File System

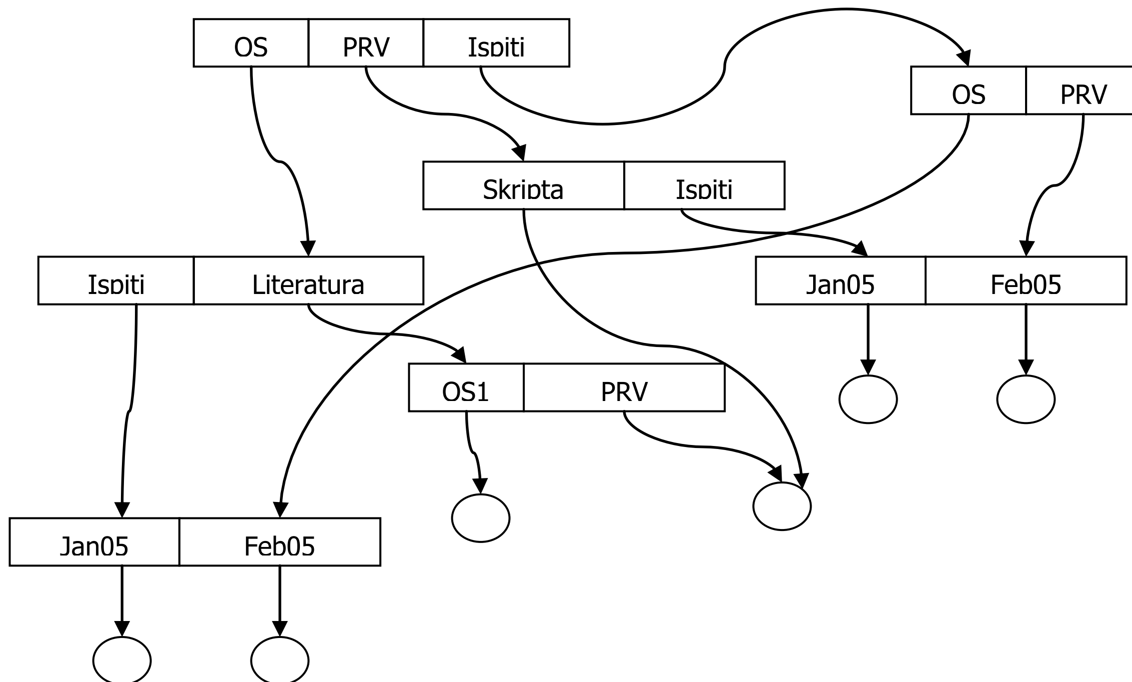
Zadatak 1.

(a)(10) Na slici je grafički prikazana trenutna struktura direktorijuma u jednom fajl sistemu koji podržava strukture direktorijuma tipa DAG (*directed acyclic graph*). Pravougaonici na slici predstavljaju direktorijume, a krugovi fajlove. Operacija `copy` ovog sistema svoj prvi argument ne kopira fizički, već samo pravi novu referencu na isti element u odredišnom direktorijumu zadatom kao drugi argument ("plitko kopiranje"). Operacija `clone` fizički kopira prvi argument u odredišni direktorijum zadat kao drugi argument (pravi novi fajl pod istim imenom u odredišnom direktorijumu i fizički kopira njegov sadržaj - "duboko kopiranje"). Operacija `delete` briše referencu na fajl i sam fizički fajl ukoliko na njega više nema referenci. Nad ovim fajl sistemom izvršena je sledeća sekvenca operacija:

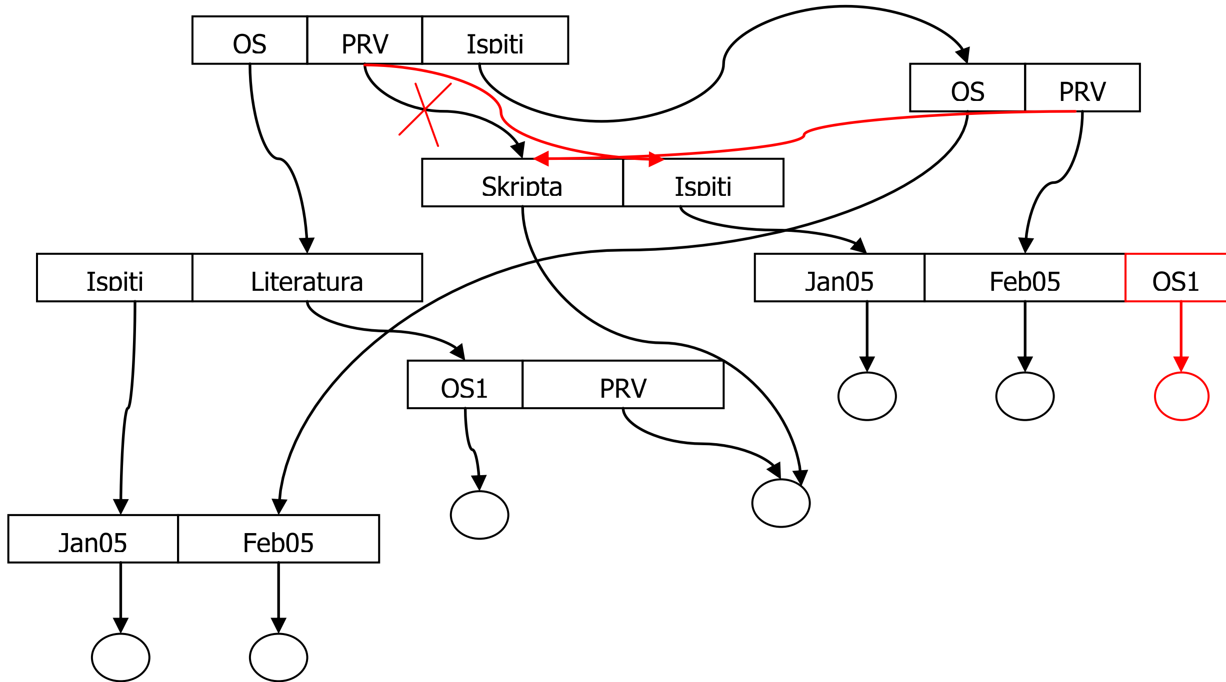
```
copy  /PRV/Skripta  /Ispiti/PRV
delete  /PRV/Skripta
clone  /OS/Literatura/OS1  /Ispiti/PRV
```

Komanda `dir` ispisuje nazive elemenata navedenog direktorijuma. Na liniju pored svake dole navedene komande napisati šta će ispisati sledeće operacije (listu naziva elemenata razdvojiti zarezima, npr. `Jan05, PRV, Literatura`):

1. `dir /PRV` ____
2. `dir /Ispiti/PRV` ____
3. `dir /OS/Literatura` ____



Rešenje:



1. `Ispiti`
2. `Jan05, Feb05, Skripta, OS1`
3. `OS1, PRV`

Zadatak 2.

(a)(10) U nekom fajl sistemu postoji koncept „tekućeg direktorijuma“ za dati proces, pa se fajlovi mogu referisati svojom punom stazom od korenog direktorijuma, ili relativno u odnosu na tekući direktorijum procesa, pri čemu oznaka `..` (dve tačke) označava roditeljski direktorijum (jedan korak naviše u stablu direktorijuma), a vodeći znak `/` (kosa crta) u punoj stazi označava koreni direktorijum. Direktorijumi su organizovani u stablo. Dopuniti prazna polja u sledećoj tabeli:

Puna staza	Tekući direktorijum	Relativna staza
`/users/dmilicev/nastava/os1`		`nastava/os1`
`/users/dmilicev`	`/users/dmilicev/nastava/os1`	
	`/users/dmilicev/nastava/os1/ispiti`	`..`
`/users/dmilicev/nastava/os2`	`/users/dmilicev/nastava/os1`	
	`/users/dmilicev/nastava/os1`	`../../radovi/oois`

Rešenje:

Puna staza	Tekući direktorijum	Relativna staza
/users/dmilicev/nastava/os1	/users/dmilicev	nastava/os1
/users/dmilicev	/users/dmilicev/nastava/os1	../..
/users/dmilicev/nastava/os1	/users/dmilicev/nastava/os1/ispiti	..
/users/dmilicev/nastava/os2	/users/dmilicev/nastava/os1	../os2
/users/dmilicev/radovi/oois	/users/dmilicev/nastava/os1	../../radovi/oois

Zadatak 3.

Neki fajl sistem podržava implicitno zaključavanje fajla prilikom njegovog otvaranja. Postoje dve vrste ključa: deljeni (*shared*, S), koji se traži prilikom otvaranja fajla samo za čitanje (proces koji je tako otvorio fajl ima pravo samo da čita iz fajla) i ekskluzivni (*exclusive*, X), koji se traži prilikom otvaranja fajla i za upis (proces koji je otvorio fajl ima pravo upisa). Popuniti sledeću tabelu upisivanjem oznaka onih procesa čiji će zahtev za otvaranjem istog fajla biti ispunjen, za svaki od dva data nezavisna slučaja. Procesi postavljaju zahteve redom navedenim u drugoj koloni, pri čemu oznaka npr. A-Rd označava da proces A postavlja zahtev za otvaranjem fajla za čitanje, a B-Wr da proces B postavlja zahtev za otvaranjem fajla za upis.

Rešenje:

Slučaj	Zahtevi za otvaranje fajla	Procesi koji su uspeli da otvore fajl
1	A-Rd, B-Rd, C-Wr, D-Rd, E-Wr	A, B, D
2	A-Wr, B-Wr, C-Rd, D-Rd, E-Rd	A

Zadatak 4.

Objasniti zašto se prava pristupa do fajla (prava na izvršenje određenih operacija nad fajlom) po pravilu čuvaju u tabeli otvorenih fajlova koja pripada kontekstu procesa, a ne u globalnoj (sistemskoj) tabeli otvorenih fajlova zajedničkoj za sve procese.

Odgovor:

- Po pravilu, opšta prava pristupa do fajla definišu se na nivou korisnika. Svaki proces se pokreće „u vlasništvu“ datog korisnika, pa time i nasleđuje prava pristupa do fajla podešena za tog korisnika. Zato se prava pristupa do fajla razlikuju od procesa do procesa i nisu globalna za dati fajl u celom sistemu.
- Bez obzira na opšta prava koja korisnik koji je kreirao proces ima do fajla, API za fajl sistem po pravilu omogućava da proces otvori fajl i da pri tome deklariše kako će ga koristiti (koje grupe operacija će koristiti). Ovo omogućava kontrolu grešaka u samom programu, tako što sistem zabranjuje one operacije koje nisu predviđene pri otvaranju fajla, smatrajući ih prekršajem zbog greške u programu. Zato se prava pristupa razlikuju od procesa do procesa, čak i za istog korisnika.

Zadatak 5.
Posmatra se disk kapaciteta 80MB i blokom veličine 1KB. Ako se za evidenciju slobodnog prostora koristi bit-vektor sa maksimalnom kompakcijom (svih 8 bita u bajtu su iskorišćeni itd.), koliko celih blokova treba zauzeti na disku za smeštanje ovog vektora?

Odgovor:
Račun: Broj blokova: 80MB/1KB = $80*2^{20}$B / 2^{10} B = $80*2^{10}$ = 80K. Veličina bit-vektora: 80Kbita / 8 bita po bajtu = 10KB. Broj blokova za bit-vektor: 10.

Zadatak 6.

Neki fajl sistem podržava montiranje nekog direktorijuma fajl sistema sa nekog eksternog uređaja ili udaljenog računara u bilo koji prazan lokani direktorijum, i to na više mesta (u različite prazne lokalne direktorijume). U fajl sistemu eksternog uređaja `X` nalazi se direktorijum `\first\second` u kome su tri fajla: `a.txt`, `b.txt` i `c.txt`. Šta će ispisati poslednja naredba u sledećoj sekvenci naredbi (sve su izvršene uspešno; prvi argument komandi `mount` i `copy` je izvorište, drugi odredište):

```
mount X:\first\second \home\buzz
mount X:\first\second \home\foo
del \home\foo\a.txt
copy \home\buzz\b.txt \home\foo\d.txt
dir \home\buzz
```

Odgovor:

```
b.txt
c.txt
d.txt
```

OS menja strukturu direktorijuma svog fajl sistema tako da montirani fajl sistem postaje podstruktura (poddirektorijum) od tačke montiranja

Zadatak 7.

Neki fajl sistem koristi indeksirani pristup alokaciji blokova za fajlove. U jedan indeksni blok može da stane *N* pokazivača na blokove na disku. Odrediti maksimalnu veličinu fajla (u blokovima) za svaku od sledećih varijanti organizacije indeksa:

i)(3) Jedan indeksni blok na koga ukazuje određeno polje u FCB.

Odgovor: *N* blokova

ii)(3) Ulančavanje indeksnih blokova, pri čemu na prvi indeksni blok ukazuje određeno polje u FCB, a poslednji ulaz u svakom indeksnom bloku je pokazivač na sledeći indeksni blok u lancu.

Odgovor: Nema logičkog ograničenja.

iii)(4) Indeksiranje u dva nivoa, s tim da na indeksni blok prvog nivoa ukazuje određeno polje u FCB, a svaki indeks prvog nivoa sadrži pokazivače na indeksne blokove drugog nivoa.

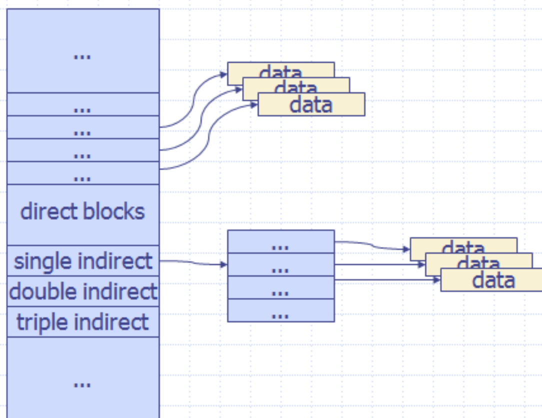
Odgovor: N^2 blokova

Zadatak 8.

Neki fajl sistem koristi indeksirani pristup alokacije blokova za fajlove na disku, sa kombinovanom tehnikom indeksiranja u jednom, dva i tri nivoa, kao kod UNIX *inode* strukture. Pretpostavljajući da disk ima uniformno srednje vreme pristupa do bilo kog bloka na disku, da li je vreme pristupa do različitih delova veoma velikih fajlova jednako? Ako jeste, precizno objasniti zašto jeste, a ako nije, objasniti kako se i zašto razlikuje.

Odgovor:

Nije. Vreme pristupa raste za blokove bliže kraju veoma velikog fajla, jer se za pristup do njih mora prolaziti kroz višestepene indekse. Pristup do delova bližim početku fajla je brži jer prolazi samo kroz jednostepeni indeks.



Zadatak 9.

Neki fajl sistem podržava volumene maksimalnog kapaciteta 64MB i blokom veličine 1KB. Fajl sistem baziran je na FAT (*File Allocation Table*). Vrednosti 0 i 1 u ulazu u FAT se mogu iskoristiti za označavanje slobodnog bloka ili završnog bloka u lancu zauzetih, pošto su blokovi 0 i 1 uvek sigurno rezervisani za sistemske potrebe (samu FAT). Koliko blokova zauzima FAT?

Račun:
Broj blokova na volumenu iznosi 64MB/1KB = 64K = 2^{16}.
Jedan ulaz u FAT treba da smesti adresu bloka od 16 bita, pa zauzima 2B.
Jedan blok može da smesti 1KB/2B = 512 = 2^9 ulaza u FAT.
FAT ima 2^{16} ulaza, pa tako zauzima $2^{16}/2^9 = 2^7 = 128$ blokova.

Odgovor: FAT zauzima 2^7=128 blokova.

Zadatak 10.

Veličina bloka na disku je 512B, a fajl sistem podržava adresiranje 4G blokova na disku. Fajlovi se alociraju indeksirano (*indexed allocation*) sa indeksima u jednom nivou i bez ulančavanja indeksnih blokova (indeks je samo jedan blok). Kolika je maksimalna veličina fajla koju podržava ovaj sistem?

Račun:
Maksimalan broj blokova na disku: $4G = 2^{32}$
Zbog toga jedan ulaz u indeksu zauzima 32 bita, odnosno $4B = 2^2B$
Broj ulaza u indeksnom bloku: $2^9/2^2=2^7=128$
Maksimalna veličina fajla je: 128 blocks * 512B/block = 64KB.

Odgovor: 64KB

Zadatak 11.

Neki fajl sistem koristi indeksiranu alokaciju fajlova na disku sa jednostrukim indeksom. Ako se pretpostavlja da je prostor za smeštanje fajlova (uključujući i njihove indekse) na disku veličine 32 GB, veličina klastera (jedine jedinice alokacije) 2 KB, i ceo prostor potpuno ispunjen fajlovima tako da je svaki fajl maksimalne veličine takve da ima samo jedan indeksni klaster, koliki procenat ukupnog prostora za smeštanje fajlova na ovom disku zauzimaju indeksi?

Odgovor: 100/__________% (odgovor izraziti kao razlomak sa brojiocem 100)

Račun:
Ukupan broj klastera: $32GB/2KB = 2^5 \times 2^{30}B/2^1 \times 2^{10}B = 2^{24} = 16M$
Jedan ulaz u indeksnom klasteru mora da adresira 16M klastera, pa zauzima 24 bita, tj. 3B.

Zato jedan indeksni klaster može da sadrži 2KB/3B = 2048B/3B = 682 ulaza, tj. da adresira 682 klastera sa podacima. Drugim rečima, na jedan indeksni klaster dolazi 682 klastera sa podacima, odnosno indeksni klasteri zauzimaju 1/683 deo prostora za fajlove, ili 100/683 procenata (~0.15%).

Odgovor: 100/683 procenata

Zadatak 12.

U fajl podsistemu nekog operativnog sistema ne vodi se tabela otvorenih fajlova za svaki proces, već postoji samo jedna globalna tabela otvorenih fajlova za ceo sistem. Drugim rečima, ne postoji nikakva informacija o upotrebi otvorenog fajla lokalna (privatna) za pojedinačni proces, već su sve takve informacije globalno deljene. Da li pojam pokazivača trenutne lokacije (kurzora) za čitanje/upis u fajl ima smisla čuvati u globalnoj tabeli otvorenih fajlova i zašto? Kako treba da izgleda API funkcija za čitanje bloka podataka dužine `len` iz nekog fajla, da bi se procesu obezbedila mogućnost sekvencijalnog čitanja svih podataka iz fajla? Napisati C funkciju koja iz celog datog otvorenog fajla čita sekvencijalno slogove od po *k* znakova korišćenjem predložene API funkcije.

Rešenje:

Pojam pokazivača trenutne lokacije (kurzora) za čitanje/upis u fajl nema smisla čuvati u globalnoj tabeli otvorenih fajlova, pošto bi to bio deljeni podatak koji uporedni procesi mogu da menjaju nezavisno, svaki za sebe. Naime, jedan proces koji želi sekvencijalno da čita slogove podataka iz nekog deljenog fajla ne može da se osloni na očuvanje vrednosti kurzora između dve svoje sukcesivne operacije čitanja, pošto u međuvremenu neki drugi proces može da pomeri taj kurzor.

Da bi se obezbedilo sekvencijalno čitanje, potrebno je da operacija čitanja kao svoj *argument* prenosi poziciju za čitanje, a ne da tu poziciju uzima iz deljenog podatka – kurzora:

```
IOStatus readFile (FileHandle fh, void* buffer, unsigned long cursor,
unsigned long len);
```

C funkcija koja iz celog datog otvorenog fajla čita sekvencijalno slogove od po *k* znakova:

```
void readFile (FileHandle fh) {
  char record[K];
  IOStatus ios = OK;

  for (unsigned long cur = 0; ios==OK; cur+=K) {
    ios=readFile(fh,record,cur,K);
    if (ios==OK) {
      ... // Radi nešto sa podacima učitanim u record
    }
  }
}
```

Zadatak 13.

Neki fajl sistem koristi indeksirani pristup alokaciji fajlova sa kombinovanim indeksom u jednom i dva nivoa. Polje `index1` u strukturi FCB sadrži broj fizičkog bloka u kome je indeks prvog nivoa (čiji ulazi sadrže brojeve blokova sa sadržajem), dok polje `index2` sadrži broj fizičkog bloka u kome je indeks drugog nivoa (čiji ulazi sadrže brojeve blokova sa indeksima u kojima su brojevi blokova sa sadržajem). Broj fizičkog bloka 0 u indeksu označava *null* - nealociran ulaz (fizički blok broj 0 se nikada ne koristi za fajlove). Konstanta `INDEXSIZE` definiše broj ulaza u jednom indeksu (u jednom bloku), a tip `BLKNO` predstavlja broj bloka (logičkog ili fizičkog).

a)(10) Realizovati funkciju:

```
int f_blkalloc(FCB* file, BLKNO logical, BLKNO physical);
```

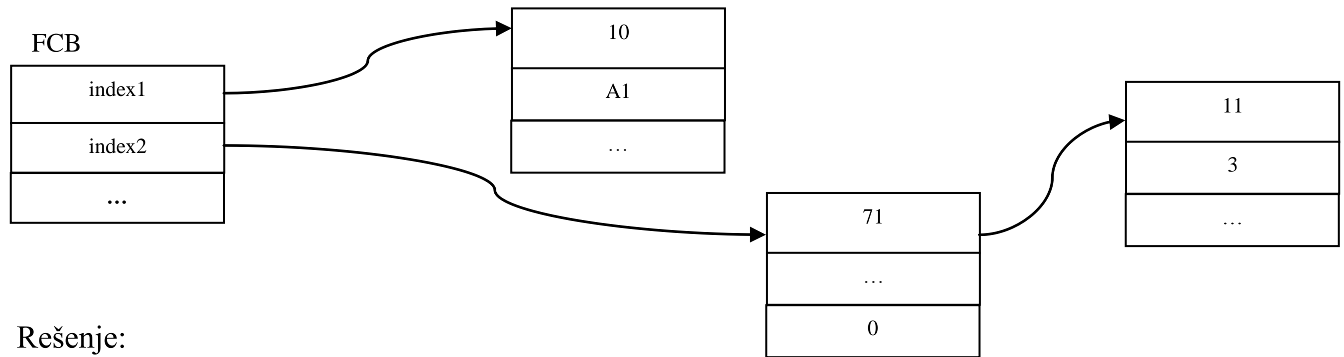
koja ažurira indekse datog fajla tako da proglašava dati logički blok alociranim u datom fizičkom bloku. U slučaju uspeha funkcija vraća 0, u slučaju greške -1. Pretpostaviti da je indeks prvog nivoa uvek inicijalno alociran, dok se indeks drugog nivoa alocira samo po potrebi. Na raspolaganju je funkcija:

```
void* f_getblk(BLKNO physical);
```

koja vraća pokazivač na keširani blok sa datim fizičkim brojem i učitava ga u keš ako je potrebno, kao i funkcija:

```
BLKNO f_blkalloc();
```

koja alocira jedan slobodan fizički blok i vraća njegov broj (0 u slučaju neuspeha).



Rešenje:

```
// Allocates a physical block, maps it in the cache,
// and initializes it with zeros:
BLKNO f_newblk () {
  BLKNO blk = f_blkalloc();
  if (blk==0) return 0;

  BLKNO* p = (BLKNO*)f_getblk(blk);
  if (p==0) return 0;
  for (int i=0; i<INDEXSIZE; i++) p[i]=0;
  return blk;
}

int f_blkalloc(FCB* file, BLKNO lb, BLKNO pb) {
  if (file==0 || pb==0) return -1; // Error

  if (lb<INDEXSIZE) {  // Single-level index
    BLKNO* pIndex = (BLKNO*)f_getblk(file->index1);
    if (pIndex==0) return -1;
    pIndex[lb]=pb;

  } else {             // Double-level index
    int i2=(lb-INDEXSIZE)/INDEXSIZE; //second level index
    int i1=(lb-INDEXSIZE)%INDEXSIZE; //first level index
    if (i2>=INDEXSIZE) return -1;

    if (file->index2==0) {
      file->index2=f_newblk();
      if (file->index2==0) return -1;
    }
    BLKNO* pIndex2 = (BLKNO*)f_getblk(file->index2);
    if (pIndex2==0) return -1;

    BLKNO index1 = pIndex2[i2];
    if (index1==0) {
      pIndex2[i2]=f_newblk();
      if (pIndex2[i2]==0) return -1;
    }
    BLKNO* pIndex1 = (BLKNO*)f_getblk(pIndex2[i2]);
    pIndex1[i1]=pb;
  }

  return 0;
}
```

b)(5) Kolika je maksimalna moguća veličina fajla, ako je `INDEXSIZE`=1024, a tip `BLKNO` je 64-bitni neoznačeni ceo broj?

Rešenje:

`INDEXSIZE` definiše broj ulaza u jednom indeksu (u jednom bloku), pa je:

```
BLOCKSIZE = BLKNO*INDEXSIZE = 8B*1024 = 8KB

MAXFILESIZE = (INDEXSIZE + INDEXSIZE*INDEXSIZE)*BLOCKSIZE =
            = (1K+1M)*8KB = 8MB + 8GB ~ 8GB
```

Zadatak 14.

Neki fajl sistem koristi indeksirani pristup alokaciji fajlova sa neograničenim brojem blokova za indeks fajla, ulančanim u jednostruku listu. Polje `index` u strukturi FCB sadrži broj prvog indeksnog bloka u listi. Broj narednog indeksnog bloka u listi nalazi se na samom početku svakog indeksnog bloka. Broj bloka 0 u indeksu označava *null* - nealociran ulaz (fizički blok broj 0 se nikada ne koristi za fajlove). Konstanta `INDEXSIZE` definiše broj ulaza u jednom indeksu (u jednom bloku), uključujući i prvi ulaz koji ukazuje na sledeći indeksni blok, a tip `BLKNO` predstavlja broj bloka (logičkog ili fizičkog).

a)(10) Realizovati funkciju koja se koristi kod direktnog pristupa fajlu:

```
void* f_getblk(FCB* file, BLKNO logical);
```

koja vraća pokazivač na dohvaćeni i keširani blok sa podacima fajla sa datim logičkim brojem. Na raspolaganju je funkcija:

```
void* f_getblk(BLKNO physical);
```

koja vraća pokazivač na keširani blok sa datim fizičkim brojem i učitava ga u keš ako je potrebno.

Rešenje:

```
struct Block {
  BLKNO next;
  BLKNO index[INDEXSIZE-1];
};

void* f_getblk(FCB* file, BLKNO lb) {
  if (file==0) return 0; // Exception: null FCB

  Block* blk = (Block*)f_getblk(file->index);
  if (blk==0) return 0; // Exception: error accessing index!

  while (lb>=INDEXSIZE-1) {
    lb -= INDEXSIZE-1;
    blk = (Block*)f_getblk(blk->next);
    if (blk==0) return 0;  // Exception: lb out of bound!
  }

  return f_getblk(blk->index[lb]);
}
```

Zadatak 15.

Neki fajl sistem koristi ulančani pristup alokaciji blokova za fajlove, pri čemu se u prvoj reči (tip `unsigned int`) svakog bloka alociranog za sadržaj fajla nalazi broj bloka koji je sledeći u lancu (0 za kraj). U FCB fajla polje `head` ukazuje na prvi, polje `tail` na poslednji blok u lancu alociranih blokova, a polje `size` sadrži broj blokova alociranih za sadržaj. Podsistem za alokaciju slobodnih blokova takođe ulančava slobodne blokove i pri alokaciji nekoliko slobodnih blokova od jednom vraća ceo lanac alociranih blokova. Postoji i podsistem koji kešira blokove sa sadržajem fajlova. U sistemu postoje realizovane sledeće funkcije:

```
int alloc(int numOfBlocks,unsigned int* head,unsigned int* tail);
```

Alocira lanac od `numOfBlocks` blokova i broj prvog bloka u alociranom lancu upisuje u promenljivu na koju ukazuje argument `head`, a poslednjeg u promenljivu na koju ukazuje `tail`; u slučaju uspeha vraća 0, u slučaju greške vraća kod greške (!=0).

```
int block_write(unsigned int blockNo,int word,int length,unsigned int* buf)
```

U podsistemu koji kešira blokove, u blok `blockNo`, počev od reči sa brojem `word` (broji se počev od 0), upisuje `length` reči sa lokacije na koju ukazuje `buf`; u slučaju uspeha vraća 0, u slučaju greške vraća kod greške (!=0).

Realizovati funkciju

```
int append(FCB* file, int numOfBlocks);
```

koja sadržaj datog fajla na čiji FCB ukazuje prvi argument proširuje dodavanjem `numOfBlocks` slobodnih blokova na kraj lanca. Ova funkcija u slučaju uspeha treba da vraća 0, a u slučaju greške da vraća kod greške (!=0).

Rešenje:

```
int append(FCB* file, int numOfBlocks) {
  unsigned int appendChainHead=0, appendChainTail=0;
  int status = alloc(numOfBlocks,&appendChainHead,&appendChainTail);
  if (status!=0) return status;
  if (file->tail) {
    status=block_write(file->tail,0,1,&appendChainHead);
    if (status!=0) return status;
  } else
    file->head=appendChainHead;
  file->tail=appendChainTail;
  file->size+=numOfBlocks;
  return 0;
}
```

Zadatak 16.

b)(10) Neki fajl sistem koristi FAT (*File Allocation Table*) pristup alokaciji blokova za fajlove. FAT se keširа u memoriju u strukturu deklarisanu na sledeći način:

```
typedef unsigned long int FATIndex;  // FAT entry number
const FATIndex FATSize = ...; // FAT (and disk) size

struct FATEntry {
  FATIndex next;  // Next block in the block chain; 0 for terminator
};

FATEntry fat[FATSize] ; // The FAT
FATIndex freeHead; // The head of the list of free blocks
FATIndex freeTail; // The tail of the list of free blocks
```

Slobodni ulazi ulančani su u jednostruku listu na čiji prvi ulaz pokazuje `freeHead`, a na poslednji `freeTail`. Kontrolni blok fajla (*File Control Block*, FCB) je deklarisan na sledeći način:

```
struct FCB {
  ...
  FATIndex sizeInBlocks; // Current file size in number of blocks
  FATIndex head; // Pointer to the first block (head FAT entry number)
  FATIndex tail; // Pointer to the last block (tail FAT entry number)
  ...
};
```

Potrebno je realizovati funkciju:

```
void clearFile (FCB* file);
```

koja treba da „obriše" sadržaj fajla, odnosno dealocira blokove koje fajl zauzima, bez uklanjanja fajla kao objekta iz fajl sistema.

Resenje:

```
void clearFile (FCB* file) {
  if (file==0 || file->sizeInBlocks==0 ||
      file->head==0 || file->tail==0) return;
  if (freeHead!=0)
    fat[freeTail].next = file->head;
  else
    freeHead = file->head;
  freeTail = file->tail;
  file->head=0;
  file->tail=0;
  file->sizeInBlocks=0;
}
```

Zadatak 17.

U nekom fajl sistemu direktorijum i fajl se uopšteno nazivaju *ulazom* (*entry*) i predstavljaju se istom strukturom FCB. U sistemu su implementirane sledeće elementarne operacije:

- `int f_find_entry(char* name, FCB** fcb)`: pronalazi ulaz sa zadatom punom stazom (imenom) i u `*fcb` upisuje pokazivač na FCB strukturu tog ulaza;
- `int f_find_entry(FCB* dir, char* name, FCB** fcb)`: pronalazi ulaz u direktorijumu datom prvim argumentom sa zadatim imenom (nazivom bez staze) datim drugim argumentom i u `*fcb` upisuje pokazivač na FCB strukturu tog ulaza;
- `int f_create_file(FCB** fcb)`: kreira novi fajl u fajl sistemu i u `*fcb` upisuje pokazivač na inicijalizovanu FCB strukturu tog kreiranog fajla;
- `int f_register_entry(FCB* dir, FCB* entry, char* name)`: u direktorijum dat prvim argumentom registruje ulaz dat drugim argumentom pod nazivom datim trećim arg.;
- `int f_remove_entry(FCB* dir, char* name)`: iz direktorijuma datog prvim argumentom izbacuje ulaz pod nazivom datim drugim argumentom (ne briše taj ulaz, već ga samo izbacuje iz direktorijuma, a taj ulaz ostaje kao objekat u fajl sistemu);
- `int f_delete_entry(FCB* entry)`: iz fajl sistema uklanja (briše, uništava) dati ulaz.

Sve ove operacije u slučaju uspeha vraćaju vrednost 0, a u slučaju greške ostavljaju sistem u konzistentnom stanju, kao da promena nije ni započeta i vraćaju kod greške koji je različit od 0. Korišćenjem ovih operacija implementirati sledeće operacije koje treba da vraćaju status na isti način:

- `int f_create_file(char* dname, char* fname)`: u direktorijumu sa punim imenom (sa stazom) zadatim prvim argumentom kreira novi fajl sa imenom zadatim drugim arg.;
- `int f_move_entry(char* sdname, char* fname, char* tdname)`: iz direktorijuma sa punim imenom zadatim prvim argumentom premešta ulaz sa imenom zadatim drugim argumentom u direktorijum sa punim imenom zadatim trećim argumentom.

Resenje:

```
int f_create_file(char* dname, char* fname) {
  int status = 0;
  FCB* dir = 0;
  status = f_find_entry(dname,&dir);
  if (status!=0) return status;
  FCB* file = 0;
  status = f_create_file(&file);
  if (status!=0) return status;
  status = f_register_entry(dir,file,fname);
  if (status!=0) f_delete_entry(file); //keep consistent state
  return status;
}
```

```
int f_move_entry(char* sdname, char* fname, char* tdname) {
  int status = 0;
  FCB* sdir = 0;
  status = f_find_entry(sdname,&sdir);
  if (status!=0) return status;
  FCB* tdir = 0;
  status = f_find_entry(tdname,&tdir);
  if (status!=0) return status;
  FCB* file = 0;
  status = f_find_entry(sdir,fname,&file);
  if (status!=0) return status;
  status = f_remove_entry(sdir,fname);
  if (status!=0) return status;
  status = f_register_entry(tdir,file,fname);
  if (status!=0) {
    f_register_entry(sdir,file,fname); //keep consistent state
  }
  return status;
}
```

Zadatak 18.

Upoređuju se sledeća tri načina alokacije blokova za sadržaj fajla:

A) Kontinualna alokacija, s tim da se za fajl odmah prilikom njegovog kreiranja alocira onoliko blokova kolika je maksimalna dozvoljena veličina fajla, iako ne moraju svi blokovi biti zauzeti sadržajem; u FCB je pokazivač na prvi blok i broj zauzetih blokova.

B) Ulančana alokacija, s tim da su u FCB pokazivači na prvi i poslednji blok sa sadržajem fajla, a pokazivači na sledeći blok u fajlu su na kraju svakog bloka sa sadržajem.

C) Indeksna alokacija, s tim da je u FCB pokazivač na (jedini) indeksni blok.

Za svaki od ovih načina alokacije posmatra se broj blokova koje je potrebno učitati u memoriju, pod sledećim pretpostavkama:

- svaka operacija posmatra se nezavisno, za isto početno stanje;
- početno stanje je takvo da je u memoriju učitan FCB i ni jedan drugi blok, osim onih koji su eksplicitno navedeni u operaciji.

Operacije su sledeće:

1) Direktan pristup n-tom bloku sa sadržajem u odnosu na početak fajla (nije učitan indeksni blok za način C).

2) Sekvencijalni pristup: pristup bloku n+1 nakon što je već učitan blok n (time je već učitan i indeksni blok za način C).

3) Proširenje sadržaja fajla (ispod maksimalne dozvoljene veličine fajla) slobodnim blokom koji je već alociran i učitan u memoriju (nije učitan indeksni blok za način C).

U donju tabelu upisati koliko blokova treba učitati za svaki od navedenih načina i svaku od navedenih operacija.

	A	B	C
1)	1	n	2
2)	1	1	1
3)	0	1	1

Zadatak 19.

Navedite bar jedan razlog zašto bi neki operativni sistem zahtevao ili bar omogućavao da korisnički proces prilikom kreiranja fajla eksplicitno navede da će taj fajl biti isključivo ili dominantno korišćen tako što će se sekvencijalno čitati.

Rešenje:

Ulančana alokacija blokova na disku za taj fajl i učitavanje unapred (*read-ahead*) su tehnike koje povećavaju efikasnost za ovakav pristup.

Zadatak 20.

Neki proces izvršava sistemski poziv za upis u fajl koji je prethodno uspešno otvorio i sistem mu to ne dozvoljava, sa porukom da tom procesu nije dozvoljena ta operacija, iako je korisniku u čije ime se taj proces izvršava dozvoljena operacija upisa u taj fajl. Objasnite zašto se ovo dogodilo.

Rešenje:

Pored prava pristupa prilikom izvršavanja neke operacije proverava se i u kom modu je otvoren fajl od strane procesa, tako da u ovom konkretno slučaju proces je otvorio fajl za čitanje, a pokušao je upis u njega.