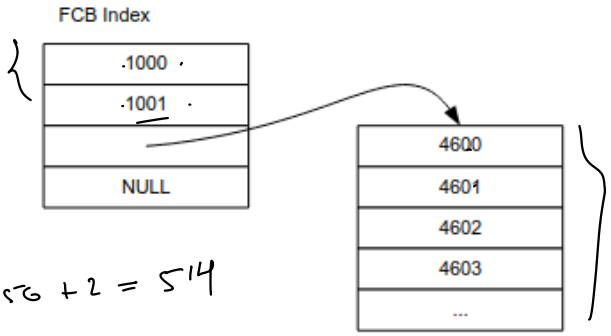
```
Byte * getFileBlock (FCB *f, ulong block) {
    int index_num = block/(NumOfEntries - 1);
    BlkNo * index = getDiskBlock (f->index);
    if (!index) return 0;
    for (int i=0; i<index_num; i++) {
        index = getDiskBlock (index[NumOfEntries-1]);
        if (!index) return 0;
    }
    return getDiskBlock (index[block % (NumOfEntries-1)]);
}
```

if (index[NumOfEntries-1] == 0) { ... error }

3. (10 poena) Implementacija fajl sistema

Neki fajl sistem primenjuje kombinovanu tehniku indeksirane alokacije sadržaja fajla. U FCB fajla nalaze se dva ulaza koji predstavljaju indeks nultog nivoa (direktni pokazivači na dva prva bloka sadržaja fajla) i još dva ulaza koji ukazuju na blokove sa indeksima prvog nivoa.

Na slici je prikazan deo FCB nekog fajla. Blok je veličine 1KB, a broj bloka (svaki ulaz u indeksnom bloku) zauzima 4 bajta. Bajtovi sadržaja fajla se broje počev od 0.



U kom bloku na disku se nalazi bajt sadržaja ovog fajla sa datim rednim brojem (pored konačnog odgovora, dati i celu računicu):

a)(3) 2000 (decimalno)?

Odgovor: 1001

Postupak i obrazloženje:

b)(3) 3570 (decimalno)?

Odgovor: 4601

Postupak i obrazloženje:

c)(4) Kolika je maksimalna veličina fajla koju dozvoljava ovaj sistem?

Odgovor: 514 KB.

Postupak i obrazloženje:

3. (10 poena)

Neki fajl sistem koristi FAT, uz dodatni mehanizam detekcije i oporavka od korupcije ulančanih lista na sledeći način. Kada se FAT kešira u memoriji, vidi se kao niz struktura tipa `FATEntry`. Ova struktura ima dva celobrojna polja. Polje `next` je broj ulaza u FAT u kome se nalazi sledeći element u ulančanoj listi; vrednost 0 označava kraj liste. Polje `fid` ove strukture sadrži identifikator fajla kome pripada taj element liste. U strukturi FCB celobrojno polje `id` predstavlja identifikator datog fajla, a polje `head` sadrži redni broj ulaza u FAT koji je prvi element u ulančanoj listi datog fajla.

Implementirati funkciju čiji je potpis dat dole. Ona treba da proveri konzistentnost ulančane liste datog fajla, proverom da li svi elementi liste pripadaju baš tom fajlu. Ukoliko je sve u redu, ova funkcija treba da vrati 1. Ukoliko naiđe na element u listi koji je pogrešno ulančan, odnosno ne pripada tom fajlu (tako što polje `fid` ne odgovara identifikatoru tog fajla), taj pogrešno ulančani ostatak liste treba da „odseče“ postavljanjem terminatora liste (vrednost 0 u polje `next`) u poslednji ispravan element liste (ili glavu liste, ako je prvi element pogrešan) i da vrati 0.

```
FATEntry FAT[...];

int check_consistency (FCB* file);
```

Rešenje:

```
int check_consistency (FCB *file) {
    int prev = -1;
    int cur = file->head;
    while (cur != 0) {
        if (FAT[cur].fid != file->id) {
            if (cur == file->head) {
                file->head = 0;
            } else {
                FAT[prev].next = 0;
            }
            return 0;
        }
        prev = cur;
        cur = FAT[cur].next;
    }
    return 1;
}
```

3. (10 poena) Fajl sistem

U implementaciji nekog fajl sistema evidencija slobodnih blokova na disku vodi se pomoću bit-vektora koji se kešira u memoriji u nizu `blocks` veličine `NumOfBlocks/BITS_IN_BYTE` (konstanta `NumOfBlocks` predstavlja broj blokova na disku). Svaki element ovog niza je veličine jednog bajta, a svaki bit odgovara jednom bloku na disku (1-zauzet, 0-slobodan). Pretpostaviti da je `NumOfBlocks` umnožak broja 8 (`BITS_IN_BYTE`).

```
typedef unsigned char byte;
const unsigned int BITS_IN_BYTE = 8;
const unsigned long NumOfBlocks = ...;
byte blocks[NumOfBlocks/BITS_IN_BYTE];
```

a)(3) Implementirati sledeće dve funkcije:

```
void blockToBit(unsigned long blkNo, unsigned long& bt, byte& mask);
void bitToBlk(unsigned long& blkNo, unsigned long bt, byte mask);
```

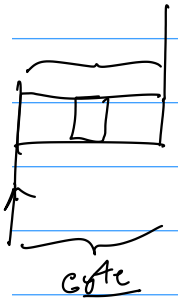
Funkcija `blockToBit` prima kao ulazni parametar redni broj bloka `blkNo` i na osnovu njega izračunava i upisuje u izlazne parametre broj bajta (`bt`) u bit-vektoru i jednu jedinu jedinicu u onaj razred parametra `msk` koji odgovara bitu u tom bajtu za dati blok. Funkcija `bitToBlock` radi obrnutu konverziju: za ulazni parametar koji je redni broj bajta u bit-vektoru (`bt`) i najniži razred u parametru `msk` koji je postavljen na 1, izračunava i upisuje u izlazni parametar `blk` redni broj bloka koji odgovara tom bajtu i bitu.

b)(7) Korišćenjem funkcija pod a), implementirati funkcije `allocateBlock` i `freeBlock`. Funkcija `allocateBlock` treba da pronađe prvi slobodan blok i označi ga zauzetim. Ako takav blok nađe, vraća broj tog bloka, a ako slobodnog bloka nema, vraća 0 (blok broj 0 je rezervisan i nikada se ne koristi u fajl sistemu). Kako bi se optimizovao pristup fajlovima, slobodan blok se traži u blizini bloka sa datim rednim brojem `startingFrom`, na sledeći način:

- slobodan blok treba tražiti počev od datog bloka naviše (i alocirati prvi slobodan na koji se naiđe), pri tom može (ali ne mora) da se alocira i bilo koji slobodan blok koji je u istom bajtu bit-vektora kao i dati blok, bez obzira da li je ispred ili iza datog bloka (i on se smatra dovoljno bliskim).
- ukoliko se na ovaj način ne pronađe, počinje se pretraga od prvog bloka (blok broj 0 je rezervisan) pa sve do zadatog bloka `startingFrom`.

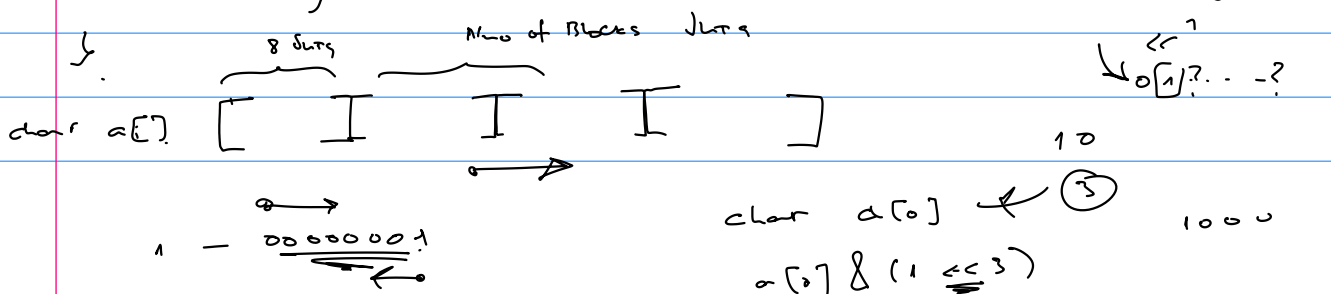
Funkcija `freeBlock` označava dati blok slobodnim.

```
unsigned long allocateBlock (unsigned long startingFrom);
void freeBlock (unsigned long blk);
```

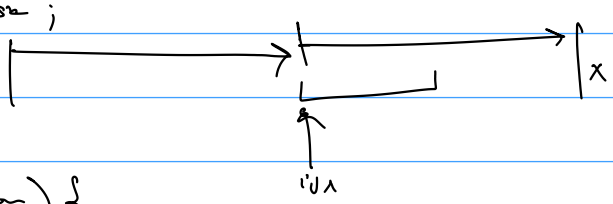



```
void BlockToBit ( ulong BlkNo, ulong &bt, byte &mask ){
    bt = blkNo / BITS_IN_BYTE;
    mask = (byte) 1 << (BlkNo % BITS_IN_BYTE);
}
```

```
void BitToBlock ( ulong &BlkNo, ulong bt, ulong mask ){
    blkNo = bt * BITS_IN_BYTE;
    while (!(mask & 1)) {
        mask >>= 1;
        blkNo += 1;
    }
}
```



```
void freeBlock (ulong blk) {
    ulong idx;
    byte mask;
    BlockToBit (blk, idx, mask);
    blocks[idx] &= ~mask;
}
```




```
ulong allocateBlock (ulong startingFrom) {
    ulong idx;
    byte mask;
    BlockToBit (startingFrom, idx, mask);
    for (int i=0; i < NumOfBlocks/BITS_IN_BYTE; i++) {
        ulong byte_idx = (idx + i) % (NumOfBlocks/BITS_IN_BYTE);
        for (int j=0; j < BITS_IN_BYTE; j++){
            if(!(blocks[byte_idx] & (1<<j))) {
                ulong blk;
                BitToBlock (blk, byte_idx, 1<<j);
                blocks[byte_idx] |= (1<<j);
                return blk;
            }
        }
    }
    return 0;
}
```

```
if (byte_idx == 0)
    continue;
```

3. (10 poena) Fajl sistem

U implementaciji nekog fajl sistema evidencija slobodnih blokova na disku vodi se na sledeći način. Indeks (spisak) slobodnih blokova je neograničen i zapisuje se u samim slobodnim blokovima, ulančanim u jednostruku listu. Prema tome, prvi slobodan blok sadrži spisak najviše N slobodnih blokova (*ne* uključujući njega samog, tj. on nije na spisku), dok poslednji ulaz u tom spisku u tom bloku sadrži broj sledećeg bloka u listi, u kome se nalazi nastavak spiska slobodnih blokova itd. Ukoliko je neki slobodni blok bio na spisku, a više nije, njegov ulaz u indeksu slobodnih blokova ima vrednost 0 (blok broj 0 nikada nije slobodan). Kada se zahteva jedan slobodan blok, treba jednostavno uzeti prvi slobodan blok sa spiska. Pri tome, ukoliko je ceo spisak sadržan u prvom bloku u listi ispražnjen, treba alocirati upravo taj prvi blok iz liste i izbaciti ga iz liste.

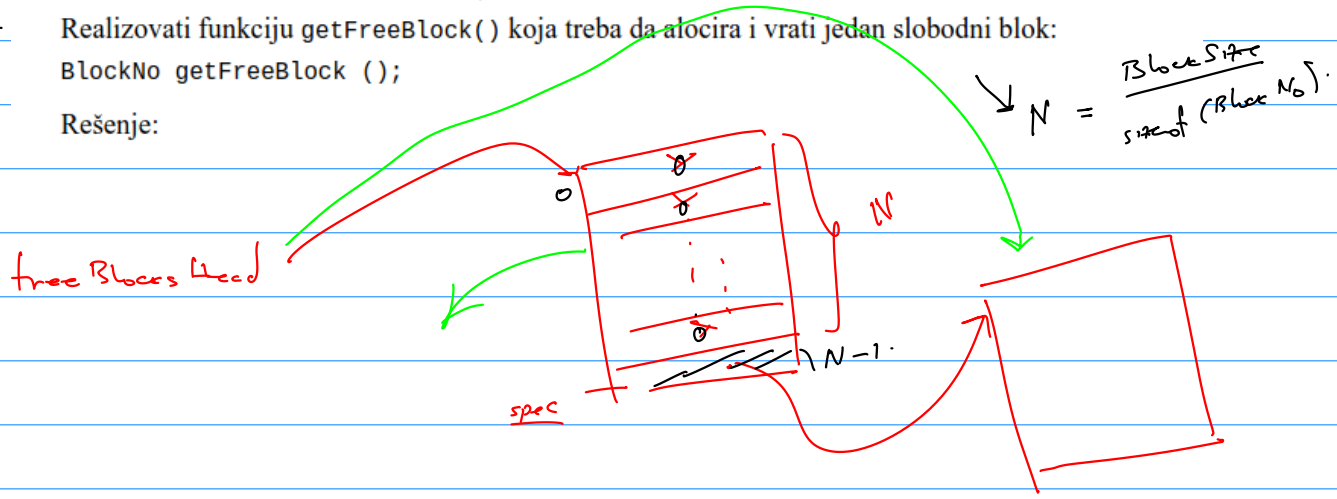
Na prvi blok u listi ukazuje globalna promenljiva `freeBlocksHead`. Na raspolaganju je i funkcija `getBlock` koja vraća pokazivač na deo memorije u kešu u koju je učitan traženi blok sa diska. (Napomena: broj N je određen veličinom bloka i veličinom tipa `BlockNo`.)

```
typedef … BlockNo; // Disk block number
extern int blockSize; // Disk block size
void* getBlock (BlockNo block);
extern BlockNo freeBlocksHead;
```

Realizovati funkciju `getFreeBlock()` koja treba da alocira i vrati jedan slobodni blok:

```
BlockNo getFreeBlock ();
```

Rešenje:




```
BlockNo getFreeBlock () {
    BlockNo * list = getBlock (freeBlocksHead);
    int N = BlockSize / sizeof(BlockNo);

    for ( int i=0; i < N-1; i++) {
        if ( list[i] ≠ 0) {
            BlkNo blk = list[i];
            list[i] = 0;
            return blk;
        }
    }

    BlkNo blk = freeBlocksHead;
    freeBlocksHead = list[N-1];
    return blk;
}
```

```
if ( list[i] == 0)
    return 0;
}
```

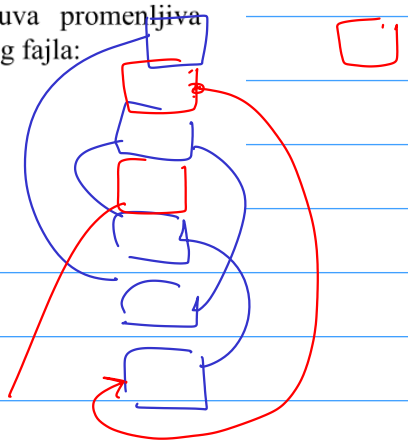
3. (10 poena) Fajl sistem

U implementaciji nekog FAT fajl sistema ceo FAT keširan je u memoriji u nizu `fat`:

```
extern uint32 fat[];
extern uint32 freeHead, freeCount;
```

Za ulančavanje se kao *null* vrednost u ulazu u FAT koristi 0 (blok broj 0 je rezervisan). U FCB fajla polje `head` sadrži redni broj prvog bloka sa sadržajem fajla i polje `size` koje sadrži veličinu sadržaja fajla u bajtovima. Slobodni blokovi su ulančani u jednostruku listu čija glava je u promenljivoj `freeHead`, dok ukupan broj slobodnih blokova čuva promenljiva `freeCount`. Implementirati sledeću funkciju koja treba da obriše sadržaj datog fajla:

```
void truncate (FCB* fcb);
```



Rešenje:

```
void truncate (FCB *f) {
    if (f == 0) return;
    if (file->size == 0) return;
    uint32 cur = f->head;
    int size_blks = 1;
    while (fat[cur] != 0) {
        size_blks++;
        cur = fat[cur];
    }

    if (freeHead == 0) {
        freeHead = f->head
    } else {
        uint32 cur = freeHead;
        while (fat[cur] != 0) {
            cur = fat[cur];
        }
        fat[cur] = f->head;
    }

    f->size = 0;
    freeCount += size_blks;
}
```