Univerza *v Ljubljani*
Fakulteta *za računalništvo*
*in informatiko*

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko

Seminar

# Defragmentacija

Žan Magerl

Mentor: doc. dr. Tomaž Dobravec

19. januar 2020

# 1 Opis problema

Tekom shranjevanja na disku se komponenta operacijskega sistema za upravljanje z diskom trudi, da iste dele datoteke shranjuje na isti del diska. Operacijskemu sistemu je bolje, da ima datoteko shranjeno na strnjenem mestu in nima delcev te datoteke razdrobljene po celem disku.
Nagrada za tako dosledno shranjevanje se kaže v manjšem dostopnem času do datotek in posledično hitrejšemu delovanju računalnika, saj je disk mnogokrat ozko grlo računalniške arhitekture in je odgovoren za počasno delovanje računalnika.
Strnjene datoteke namreč omogočajo manjši dostopni čas. Večina časa trdi diski[1] namreč porabijo za fizično premikanje bralno-pisalne glave po disku in v primeru razdrobljenih datotek je tega premikanja mnogo več, kar posledično pomeni daljši dostopni čas.
Razdrobljenim datotekam drugače pravimo tudi fragmentirane datoteke, samemu procesu pa pravimo fragmentacija. Preprosto rečeno bi lahko dejali, da defragmentacija ni nič drugega kot sestavljanje drobcev datotek v večje celote.

[1] v vseh primerih govorimo o navadnih HDD diskih in ne novejših SSD diskih o katerih bomo nekaj več spregovorili kasneje

# 2 Fragmentacija

Kot nam že samo ime pove, je defragmentacija močno povezana s fragmentacijo in moramo zato če želimo dobro razumeti defragmentacijo, razumeti fragmentacijo in kako do nje sploh pride.

## 2.1 Vrste fragmentacije

Poznamo več različnih vrst fragmentacije datotečnega sistema. Na grobo jih delimo v 3 kategorije:

1. **Datotečna fragmentacija**
   To je vrsta fragmentacija na katero mislimo ko ponavadi govorimo o fragmentaciji. Fragmentacija datoteke je, ko je ena datoteka razbita na več fragmentov. Diskovni datotečni sistemi se seveda trudijo obdržati datoteko skupaj, a to vedno ni mogoče. Večina programske opreme za defragmenatcijo se ubada predvsem s to vrsto fragmnentacije in v nadaljevanju bomo tudi mi njej posvetili največ časa.

2. **Fragmentacija prostega prostora**
   V idealnem svetu je ves prosti prostor na disku v kontinuirani obliki, a v realnosti temu ni tako. Prosti prostor je fragmentiran kot posledica brisanja in zmanjševanja velikosti datotek. Ta prostor lahko uporabimo za majhne datoteke, če pa je premajhen pa ponavadi ostane neizkoriščen ali pa je zasedan z drobnimi fragmenti ostalih datotek.

3. **Fragmentacija povezanih datotek**
   Ta vrsta fragmentacije se nanaša na lokalnost povezanih datotek. Z drugimi besedami je to fragmentacija v smislu, da si datoteke, ki se pogosto uporabljajo sočasno, na disku niso blizu. Koncept časovne in prostorske lokalnosti je glavno gonilo uspešnosti predpomnilnika. Ker iskanje podatkov na disk traja dolgo, bi bilo optimalno, da se pogosto rabljene datoteke nahajajo skupaj in tako pohitrijo sam proces. Preprečavanje te vrste fragmentacije je najtežje, saj je treba narediti določene predpostavke, ki niso vedno resnične.

## 2.2 Vzroki

Fragmentacija sploh ne bi bila problem, če datotečni sistem oz. disk ne bi podpiral brisanja datotek in se obstoječe datoteke ne bi mogle povečevati. Če bi lahko datoteke zgolj dodajali in bi datoteke vedno ostale fiksne velikosti, bi lahko to počeli zaporedno in do lukenj med datotekami nikoli ne bi prišlo.

### 2.2.1 Brisanje

Problem se pojavi ko omogočimo brisanje datotek in med datotekami nastanejo luknje. Nato ima komponenta operacijskega sistema za upravljanje z diskom več različnih možnosti kako in kam bo shranila naslednjo datoteko:

1. Na prvo dovolj veliko mesto za celo datoteko (*first-fit algorithm*)
2. Na najmanjšo še dovolj veliko mesto za celo datoteko (*best-fit algorithm*)
3. Na največjo in dovolj veliko mesto za celo datoteko (*worst-fit algorithm*)

Zgornje možnosti pridejo v poštev samo če obstaja luknja, ki je dovolj velika za celo datoteko. V nasprotnem primeru moramo datoteko razdeliti na več delcev (fragmentacija) in delce datoteke vstavljati po enemu izmed zgornjih algoritmov.

### 2.2.2 Razširanje

Podoben primer se pojavi, če omogočamo da se datoteke širijo in krajšajo[2]. Ta problem je še posebej pereč pri datotekah, ki se pogosto spreminjajo: logi, datoteke o tekočih procesih ipd. V interesu komponente operacijskega sistema za upravljanje z diskom je, da se datoteka zaporedno nadaljno širi. A to seveda ni vedno mogoče, saj se lahko zgodi, da se takoj za datoteko ki se želi širiti nahaja druga datoteka in ne prazen prostor. Tu se vidi dvoreznost meča, če komponenta za upravljanje diska datoteke neprekinjeno zlaga zaporedno in med njimi ne pusti prostora, da optimizira izrabljenost prostora. Ta problem lahko omilimo tako, da komponenta za upravljanje diska loči datoteke za katere presodi, da se bodo pogosto širile, od tistih za katere misli, da bodo obržale konstantno dolžino. A to še zdaleč ni enostavno.
Če se torej neka datoteka razširi (npr. program doda novo vrstico v log datoteko) in datoteka za seboj nima prostora da se razširi, je naloga komponente operacijskega sistema za upravljanje diska, da prostor zagotovi. Na voljo ima dve opciji:

1. Datoteko premakne na novo lokacijo, kjer bo dovolj prostora za predvideno širitev
2. Nov dodani del datoteke kot delec zapiše nekam drugam na disk

Obe metodi imata svoje prednosti in slabosti. Premikanje datoteke na novo lokacijo je dokaj časovno potratno, saj je treba po disku iskati nov dovolj velik prostor, poleg tega nimamo garancije koliko časa bo preteklo preden se bo datoteka znova razširila, za seboj pa zopet pustimo luknjo. Fragmentacija datoteke je hitrejši proces, saj so dodani fragmenti ponavadi manjhni in lahko prostor zanje najdemo po enem izmed zgoraj omenenjih algoritmov. Slaba stran je seveda da se lahko zgodi, da dobimo po mnogih širitvah datoteke močno fragmentirano datoteko, do česar ne pride če uporabljamo prvo metodo.

## 2.3 Datotečni sistem

Da lahko sploh bolje razumemo kako fragmentacija in defragmentacija res delujeta, moramo poznati vsaj osnove datotečnih sistemov. Datotečni sistem organizira in nadzoruje kako se podatki shranjujejo na disk. Brez datotečnega

[2]kar je seveda stalnica na sodobnih operacijskih sistemih

sistema bi bil disk zgolj zaporedje 0 in 1 brez da bi kdorkoli vedel kam te 0 in 1 spadajo. Datoteka je osnovna enota skupine podatkov in datotečni sistem skbi za upravljanje z njimi. Da lahko to počne dobro in dosledno, vsaki datoteki pripne tako imenovane metapodatke, ki povedo pomembne podatki o sami datoteki: ime, velikost, lokacijo, ID-je, dovoljenja, tipe ipd. Kam in kako se ti podatki shranijo je odvisno od samega datotečnega sistema. Unix-ove različice datotečnih sistemov uporabljajo posebno podatkovno strukturo imenovano *inode* v katero shranjujejo večino metapodatkov, ostale pa shranijo v direktorijsko tabelo direktorija v katerem se datoteka nahaja.

### 2.3.1 Terminologija

Na področju datotečnih sistemov in diskov se je skozi zgodovino pojavilo mnogo pojmov, ki jih je smotrno definirati:

- Sector (*angl. sector*): sektor je najmanjša enota diska, saj predstavlja del tračnice na disku. Če datoteka ne zapolni celega sektorja, je preostanek sektorja tipično zapolnjen z ničlami. Tradicionalno je sektor velik okoli 512 B, novejši format *AF* pa podpira sektorje velike 4 kiB (4096 B).
- Blok (*angl. block*): blok je zaporedje bitov oz. bajtov in z njimi za razliko od sektorjev upravlja datotečni sistem. Operacijski sistem tako ponavadi namesto nad sektorji, raje operira nad bloki, ki se tipično raztezajo čez več sektorjev. Bloki imajo ponavadi fiksno maksimalno dolžino, ki je odvisna od nastavitev datotečnega sistema. Napravam (npr. diski), ki uporabljajo bloke, pravimo tudi bločno-orientirane naprave.
- Ekstent (*angl. extent*): ekstent je kontinuiran prostor za datoteke v datotečnem sistemu. Predstavljen je s številko prvega bloka in številko zadnjega bloka. Datoteka lahko sestoji iz več ekstentov in v primeru fragmentirane datoteke vsak fragment potrebuje svoj ekstent.

### 2.3.2 Vrste datotečnih sistemov

Tekom zgodovine so se z razvojem različnih operacijskih sistemov razvijali tudi različni datotečni sistemi. Vsak izmed glavnih operacijskih sistemov je enega izmed formatov vzel za svojega in morajo biti posledično diski na katerih je naložen operacijski sistem formatirani s primernim datotočenim sistemom.

- Linux: ext4 je standardni datotečni sistem za številne Linux distribucije.
- Windows: NTFS je standardni datotečni sistem za operacijski sistem Windows. Sam datotečni sistem je nadgradnja "zastarelega" sistema FAT.
- MacOS: APFS je standardni datotečni sistem za MacOS razvit strani Apple-a. Sloni na svojem predniku HFS+ s fokusom na SSD diskih.

V naslednjem poglavju, ko se bomo spustili podrobneje v defragmentacijo, bomo za model datotečnega sistema vzeli ext4. Razlog temu je, da je prisoten na večini sodobnih Linux distribucijah in implementira številne koncepte, ki smo jih že omenili tekom seminarja. Prav tako je zaradi odprtokodnosti številnih programov, ki so napisani za operacijski sistem Linux možno podrobneje pogledati v ozadje raznih programov, ki se ukvarjajo z iskanjem fragmentiranih datotek.

# 3 Defragmentacija

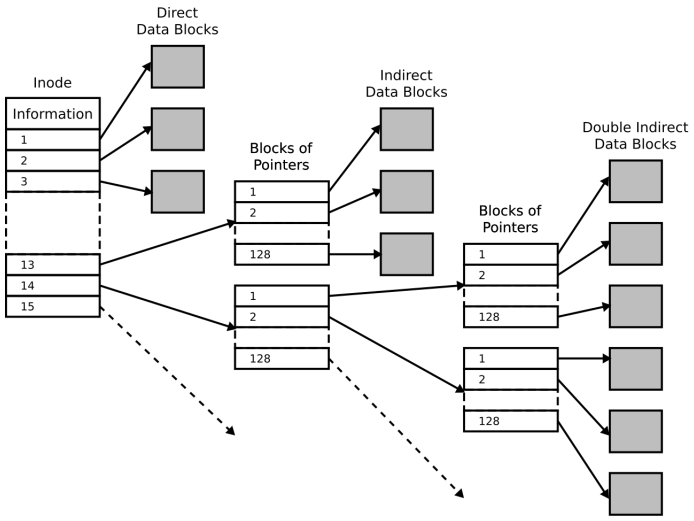
Kot smo že izvedeli, imajo številni novejši datotečni sistemi podprte razne mehanizme za preprečavanje pretirane fragmentacije datotek in diska. Čeprav bolj pogosto pred leti kot danes, je fragmentacija še vedno problem. Takrat pride prav defragmentacija, ki želi odpraviti datotečno fragmentacijo in spraviti datoteke v čim bolj kontinuirano obliko.

## 3.1 Iskanje fragmentiranih datotek

Če želimo defragmentirati fragmentirane datoteke je treba fragmentirane datoteke najprej najti. Sam postopek iskanja je seveda odvisen od vrste datotečnega sistema, ki ga operacijski sistem uporablja. Za lažjo razumevanje se bomo uprli na Linux-ov datotečni sistem ext(4), seveda pa bodo osnovni principi enaki na vseh sistemih.

### 3.1.1 Lokacije datotek

Kot smo že omenili v prejšnjem poglavju, so številni pomembni podatki o datotekah zapisani v metapodatkih same datoteke. V inode-ih se nahaja del teh podatkov, poleg tega pa inode vsebuje tudi kazalce na lokacije blokov kjer se podatki dejansko nahajajo. Tabela vsebuje 15 kazalcev, prvih 12 kaže direktno




Slika 1: Struktura inode kazalcev

na bloke, kjer se podatki nahajajo. To zadostuje za večino datotek, za večje datoteke pa so na voljo kazalci 13, 14 in 15, ki vsak omogočajo eno stopnjo večje indirektnosti. 13. kazalec tako kaže na tabelo novih kazalcev na bloke, 14. kazalec kaže na tabelo, v katerih so kazalci na tabele, ki vsebujejo kazalce na bloke, medtem ko 15. to počne s še enim indirektnim nivojem več.

S pomočjo teh kazalcev lahko tako pridobimo lokacijo blokov, ki sestavljajo samo datoteko. Da ugotovimo, ali je datoteka fragmentirana moramo tako samo še pogledati naslove teh blokov in hitro vidimo če je datoteka fragmentirana ali ne, saj so bloki v primeru nefragmentirane datoteke zaporedni in ni nobenega skakanja v naslovih.
Datotečni sistem ext4 je dodal spremembe v inode strukturo in kazalci ne kažejo več na datotečne bloke, ampak prav na ekstente, kar izboljša prostorsko učinkovitost shrambe inode-ov in olajša izvedbo defragmentacije, saj ni potrebno pregledovati skakanj v naslovih.
Na Linux-u obstajajo že vgrajeni (*built-in*) programi, ki zgoraj opisan postopek naredijo za nas in nam rezultat lepo formatirajo. Eden izmed takih programov je `filefrag`, ki omogoča kopico različnih zastavic, ki regulirajo kaj želimo od programa izvedet.
Enega izmed možnih načinov uporabe, lahko vidimo na naslednji sliki:

```
[zanmagerl@polhec test]$ filefrag -v large_file.mkv
Filesystem type is: ef53
File size of large_file.mkv is 1062965839 (259514 blocks of 4096 bytes)
 ext:     logical_offset:        physical_offset: length:   expected: flags:
   0:        0..    2047:    9189376..   9191423:   2048:
   1:     2048..   34815:    9193472..   9226239:  32768:    9191424:
   2:    34816..   67583:    9226240..   9259007:  32768:
   3:    67584..  100351:    9259008..   9291775:  32768:
   4:   100352..  133119:    9291776..   9324543:  32768:
   5:   133120..  165887:    9324544..   9357311:  32768:
   6:   165888..  198655:    9357312..   9390079:  32768:
   7:   198656..  231423:    9390080..   9422847:  32768:
   8:   231424..  245759:    9422848..   9437183:  14336:
   9:   245760..  247807:    9648128..   9650175:   2048:    9437184:
  10:   247808..  259513:    9658368..   9670073:  11706:    9650176: last,eof
large_file.mkv: 4 extents found
```

Slika 2: Seznam ekstentov datoteke velike približno 1 GB

Iz slike lahko razberemo več različnih stvari glede lastnosti diska in datotečnega sistema, kjer smo ta ukaz izvedli:

- Datotečni sistem je eden iz družine ext (ext2, ext3 ali ext4), saj je ef53 tako imenovan "magic number" ext datotečnih sistemov
- Velikost bloka tega datotečnega sistem je 4096 B[3]
- Čeprav piše da je datoteka velika 0,989964 GiB, v resnici zasede 0,989967 GiB prostora, saj je disk razdeljen na bloke in niso vsi bloki polno zapolnjeni. Razlika je natančno 3505 B, kar je posledica notranje fragmentacije bloka.

[3]novejši datotečni sistemi v večini podpirajo 4096 B velike bloke, namesto tradicionalnih 512 B

- V stolpcu `expected` lahko vidimo kakšen bi moral biti odmik, če datoteka ne bi bila fragmentirana
- Zadnji ekstent vsebuje zastavico `last,eof` ki data vedeti da je to zadnji ekstent te datoteke

Program `filefrag` nam torej sam najde ekstente za želeno datoteko. Da se pa bolje prepričamo v to, da zares razumemo kako datotečni sistem upravlja s samimi datotekami, je bolje če s parimi ukazi poskusimo najti fizično lokacijo datoteke na disku zgolj s pomočjo inode-a.

```
[zanmagerl@polhec test]$ stat a.txt
  File: a.txt
  Size: 47              Blocks: 8          IO Block: 4096   regular file
Device: fd02h/64770d    Inode: 3688594     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/zanmagerl)   Gid: ( 1000/zanmagerl)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2020-01-12 13:41:58.341837789 +0100
Modify: 2020-01-12 13:33:04.186149731 +0100
Change: 2020-01-12 13:33:04.193149760 +0100
 Birth: 2020-01-12 13:33:04.186149731 +0100
[zanmagerl@polhec test]$ sudo debugfs -R "blocks <3688594>" /dev/mapper/fedora-home
debugfs 1.44.6 (5-Mar-2019)
9124392
[zanmagerl@polhec test]$ address=$((9124392*4096/512))
[zanmagerl@polhec test]$ sudo dd if=/dev/mapper/fedora-home bs=512 skip=$address count=1 status=none | hexdump -C
00000000  50 6f 7a 64 72 61 76 6c  6a 65 6e 20 73 70 72 65  |Pozdravljen spre|
00000010  68 61 6a 61 6c 65 63 20  70 6f 20 64 61 74 6f 74  |hajalec po datot|
00000020  65 63 6e 65 6d 20 73 69  73 74 65 6d 75 21 0a 00  |ecnem sistemu!..|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200
[zanmagerl@polhec test]$ cat a.txt
Pozdravljen sprehajalec po datotecnem sistemu!
```

Slika 3: Prikaz iskanja datoteke zgolj z inode-om datoteke

Najprej z ukazom `stat` izpišemo metapodatke datoteke in iz njih razberemo številko *inode*, ki je v tem primeru `3688594`. S pomočjo orodja `debugfs` pridobimo številke blokov, katere naša datoteka zaseda. Ker je naša datoteka majhna, saj vsebuje samo en stavek, datoteka zaseda zgolj blok `9124392`. Nato opravimo preprosto kalkulacijo, da pridobimo pravi odmik in dobljena cifra nam predstavlja začetek datoteke. Nato s pomočjo ukaza `dd` in programa `hexdump` izpišemo vsebino bloka in dobimo pravo vsebino datoteke, kar potrdimo z ukazom `cat`.

## 3.2 Delovanje defragmentacije

Ko smo enkrat našli fragmentirane datoteke, jih moremo še defragmentirati. Pri teim imamo več možnosti, ali defragmentiramo eno samo datoteko, ali pa kar cel del diska. Pri defragmentaciji ene datoteke je postopek trivialen in ga bomo zgolj besedno opisali. Pri defragmentaciji diska, pa se stvari zapletejo. Pogledali si bomo 3 osnovne algoritme za pristop k defragramentaciji dela diska. Ti algoritmi predstavljajo zgolj koncepte in nikakor ne odražajo dejanske defragmentacije implementirane v sodobnih sistemih. Več o tem pa po predstavitvi samih algoritmov.

### 3.2.1 Defragmentacija ene datoteke

Do problema fragmentacije datoteke je možno pristopati na razne načine. Eden izmed bolj osnovnih postopkov kako defragmentirati določeno datoteko je sledeč:

1. Pregledamo disk in si shranimo velikosti in lokacije prostih prostorov v obliki teric: (lokacija, velikost)
2. Poiščemo datoteko, ki jo želimo defragmentirati
3. Pregledamo število fragmentov iz katerih je datoteka sestavljena in se odločimo ali lahko datoteko spravimo v eno izmed prej najdenih lokacij prostega prostora
4. Če se to da, datoteko enostavno premaknemo na novo mesto. To storimo tako da v pravilnem vrstnem redu zlagamo fragmente datotek enega za drugim na prosto mesto.

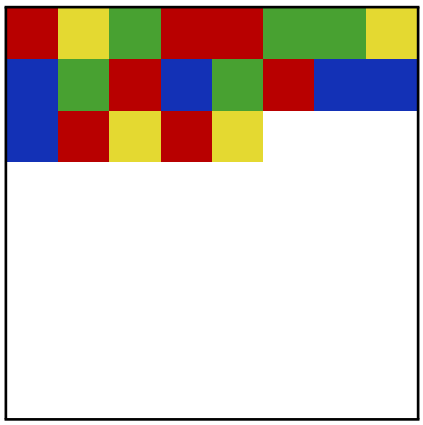
Podoben postopek implementira ena izmed rešitev za defragmentacijo izbranih datotek za operacijski sistem Windows. Za morebitne podrobnosti se povezava nahaja med viri.

### 3.2.2 Algoritmi za defragmentacijo dela diska

Prvi naivnejši pristop bo z uporabo večjega kosa praznega diska, ki bo deloval kot začasno odlagališče. Drugi pristop bo bil časovno počasnejši kot prvi algoritem, a bo porabil manj diska. Tretji pristop se bo posluževal filozofije sortiranja s štetjem, kar bomo poiskusili implementirati s čim manj porabe zunanjega prostora in bo združeval prednosti obeh prejšnih enostavnejših pristopov. Idealno bi bilo, da bi ga lahko implementirali *in-place*, čemur se bomo poskušali čim bolj približati.
Za lažjo ilustracijo bomo za oba primera imeli del diska, ki vsebuje 64 blokov, na katerem se nahaja 4 različne datoteke. Vsaka izmed teh datotek je močno fragmentirana in njihovi fragmenti skupaj zasedajo 23 blokov, fragmenti pa so naraščejoče urejeni. Preostanek diska lahko uporabimo za namene defragmentacije[4].Predpostavili bomo tudi, da fragmenti med seboj niso povezani s kazalci ampak so neodvisne enote na disku, saj se pri kazalcih iskanje prostora in menjavanje nekoliko zaplete in bi otežili samo ilustracijo.
Naš cilj je da datoteke defragmentiramo in jih zložimo zaporedno, brez vmesnega spuščanja prostora[5].

[4]to je seveda zelo velika predpostavka in skoraj zagotovo defragmenterji v praksi nimajo takih lepih okoliščin

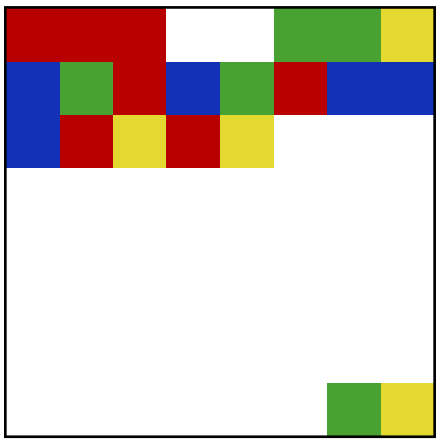
[5]kar je seveda daleč od optimalnosti, zaradi razlogov, ki smo jih že omenili v prejšnjih poglavjih



Slika 4: Začetno stanje na disku

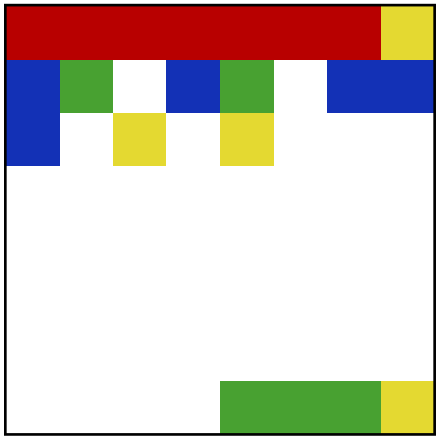
**Naivnejši pristop z veliko porabo prostora**

Iz opisa omejitev problema, lahko uvidimo, da si lahko privoščimo veliko prostora za premikanje blokov datotek po disku, kar bo ta algoritem poskušal izkoristiti.
Sprehodimo se skozi fragmentiran del diska in v $i$-ti iteraciji obhoda bo naš cilj, da defragmentiramo $i$-to datoteko. Ko pridemo do bloka, ki ne spada k naši datoteki, ga preskočimo in nadaljujemo iskanje po disku. Ko pridemo do bloka, ki spada k $i$-ti datoteki, in se za našo do sedaj defragmentirano datoteka nahaja nek drugi neprazen blok, blok odložimo na odlagališče in na njegovo mesto postavimo najdeni blok. V primeru praznega prostora pa blok samo prekopiramo na prazno mesto.



Slika 5: Stanje diska po dveh zamenjavah

Ta postopek nadaljujemo dokler ni naša datoteka v celoti defragmentirana.



Slika 6: Stanje diska po koncu prvega obhoda

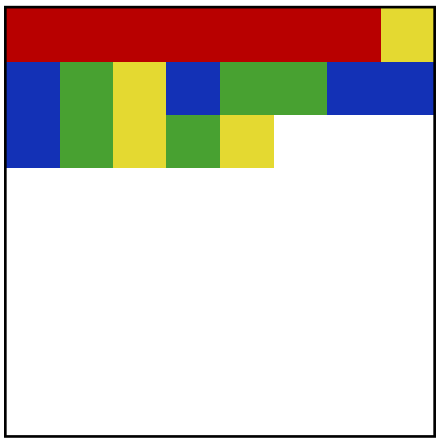
Nato ves postopek ponovimo, le da poleg iskanja fragmentov na fragmentiranemu delu diska, datoteke iščemo tudi po odlagališču. Sedaj lahko še ocenimo časovno in prostorsko zahtevnost algoritma, pri čemer $n$ označuje število blokov in $k$ število datotek:

1. Časovna zahtevnost: $\mathcal{O}(k \cdot (n))$[6] $= \mathcal{O}(k \cdot n)$
2. Prostorska zahtevnost: $\mathcal{O}(n)$[7]

### Naivnejši pristop z manjšo porabo prostora

Sprehodimo se skozi fragmentiran del diska in v $i$-ti iteraciji obhoda bo naš cilj, da defragmentiramo $i$-to datoteko. Na začetku imamo tako samo prvi fragment datoteke, nato se sprehajamo po disku in iščemo naslednji fragment te datoteke. Ko najdemo naslednji blok te datoteke ga zamenjamo z blokom, ki je takoj za zadnjim blokom datoteke. Postopek ponavljamo dokler ne pridemo do konca diska in smo zaporedno zložili vse fragmente prve datoteke.

[6] imamo $k$ obhodov in vsakič moramo v najslabšem primeru obhoditi vse bloke
[7] odlagališče je v najslabšem primeru dolgo $n$ blokov



Slika 7: Stanje diska po 1. obhodu algoritma

Nato se postavimo na blok takoj za prvo datoteko in zgornji postopek ponovimo za to datoteko. Enako storimo za vse preostale datoteke. Ko pridemo do konca imamo vse datoteke defragmentirane, a bloki znotraj datotek niso nujno v pravem vrstnem redu. Z zamenjavami smo lahko vrstni red porušili, saj smo lahko storili zamenjavo, ki je prvi blok neke druge datoteke prestavilo za nek drugi blok iste datoteke. Zato moramo vsako datoteko sedaj še notranje sortirati, kar lahko storimo s poljubnim algoritmom za sortiranje: *quicksort*, *insertion sort* ali celo *counting sort*.
Sedaj lahko še ocenimo časovno in prostorsko zahtevnost algoritma, pri čemer $n$ označuje število blokov in $k$ število datotek:

1. Časovna zahtevnost: $\mathcal{O}(k \cdot n + k \cdot \frac{n}{k} \log \frac{n}{k})$[8] $= \mathcal{O}(k \cdot n + n \log \frac{n}{k})$
2. Prostorska zahtevnost: $\mathcal{O}(1)$[9]

### Naprednejši pristop

Pri tem pristopu se bomo poslužili ideje sortiranja s štetjem (*angl. counting sort*). Ideja je, da se prvič sprehodimo čez fragmentiran disk in seštejemo število fragmentov vsake datoteke. Te fragmente nato pretopimo v indekse, kar storimo tako, da se sprehodimo skozi tabelo števila fragmentov datotek in sproti seštevamo števila fragmentov in pri $i$-ti datoteki vpišemo $i$-to vsoto po formuli:

$$sum(i) = sum(i-1) + fragments(i-1)$$

Za naš konkretni primer, bi tabela zgledala takole:

[8] imamo $k$ obhodov in vsakič moramo v najslabšem primeru obhoditi vse bloke, poleg tega moramo nato vsako od $k$ datotek še notranje sortirati

[9] ker izvajamo zgolj izmenjave, na odlagališče vedno odložimo samo 1 blok

| | Rdeča | Rumena | Zelena | Modra |
|---|---|---|---|---|
| vsota | 7 | 4 | 5 | 5 |
| indeks | 0 | 7 | 11 | 16 |

Ko enkrat imamo indekse, se sprehodimo po fragmentiranemu disku in vsakemu bloku priredimo ustrezen indeks, ki je enak vsoti indeksa v tabeli in številu že indeksiranih blokov te datoteke. Sedaj nas čaka samo še urejanje datoteke glede na pripadajoče indekse, kar se da narediti precej hitro. Sprehodimo se skozi disk, če je na $i$-tem mestu blok z $i$-tim indeksom, potem jo preskočimo, če ne, blok premaknemo na pravo lokacijo, blok na tisti lokaciji pa premaknemo na $i$-to mesto. Če je ta blok pravi, torej je njegov indeks enak $i$, gremo naprej, če ne blok zopet zamenjamo. To storimo z vsemi bloki na disku. Čar take implementacije je v tem, da medtem ko "iščemo" $i$-ti blok, hkrati že postavljamo ostale bloke na prava mesta. Sedaj lahko še ocenimo časovno in prostorsko zahtevnost algoritma, pri čemer $n$ označuje število blokov in $k$ število datotek:

1. Časovna zahtevnost: $\mathcal{O}(n+k+k+n)$[10] $= \mathcal{O}(n+k)$
2. Prostorska zahtevnost: $\mathcal{O}(1)$[11]

Za boljšo ilustracijo se tukaj nahaja izhod programa v C-ju, ki izvaja ta algoritem:

```
[polhec@localhost C]$ ./a.out
Data before being sorted:
1 2 3 1 1 3 3 2 4 3 1 4 3 1 4 4 4 1 2 1 2

Indexes for fragments:
0 7 11 1 2 12 13 8 16 14 3 17 15 4 18 19 20 5 9 6 10

Number of swaps during sorting: 17

Data after being sorted in in-place fashion:
1 1 1 1 1 1 1 2 2 2 2 3 3 3 3 3 4 4 4 4 4
```

Slika 8: Izhod programa

### 3.2.3 Implementacija v OS

Zgornji algoritmi so zgolj koncepti, njihov namen je zgolj ilustrirati probleme na katere naletimo pri defragmentaciji in časovne in prostorske zahtevnosti same

[10] kreiranje indeksov zahteva obhod preko vseh $n$ blokov, $n$ operacij prav tako zahteva končno sortiranje,$k$-ji so posledica manipulacije s tabelo

[11] ker izvajamo zgolj izmenjave, na odlagališče vedno odložimo samo 1 blok

defragmentacije. V teoriji se zadnji pristop, ki se poslužuje učinkovitega sortiranja s štetjem, zdi dober, a v praksi ima veliko problemov.
Prva opomba zgornjim algortmom je, da smo ocenili zgolj asimptotične časovne in prostorske zahtevnosti, kjer smo konstante zanemarili. To je v teoriji seveda v redu, vendar se v praksi izkaže, da je razlika ali blok samo enkrat premaknemo ali pa ga 100-krat.
Zavedati se moramo, da so bloki v resnici velike enote in je samo prepisovanje blokov časovno zahtevno. Veliko je seveda odvisno od tega kako je sam hardware zasnovan, a osnovno dejstvo ostane isto. Če bi sistem deloval tako, kot zgornji algoritmi, torej da bi prepisaval blok po blok bi to trajalo veliko časa, saj največ časa traja premikanje bralno-pisalne glave po disku. In ker so datoteke sestavljene iz mnogo blokov (v 1 GB veliki datoteki jih je recimo okoli 260.000) bi bila defragmentacija še mnogo počasnejši proces kot v resnici je.
Ideja torej je, da namesto da prepisujemo vsak blok posebej, raje prepisujemo cele fragmente skupaj. Disk bo tako namesto premikanja na lokacijo bloka, branja bloka, premikanja na odlagališče in pisanja na odlagališče, to izvedel za cel fragment skupaj in ne za vsak blok posebej[12]. To rezultira v mnogo manj premikanj bralno-pisalne glave in posledično pomeni hitrejše delovanje.
Poleg tega iskanje po disku ni implementirano z linearnim iskanjem po celem disku, saj bi to trajalo zelo dolgo časa. Kako so shranjenje lokacije blokov posamične datoteke se razlikuje od datotečnega sistema do datotečnega sistema, a v datotečnem sistemu ext4 so ti podatki shranjeni v inode-u datoteke. Iskanje fragmentov je tako potem implementirano z rekurzivnim premikanjem po drevesu kazalcev. To, da vemo kje so fragmenti je ena stvar, da pa vemo kam jih lahko damo, torej če se za njimi nahaja prazen prostor je pa povsem druga zgodba. Disk namreč na prazni prostor ne glede kot na datoteko in tako ni shranjen v inode podatkovni strukturi. Namesto tega za prazni prostor pogleda v bitno tabelo prostega prostora, kjer 1 označuje zaseden prostor in 0 prazen. Ko najde prazen prostor se vključi tako imenovan *buddy allocator*, ki za podan blok poišče ekstent zahtevane dolžine. Če je to mogoče, ga vrne in alokacija fragmenta na tisti ekstent je mogoče, če pa ne, pa je treba premakniti datoteko, ki se nahaja na tistem območju.

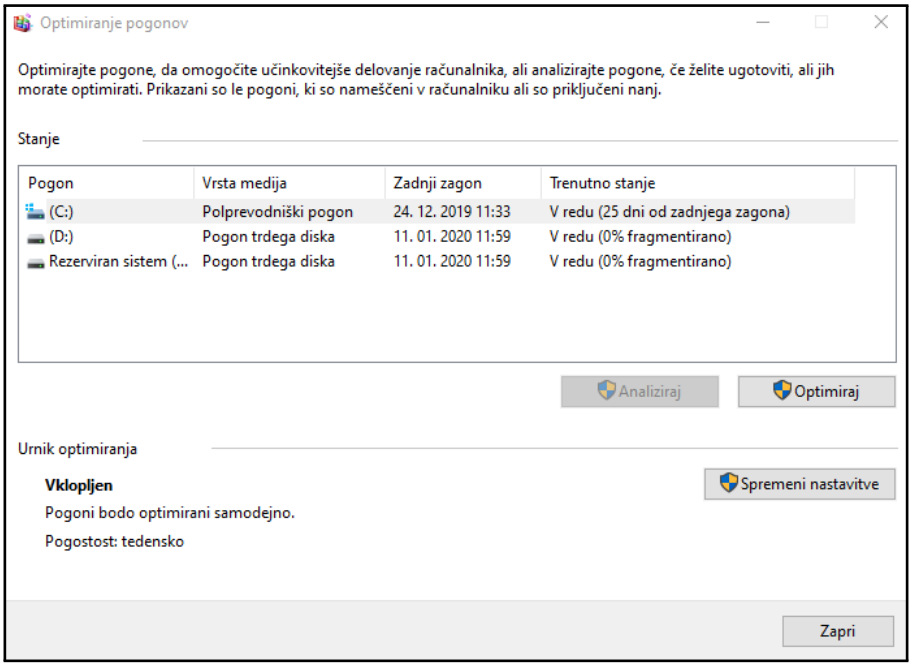
## 3.3 Preprečevanje fragmentacije

Zavedati se moramo, da so tekom razvoja postajali datotečni sistemi vedno bolj in bolj inteligentni glede preprečevanja fragmentacije in s tem omejevanja potrebe po defragmentaciji. Datotečni sistem ext4, ki ga najpogosteje uporabljajo najpopularnejše Linux distribucije, je namreč izredno dober v preprečevanju fragmentacije in defragmentacije v veliki večini primerov sploh ne potrebuje. Razlog temu je, da se datotečni sistem trudi in datoteke shranjuje razpršeno in imajo tako posledično več prostora, da se širijo in spreminjajo. Če se fragmentacija vseeno pojavi, ponavadi že sam datotečni sistem poskrbi za zmanjševanje fragmentacije, ne da bi za to potreboval poseben program za defragmentacijo.

[12] za medhranjenje bo uporabil buffer, ki je običajno velik 8-256 MiB

## 3.4 Programi za defragmentacijo na OS

Tekom let se je na trgu pojavila množica orodij za defragmentacijo, ki so delovala na različnih operacijskih in datotečnih sistemih. S prehodom na novejše tehnologije je nekaj od teh programov tudi postalo neuporabnih. Večini je skupno to, da zelo skrivajo, kako točno delujejo in žal ne moremo vedeti kakšne pristope uporabljajo za defragmentacijo.
Privzeti program za defragmentacijo in optimizacijo diskov na Windowsu se imenuje: "Defragmentiraj in optimiraj pogone", ki omogoča pregled nad diski v sistemu in kontrolo njihovega zdravja.




Slika 9: Nadzorna plošča programa

Kot lahko razvidimo iz zgornje slike Windows sam poskrbi za defragmentacijo in optimizacijo diskov z namenom da razbremeni navadnega uporabnika[13]. Kot smo že povedali v prejšnjem poglavju, Linux s svojim datotečnim sistemom ext4 ne potrebuje posebnega orodja za defragmentacijo. Podobno zgodba je pri MacOS (10.2+), ki ima prav tako že svoje vgrajene mehanizme za preprečevanje fragmentacije in defragmentacijo, ko je to potrebno. Kljub temu še vedno na trgu obstaja kopica programov za defragmentacijo diskov, ki pa se specializirajo na robnejše primere, ki se ne zadevajo tipičnih uporabnikov. Taki programi tako poskrbijo za defragmentacijo zelo polnih diskov (nad 90 % zasedenosti) ali

[13]kot lahko vidimo Windows defragmentira tudi SSD diske (polprevodniški pogon), več o tem pa v naslednjem poglavju

pa sam postopek izvajajo bolj efektivno in še bolj dosledno in varno, kar je potrebno na občutljivih sistemih (banke, zavarovalnice, ...).

## 3.5 Defragmentacija in SSD

Kot smo že večkrat povedali je "počasnost" bralno-pisalne glave glavni razlog zakaj nočemo fragmentiranih datotek. SSD diski pa niso bazirani na tehnologiji navadnih trdih diskov (HDD), ampak temeljijo na bliskovni (*angl. flash*) tehnologiji, ki ne vsebuje premikajočih se mehanskih delov kot so bralno-pisalna glava. Bliskovna tehnologija namreč omogoča prost (*angl. random-access*) dostop do podatkov na disku. To pomeni, da pri fragmentiranih datotekah ne pride do latence, ki bi bila posledica premikanja bralno-pisalne glave. Še več, ker so SSD-ji bazirani na bliskovni tehnologiji, imajo omejeno število pisalnih operacij preden odpovejo. Zaradi tega je pogosta defragmentacija SSD diskov celo močno odsvetovana, saj krajša življensko dobo diska.
A zgodba ni tako enolična in preprosta. Čeprav je res, da je defragmentacija odsvetovana, saj disk nima mehanskih delov, pri tem pa skrajšuje življensko dobo diska, v resnici močno fragmentirani SSD diski ne morejo več izvajati pisalnih operacij na disku. Razlog temu so metapodatki datotečnega sistema, saj datotečni sistem lahko pomni in nadzira le določeno število metapodatkov in če fragmentacija preseže maksimalni prag fragmentiranosti disk ne more več opravljati svojega dela. Ker večja fragmentiranost datotek na disku pomeni več metapodatkov, to tudi posledično vpliva na hitrost bralno-pisalnih operacij nad posamezno datoteko. Učinek seveda ni tako velik kot pri navadnih HDD diskih, a v teoriji je ta faktor hitrosti vseeno prisoten. Zaradi teh razlogov privzeti program za optimiziranje diskov na operacijskem sistemu Windows enkrat mesečno "optimizira" SSD diske z namenom, da podaljša njihovo življensko dobo in maksimizira njihovo učinkovitost.
Zgradba in delovanje SSD-ja imajo še svoje specifike, a te presagajo namene tega poglavja.

# 4 Zaključek

Trdi diski so že sami po sebi pogosto ozko grlo računalniškega sistema. Močno fragmentirani trdi diski vso stvar še močno poslabšajo. Pregledali smo problematiko fragmentacije in orodje za odpravite le nje: defragmentacijo. Ob tem smo si pogledali tudi osnove datotečnih sistemov in se malo globje spustili tudi v implementacijo datotečnega sistema ext4. Problem fragmentacije je bil še bolj pereč v časih pred sodobnimi datotečnimi sistemi in pred pojavom SSD diskov. Danes veliko orodij skrbi za zdravje diskov brez našega zavedanja, a to ni razlog da se ne bi problematike zavedali. Zavedanje o delovanju sistemov s katerimi vsakodnevno operiramo je pomembno, če želimo samemu področju tudi karkoli prispevati.

# 5 Viri in literatura

A. B. Downey, *Think OS: A Brief Introduction to Operating Systems*, Massachusetts: Green Tea Press, 2015

An Introduction to Linux's EXT4 File System. *Opensource*[Online]. Dosegljivo: https://opensource.com/article/17/5/introduction-ext4-filesystem (zadnji obisk 18. 1. 2020)

A Minimum Complete Tutorial of Linux ext4 File System. *Mete Balci*[Online]. Dosegljivo: https://metebalci.com/blog/a-minimum-complete-tutorial-of-linux-ext4-file-system (zadnji obisk 18. 1. 2020)

Contig. *Microsoft Docs*[Online]. Dosegljivo: https://docs.microsoft.com/en-us/sysinternals/downloads/contig (zadnji obisk 18. 1. 2020)

D. Irtegov, "Operating System Fundamentals", *Charles River Media*, str. 363-408, 2002

Defragmentation. *Wikipedia*[Online]. Dosegljivo: https://en.wikipedia.org/wiki/Defragmentation (zadnji obisk 18. 1. 2020)

Filefrag. *Linux man page*[Online]. Dosegljivo: https://linux.die.net/man/8/filefrag (zadnji obisk 18. 1. 2020)

Fragmentation. *Wikipedia*[Online]. Dosegljivo: https://en.wikipedia.org/wiki/Fragmentation (zadnji obisk 18. 1. 2020)

S. Kadekodi in S. Jain, "Taking Linux Filesystems to the Space Age: Space Maps in Ext4", v zborniku *Linux Symposium*, Ottawa, Canada, julij 2010