SVEUČILIŠTE U ZAGREBU

**FAKULTET ELETROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI ZADATAK br. 1716

# Statistička analiza algoritama za dinamičko upravljanje spremnikom

Nikola Sekulić

Zagreb, lipanj 2011.

## Sadržaj:

1. Uvod ... 1
2. Osnovni algoritmi upravljanja spremnikom ... 2
   2.1. Upravljanje spremnikom pomoću slijedno povezanih lista ... 2
   2.2. Upravljanje spremnikom pomoću odvojenih lista slobodnih blokova ... 5
   2.3. Ostali načini upravljanja spremnikom ... 7
3. Ostvarenje analize algoritama za dinamičku dodjelu spremnika ... 8
   3.1. Prikupljanje podataka ... 8
   3.2. Analiza podataka ... 10
4. Rezultati analize ... 12
5. Zaključak ... 17
6. Literatura ... 18
7. Sažetak ... 19
8. Privitak ... 21

# 1. Uvod

Radna memorija je jedna od glavnih komponenti računala. U njoj se nalaze podaci koje pojedini proces obrađuje. U proteklih nekoliko godina, brzina procesora se jako dobro razvijala, dok s druge strane brzina pristupa memoriji je ostala skoro konstantna. Radi toga se za manipulaciju memorijom moraju koristiti što efikasniji algoritmi. Operacijski sustav se brine o dodjeljivanju odgovarajućeg dijela memorije pojedinom procesu. Dio memorije koji je proces dobio od operacijskog sustava sastoji se od statičnog, automatskog i dinamičkog dijela. U statičnom dijelu se nalaze podaci programa čija se veličina zna prije izvođenja. Automatski dio služi za izvođenje funkcija i procedura programa, te je logički ostvaren pomoću stoga. Dinamički dio memorije služi za podatke programa, čije se veličina ne zna prije izvođenja. Dinamički dio je logički ostvaren pomoću podatkovne strukture gomile.

Pošto je upravljanje memorijom težak posao, da bi se olakšao posao programera, postoje funkcije za manipulaciju dinamičkim spremnikom. Dinamički spremnik se može prikazati kao niz okteta. Programer prilikom zauzimanja potrebnog dijela spremnika ne mora znati gdje će se on nalaziti, niti kako do njega doći, nego samo veličinu memorije izraženu u oktetima koja mu je potrebna. U ovom radu se opisuju osnovni algoritmi za dodjelu memorije, te se navode svojstva funkcija za dodjelu dinamičke memorije koja se mogu prikupljati za statističku analizu.

# 2. Osnovni algoritmi upravljanja spremnikom

## 2.1. Upravljanje spremnikom pomoću slijedno povezanih lista

Dinamički spremnik najlakše je kontrolirati pomoći dvostruko povezanih lista. U tom slučaju spremnik se sastoji od niza slobodnih i zauzetih blokova. Blok se sastoji od zaglavlja, i memorijskog prostora. U zaglavlju se nalazi podatak o zauzetosti bloka, podatak o veličini memorijskog prostora te pokazivač na prethodni i sljedeći blok. Veličina zaglavlja je konstantna, dok je veličina memorijskog prostora promjenjiva. Blokovi u listi su povezani slijedno, tako da se prvi blok nalazi na početku gomile, sljedeći blok je odmah iza prvog, itd. do zadnjeg bloka, a između blokova nema nikakvog drugog prostora. Redoslijed blokova u listi, odgovara redoslijedu blokova u fizičkoj memoriji. Na slici 1 nalazi se primjer dinamičkog spremnika da dvostruko poveznom listom.

| | |
|---|---|
| 100 | Veličina: 6 |
| 101 | Slobodan: 0 |
| 102 | Prethodni: NULL |
| 103 | Sljedeći: 110 |
| 104 | |
| 105 | |
| 106 | |
| 107 | |
| 108 | |
| 109 | |
| 110 | Veličina: 4 |
| 111 | Slobodan: 1 |
| 112 | Prethodni: 100 |
| 113 | Sljedeći: 118 |
| 114 | |
| 115 | |
| 116 | |
| 117 | |
| 118 | Veličina: 3 |
| 119 | Slobodan: 0 |
| 120 | Prethodni: 110 |
| 121 | Sljedeći: NULL |
| 122 | |
| 123 | |
| 124 | |

Slika 1: Dinamički spremnik sa dvostruko povezanom listom

Funkcija za alokaciju dinamičkog spremnika kao argument prima veličinu potrebnog prostora. Kada se funkcija pozove, na gomili traži slobodan blok čija je veličina, ne računajući zaglavlje bloka, jednaka ili veća od tražene veličine. Brzina izvođenja funkcije ovisi o algoritmu kojim je implementirana. Osnovni algoritmi za zauzimanje radnog spremnika organiziranog povezanim listama su:

- *Prvi odgovarajući* (engl. *first-fit*)
- *Sljedeći odgovarajući* (engl. *next-fit*)
- *Najbolji odgovarajući* (engl. *best-fit*)
- *Najgori odgovarajući* (engl. *worst-fit*)

Algoritam *prvi odgovarajući* je najjednostavniji algoritam za dodjelu radnog spremnika. u algoritmu se slijedno pretražuje lista blokova od početka, pa do prvog slobodnog bloka, čija veličina memorijskog prostora odgovara traženoj veličini. Zatim se taj blok označava zauzetim.

Algoritam *sljedeći odgovarajući*, je sličan algoritmu *prvi odgovarajući.* Razlika je u tome što algoritam *prvi odgovarajući* blokove pretražuje od početka liste, a algoritam *sljedeći odgovarajući* blokove pretražuje od mjesta gdje je zadnji put stao.

Algoritam *najbolji odgovarajući* pretražuje cijelu listu blokova. Vraća pokazivač na memorijski prostor najmanjeg slobodnog bloka koji odgovara traženoj veličini.

Algoritam *najgori odgovarajući* je suprotan algoritmu *najbolji odgovarajući* jer traži najveći slobodni blok u cijeloj listi.

Osim zauzimanja memorijskog prostora, za kvalitetno upravljanje dinamičkim spremnikom trebaju se implementirati i funkcije za oslobađanje memorijskog prostora. Funkcija za zauzimanje memorijskog prostora vraća pokazivač na memorijski dio bloka. Funkcija za oslobađanje memorije kao parametar prima taj pokazivač i oslobađa blok kojem pripada memorijski prostor na koji pokazuje pokazivač, tako da u zaglavlje bloka upisuje podatak da je blok slobodan.

*Najbolji odgovarajući* i *sljedeći odgovarajući* algoritam se brže izvode od *najboljeg odgovarajućeg* i *najgoreg odgovarajućeg*. Razlog tome je što *najbolji odgovarajući* i *najgori odgovarajući* algoritam uvijek pretražuju cijelu listu, a *prvi odgovarajući* i *sljedeći odgovarajući* pretražuju samo dio liste dok ne naiđu na blok odgovarajuće veličine. Iako su *prvi odgovarajući* i *sljedeći odgovarajući* algoritam

slično izvedeni, u većini slučajeva je *sljedeći odgovarajući* brži. Od navedenih algoritama, najsporiji je *najgori odgovarajući*.

Osim brzine, kao svojstvo algoritama je važna fragmentacija. Postoje dvije vrste fragmentacija, unutarnja i vanjska.

Unutarnja fragmentacija nastaje prilikom zauzimanja bloka čije je veličina veća od tražene veličine. Problem unutarnje fragmentacije lako je rješiv u spremnicima organiziranim na temelju povezanih listi blokova. Ako je veličina memorijskog prostora bloka koji se zauzima veća od tražene i ako je dio koji je višak veći od zaglavlja bloka, blok se dijeli na dva dijela, zauzeti blok, čija veličina odgovara traženoj veličini uvećanoj za veličinu zaglavlja, i slobodni blok, čija je veličina jednaka višku umanjenom za veličinu zaglavlja. Slobodni blok postaje sljedbenik zauzetog bloka.

Vanjska fragmentacija nastaje kada slobodni blokovi postanu premali da bi se mogli iskoristit za veće zahtjeve, tj. kada se broj malih slobodnih blokova jako poveća, dok je broj većih slobodnih blokova malen. Algoritam u daljnjem izvršavanju više neće moći pronalaziti blokove za zahtjeve sa većom zadanom veličinom. Problem vanjske fragmentacije se rješava prilikom oslobađanja bloka. Kada se blok oslobodi, promatraju se njegovi susjedni blokovi, prethodni i sljedeći blok. Ako je jedan od tih blokova slobodan, blokovi se spajaju, tako da nastane novi slobodni blok. Mogu se spojiti oslobođeni blok i njegov prethodnik ako je prethodnik slobodan, oslobođeni blok i njegov sljedbenik ako je sljedbenik slobodan i oslobođeni blok sa prethodnikom i sljedbenikom ako su i prethodnik i sljedbenik slobodni blokovi. Prilikom spajanja bloka, gube se zaglavlja koja se ne nalaze na početku novog slobodnog bloka, a njihov prostor se dodjeljuje memorijskom prostoru novog bloka.

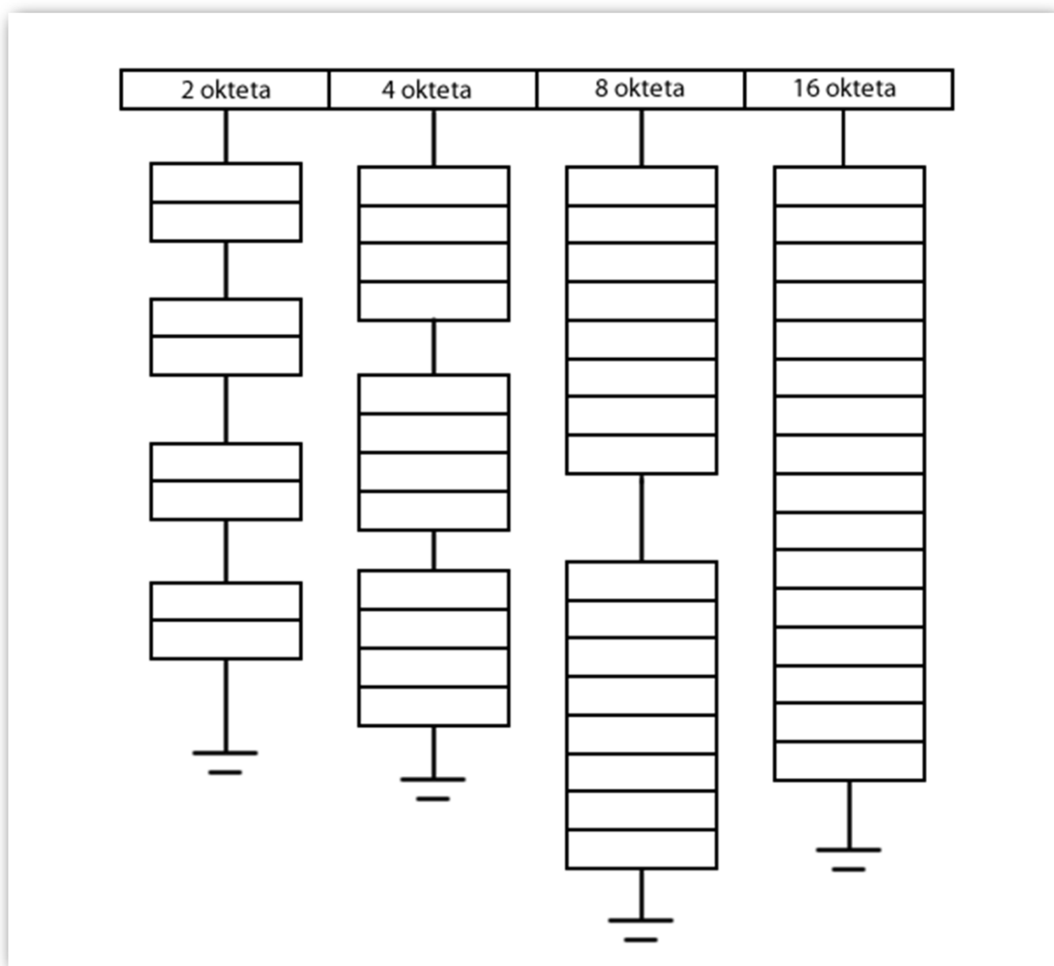
Algoritmi temeljeni na povezanim listama imaju malu unutarnju fragmentaciju. Do unutarnje fragmentacije dolazi samo onda kada je veličina viška memorijskog prostora bloka manja od veličine zaglavlja, a veličina zaglavlja je malena i iznosi nekoliko okteta. Vanjska fragmentacija ovisi o učestalosti pozivanja funkcije za zauzimanje memorije i oslobađanje memorije, te o veličini tražene memorije. Kod *najboljeg odgovarajućeg* algoritma vrlo je mala vjerojatnost da će veličina memorijskog prostora odgovarajućeg bloka biti ista kao i tražena veličina. Zato prilikom korištenja *najboljeg odgovarajućeg* algoritma postoji puno malih slobodnih blokova. Međutim ostaje i puno slobodnih blokova sa velikom veličinom

memorijskog prostora. Kod *najgoreg ogovarajućeg* algoritma, prilikom zauzimanja memorije, pojava malih slobodnih blokova je rijetka, zato jer se uvijek dodjeljuje slobodni blok sa najvećom veličinom. Ali prilikom oslobađanja memorije, nastaje puno malih slobodnih blokova. Vanjska fragmentacija kod *najboljeg odgovarajućeg* algoritma je veća prilikom učestalog zauzimanja memorije, a kod *najgoreg odgovarajućeg* algoritma je veća prilikom učestalog oslobađanja memorije.

## 2.2. Upravljanje spremnikom pomoću odvojenih lista slobodnih blokova

Kod algoritama koji koriste odvojene liste slobodnih blokova, radni spremnik je niz lista. Svaka lista u sebi sadrži slobodne blokove koji su iste veličine, ili iste klase veličina. Redoslijed blokova u listi ne odgovara redoslijedu blokova u fizičkoj memoriji. Kao i kod algoritama temeljenim na slijedno povezanim listama, postoje razne varijacije algoritma za traženje slobodnog bloka. Dvije glavne inačice algoritma su *odvojena pohrana* (engl. s*egregated-storage*) i *odvojeno poklapanje* (engl. *segregated-fit*). Kada algoritam odvojene pohrane prilikom zahtjeva za memorijom ne nađe blok *odgovarajuće veličine*, od operacijskog sustava zatraži još memorije i stvori novu listu sa blokovima zadane veličine. Ako algoritam *odvojenog poklapanja* ne nađe blok odgovarajuće veličine, nastavlja pretragu na blokovima čija je veličina veća od tražene veličine, te pronađeni blok dijeli na dva dijela, slobodni i zauzeti.

*Algoritam susjednih blokova* (engl. *buddy block*) je poseban slučaj algoritma temeljenog na odvojenim listama. Svaka lista sadrži slobodne blokove čija je veličina potencija broja dva. Na taj način spajanje i dijeljenje blokova je lako i brzo izvedivo. Prikaz spremnika sa odvojenim listama prikazan je na slici 2.




Slika 2: Dinamički spremnik sa *buddy block* sustavom

Kada se pojavi zahtjev za memorijom, sa liste čiji blokovi nemaju veličinu veću od dvostruko tražene veličine, uzima se jedan blok, i stavlja se u listu zauzetih blokova. Ako je lista prazna, uzima se blok sa liste koja sadrži blokove dvostruko veće veličine, te se blok dijeli na dva jednaka dijela. Jedan dio se stavlja u listu zauzetih blokova, a jedan se stavlja u prethodnu listu slobodnih blokova koja je bila prazna. Prilikom oslobađanja zauzetog bloka, algoritam provjerava mogu li se njegovi susjedni slobodni blokovi u memoriji spojiti u jedan blok čija je veličina potencija broja dva. Oslobođeni blok se stavlja natrag u odgovarajuću listu koja sadrži blokove iste veličine. Prednost *algoritma susjednih blokova* je ta što blokovi ne moraju imati zaglavlje. Podaci o bloku se znaju ovisno o tome u kojoj se listi blok nalazi. Traženje slobodnog bloka je brzo, jer je dovoljno samo pronaći listu slobodnih blokova i u njoj provjeriti sadrži li slobodni blok odgovarajuće veličine. Mana algoritma je pojava unutarnje fragmentacije, jer se slobodni blok

prilikom zauzeća dijeli na dva samo ako je višak memorijskog prostora bloka veći ili jednak traženoj veličini.

### 2.3. Ostali načini upravljanja spremnikom

Radni spremnik se može organizirati i pomoću naprednijih struktura od liste, kao što su na primjer stabla i bit mape. Primjer algoritma koji koristi stabala je *brzo poklapanje* (engl. *fast-fit)* koji blokove radnog spremnika organizira pomoću *Kartezijevog* stabla. Algoritmi koji rade na principu bit mapa su brzi za dodjeljivanje i oslobađanje memorije, ali je njihova izvedba složena.

Većina operacijskih sustava koristi hibridne algoritme, koji koriste više različitih mehanizama za upravljanje radnim spremnikom. Jedan od poznatijih hibridnih algoritama je *Doug Lea alokacijski mehanizam (DLmalloc)*, koji kombinira razdvojene liste, binarna stabla i dvostruko povezane liste. Ovisno o kategoriji veličine bloka, koristi se mehanizam koji ima najbolje performanse za tu kategoriju. Modificiranu verziju algoritma koristi GNU C biblioteka (*glibc*).

# 3. Ostvarenje analize algoritama za dinamičku dodjelu spremnika

U ovom radu statistička analiza se izvodi nad funkcijom za alokaciju memorije u programskom jeziku C. Funkcija *void *malloc(size_t size)* kao ulazni argument prima potrebnu veličinu memorije izraženu u broju okteta. Rezultat funkcije je pokazivač na slobodni dio memorije koji odgovara traženoj veličini. Ako funkcija ne uspije naći traženu slobodnu memoriju, funkcija vraća *NULL* pokazivač.

Funkcija *void free(void *_Memory)* oslobađa memoriju. Ulazni parametar *_Memory* je pokazivač na dio memorije koji se želi osloboditi. Da bi se funkcija *free* koristila ispravno, ulazni argument koji joj se predaje u prethodnom izvođenju programa mora biti pokazivač dobiven funkcijom *malloc*.

## 3.1. Prikupljanje podataka

Statistička analiza započinje prikupljanjem podataka. Funkcije za alokaciju dinamičkog spremnika kao parametar primaju veličinu traženog prostora, a kao rezultat vraćaju pokazivač na slobodni prostor tražene veličine. To su dva osnovna podatka koja se mogu prikupljati na temelju ulaznih i izlaznih parametara funkcije. Vrlo važno svojstvo funkcije je brzina, pa se uz veličinu zahtjeva i vraćenu adresu prikuplja i vrijeme potrebno za izvršavanje funkcije.

Podatke treba spremati u određenu strukturu podataka. Ako se prilikom izvođenja testa ne zna unaprijed broj pozivanja funkcije za testiranje, struktura podataka za statističke podatke treba biti dinamična. Dinamična struktura bi koristila funkcije koje testiramo, i tako bi interferirala sa rezultatima testiranja. Zato se podaci zapisuju u datoteku, a ne u radni spremnik.

Za testiranje funkcije *malloc* koristi se funkcija *s_malloc.* Pseudokod za funkciju *s_malloc* prikazan je na slici 3.

```
void *s_malloc(size_t size)
{
      void *pokazivac;
      pocniMjeritiVrijeme();
      pokazivac=malloc(size);
      prestaniMjeritiVrijeme;
      zapisiUDatoteku(pokazivac, velicina, vrijeme);
      return pokazivac;
}
```

Slika 3: Pseudokod funkcije *s_malloc*

Na slici se vidi da funkcija *s_malloc* ima iste ulazne i izlazne parametre kao i funkcija *malloc.* Funkcija *s_free* se koristi na isti način kao i funkcija *free*. Pošto parametri funkcije *free* nemaju podatak o veličini memorije koja se oslobađa, funkcija *s_free* u datoteku zapisuje samo adresu prostora koji se oslobađa i vrijeme izvršavanja funkcije. Pseudokod funkcije *s_free* je prikazan slikom 4.

```
void s_free(void *_Memory)
{
      pocniMjeritiVrijeme();
      free(_Memory);
      prestaniMjeritiVrijeme();
      zapisiUDatoteku(adresa, vrijeme);
}
```

Slika 4: Pseudokod funkcije *s_free*

Izvođenje funkcije *malloc* iznosi par stotina nanosekundi. Da bi se vršilo testiranje s takvom rezolucijom vremena, operacijski sustav mora podržavati takve brojače vremena. Implementacija mjerenja vremena ovisi o operacijskom sustavu na kojemu se vrši testiranje. U privitku ovog rada, nalazi se zaglavlje *s_malloc_win.h* u kojem se nalaze funkcije *s_malloc* i *s_free* za Windows operacijske sustave, i zaglavlje *s_malloc_unix.h*. u kojem se nalaze iste funkcije za Unix operacijske sustave. Oba zaglavlja sadrže i funkciju *void s_init(char datotekaMalloc[ ], char datotekaFree[ ]).* Prvi parametar funkcije prima ime datoteke u koju se zapisuju podaci prilikom izvođenja *s_malloc* funkcije, a drugi ime datoteke u koju se zapisuju podaci prilikom izvođenja *s_free* funkcije.

U datoteku se podaci za jedno izvođenje funkcije zapisuju u jednom redu. Za funkciju *s_malloc,* prvo se zapisuje adresa u dekadskom zapisu, zatim veličina i vrijeme izvođenja u nanosekundama. Za funkciju *s_free* prvo se zapisuje adresa u dekadskom zapisu, a onda vrijeme izvođenja u nanosekundama. Podaci u jednom redu datoteke su odmaknuti točno jednim razmakom.

### 3.2. Analiza podataka

Analiza prikupljenih podataka može se vršiti pomoću histograma. U ovom radu analiza se vrši pomoću histograma adresa, histograma veličina i histograma vremena. Na vodoravnoj osi histograma adresa nalaze se adrese koje je vratila funkcija *malloc* ili koje je funkcija *free* oslobodila, a na okomitoj osi broj poziva funkcija *malloc* ili *free*. Na vodoravnoj osi histograma veličina se nalaze veličine sa kojima se pozivala funkcija *malloc*, a na okomitoj osi broj pozivanja funkcije *malloc*. Na vodoravnoj osi histograma vremena nalazi se vrijeme izvođenja funkcije *malloc* ili funkcije *free*, a na okomitoj osi broj poziva funkcija *malloc* ili *free*.

Podaci na vodoravnoj osi histograma su podijeljeni u elemente, tako da se vrijednost sa okomite osi pridružuje jednom elementu, a ne jednoj vrijednosti sa vodoravne osi. Podaci sa vodoravne osi mogu biti raspoređeni po elementima linearnom razdiobom ili eksponencijalnom razdiobom, tako da broj podataka vodoravne osi u jednom elementu ovisi o linearnom ili eksponencijalnom koraku. Broj podataka vodoravne osi u jednom elementu histograma sa linearnom razdiobom jednak je linearnom koraku. Broj podataka vodoravne osi u jednom elementu histograma sa eksponencijalnom razdiobom jednak je broju podataka u prethodnom elementu pomnoženim s eksponencijalnim korakom.

U privitku ovog rada nalaze se programi *malloc_histogrami* i *free_histogrami.* Program *malloc_histogrami* služi za izradu histograma na temelju podataka koje je skupila funkcija *s_malloc.* Prilikom pokretanja programa, od korisnika se traži upis imena datoteke u kojoj se nalaze podaci. Kada se upiše ime datoteke, vrši se analiza podataka. Izračunavaju se minimalne, maksimalne i srednje vrijednosti adresa, veličina i vremena. Zatim korisnik upisuje kakvu razdiobu histograma želi (eksponencijalnu ili linearnu), vrstu podatka za koji želi napraviti histogram (histogram adresa, histogram veličina ili histogram vremena), gornju granicu

histograma, donju granicu histograma i broj elemenata histograma. Zatim se može stvoriti novi histogram, ili se može izaći iz programa. Prvi element histograma će biti broj zapisa u ulaznoj datoteci čija je vrijednost manja od donje granice, a zadnji element će biti broj zapisa čija je vrijednost veća od gornje granice histograma, ako takvi zapisi u datoteci postoje. Tako histogram može imati jedan ili dva elementa više nego što je korisnik zadao. Program histograme sprema u tekstualnu datoteku *histogrami.txt.* Primjer histograma zapisanih u datoteci je prikazan na slici 5. Na isti način radi i program *free_histogrami,* samo što on ne radi sa podacima o veličini.

```
Histogram velicina
Donja granica: 200
Gornja granica: 299
Broj elemenata: 10
Linearni korak: 10
Eksponencijalni korak: 0.000000
[ 4 - 199 ] 6567
[ 200 - 209 ] 327
[ 210 - 219 ] 369
[ 220 - 229 ] 327
[ 230 - 239 ] 320
[ 240 - 249 ] 316
[ 250 - 259 ] 297
[ 260 - 269 ] 312
[ 270 - 279 ] 364
[ 280 - 289 ] 344
[ 290 - 299 ] 296
[ 300 - 1515 ] 40001

Histogram vremena
Donja granica: 600
Gornja granica: 799
Broj elemenata: 5
Linearni korak: 0
Eksponencijalni korak: 1.059224
[ 26 - 599 ] 47916
[ 600 - 634 ] 130
[ 635 - 671 ] 47
[ 672 - 711 ] 22
[ 712 - 753 ] 6
[ 754 - 797 ] 2
[ 798 - 302146 ] 1717
```

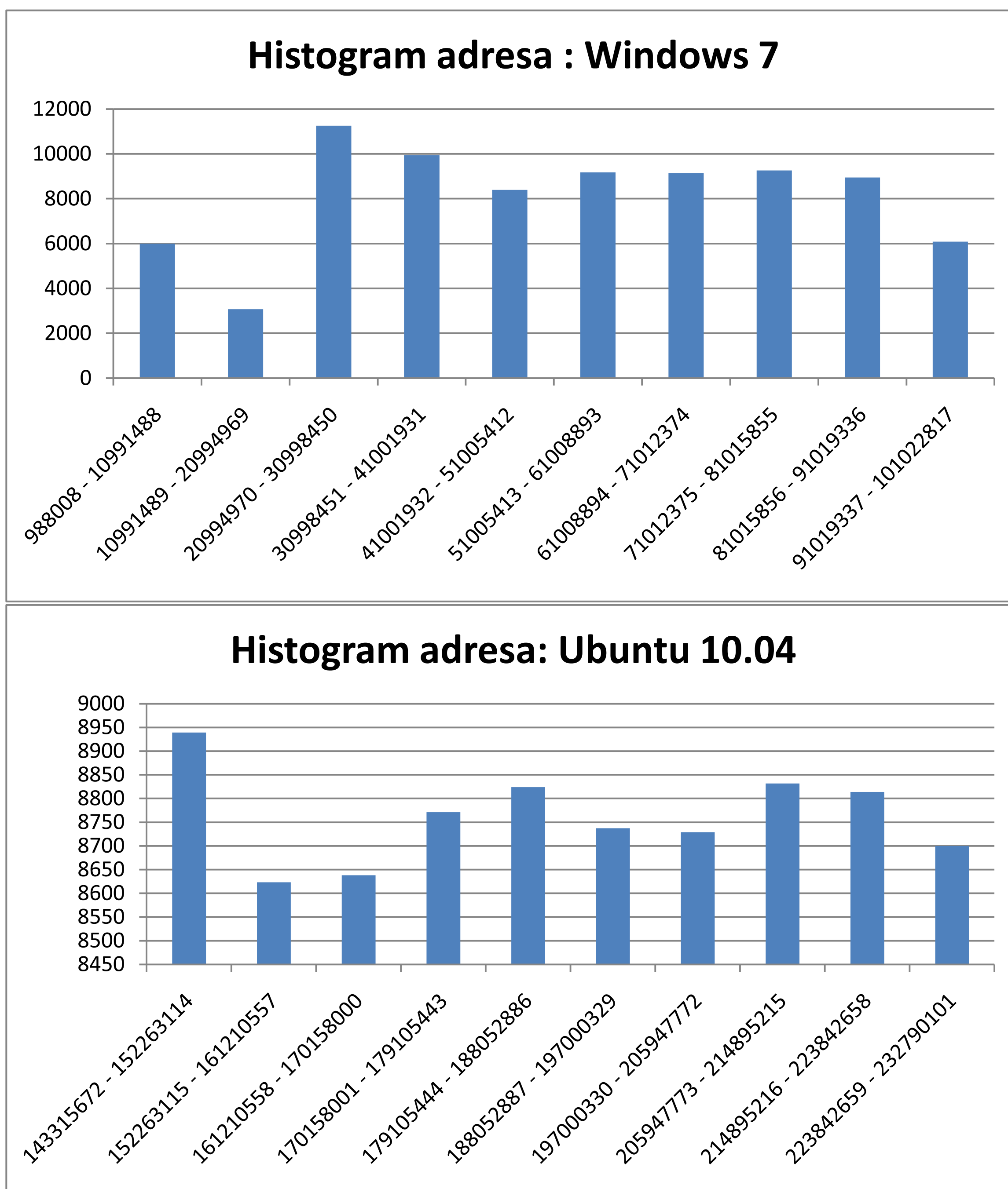
Slika 5: Datoteka histogrami.txt

# 4. Rezultati analize

U privitku ovog rada, nalazi se program *testiranje* za testiranje funkcije *malloc*. Funkcija se testirala na operacijskom sustavu sa Ubuntu 10.04. i na operacijskom sustavu Windows 7. Program poziva funkciju *s_malloc* sa slučajnom veličinom. Najmanja veličina s kojom se poziva *s_malloc* je jedan oktet, a najveća 4096 okteta. Program pamti zadnjih 10000 pokazivača dobivenih funkcijom *s_malloc,* te može pozvati funkciju *s_free* s bilo kojim od tih 10000 pokazivača. Vjerojatnost pozivanja funkcije *s_malloc* je 0.5. Ako se ne pozove funkcija *s_malloc,* onda se poziva funkcija *s_free.* Program se izvodi sve dok funkcija *s_malloc* ne vrati *NULL* pokazivač, ili dok se silom ne prekine.

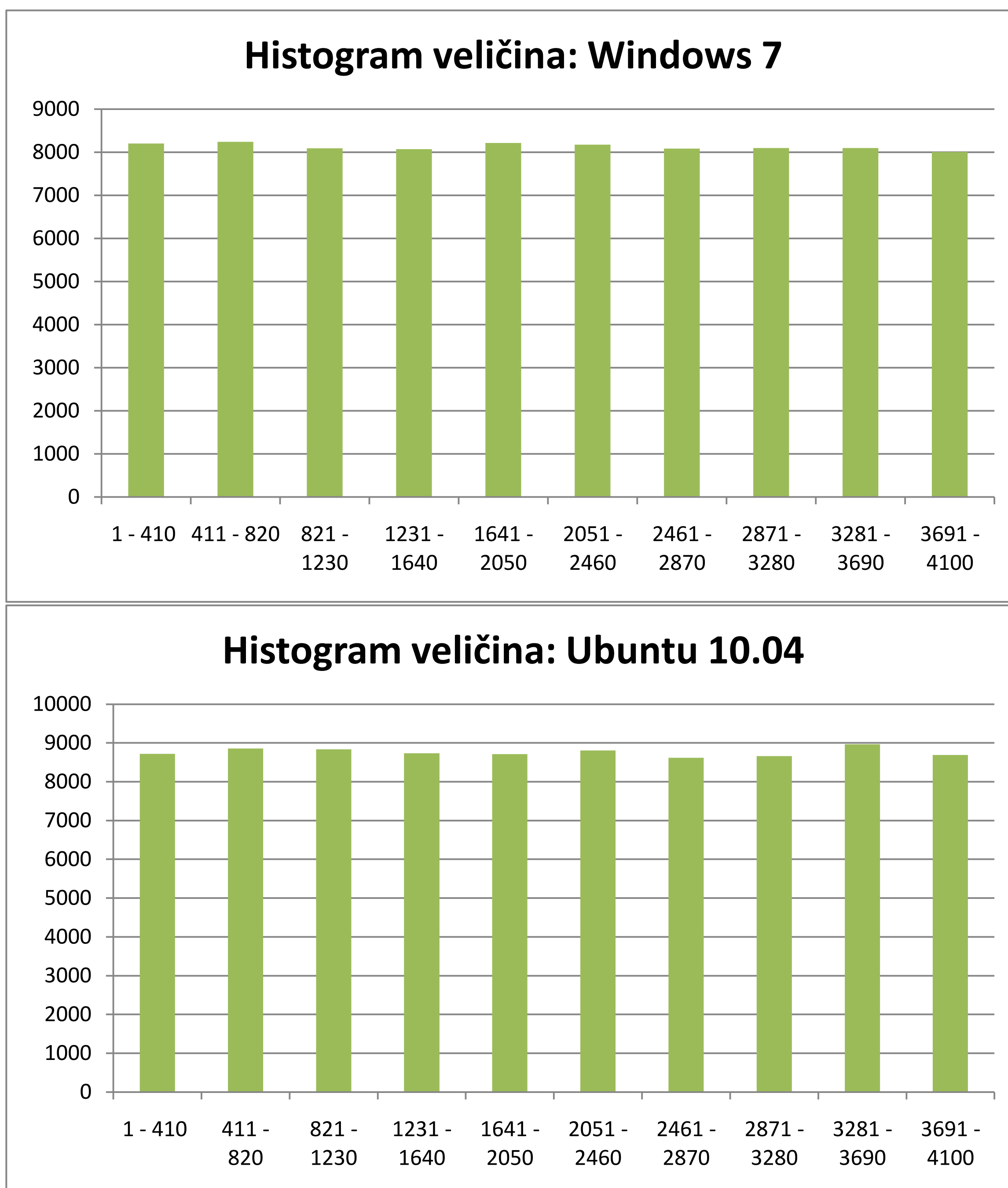
Prilikom testiranja funkcije na operacijskom sustavi Windows 7, broj poziva funkcije *malloc* je 81263, a funkcije *free* 40684. Na operacijskom sustavu Ubuntu 10.4 broj poziva funkcije *malloc* je 87609, a funkcije *free* 43962.

Histogrami adresa za funkciju *malloc* prikazani su na slici 6. Na slici se vidi da je raspodjela adresa slična za oba operacijska sustava. Raspon adresa na operacijskom sustavu Windows 7 je veći, pa se može zaključiti da je Windows 7 troši više memorije za dinamički spremnik, ali prilikom testiranja na operacijskom sustavu Windows 7 funkcija *malloc* je pozvana više puta nego na operacijskom sustavu Ubuntu 10.04., te je zbog toga u dinamičkom spremniku ukupna veličina zauzete memorije veća.




Slika 6: Histogrami adresa za funkciju *malloc*

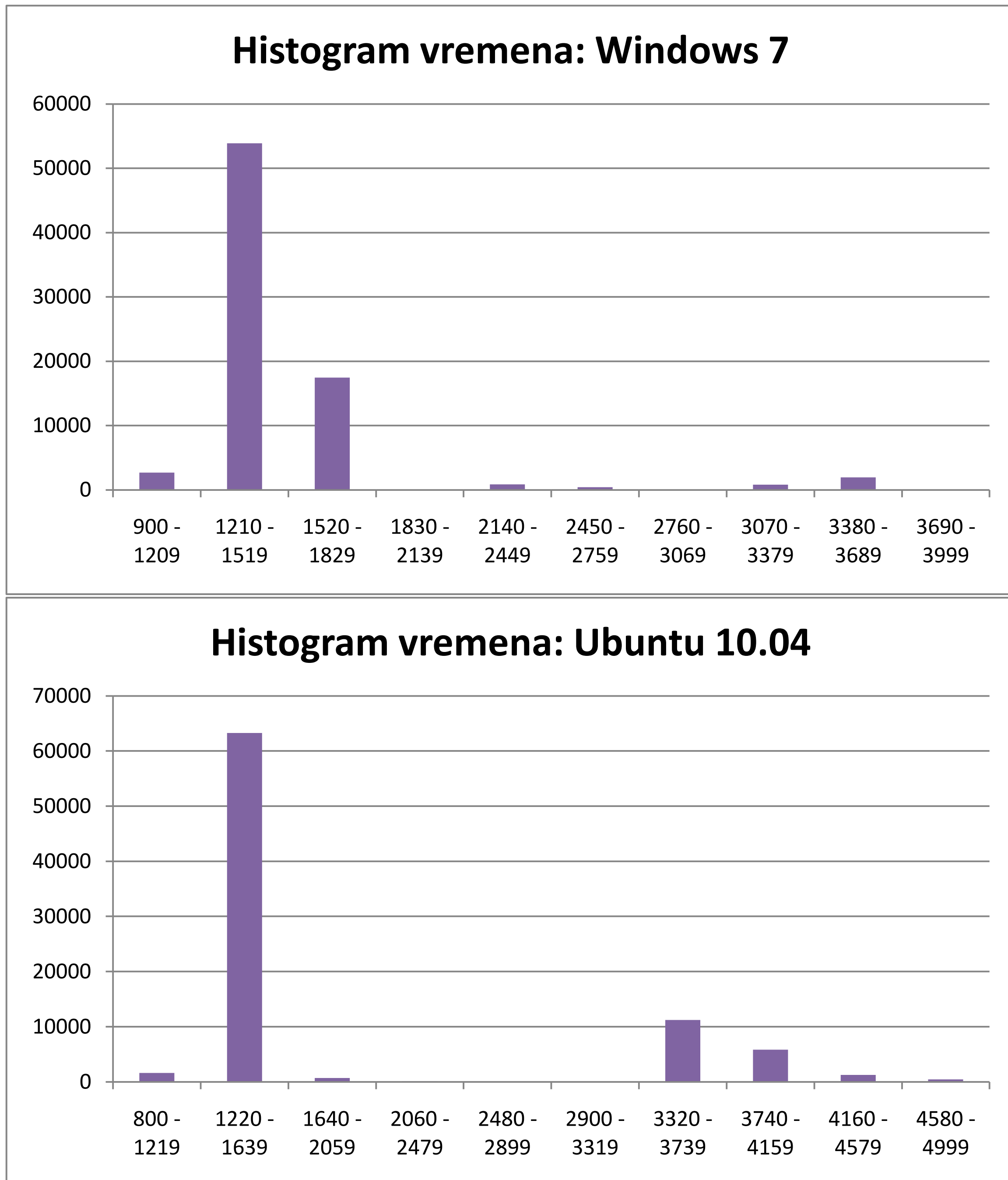
Histogrami veličina za funkciju *malloc* nalaze se na slici 7. Raspodjela veličina na histogramima je jako slična. Razlog tome je što su se u oba testa veličine generirale slučajnim brojem u intervalu čija je donja granica 1, a gornja 4096. Zato broj pozivanja funkcije *malloc* za pojedinu veličinu približno jednak broju pozivanja za ostale veličine.




Slika 7: Histogrami veličina za funkciju *malloc*

Histogrami vremena za funkciju *malloc* nalaze se na slici 8. Prosječno izvođenje funkcije na operacijskom sustavu Windows 7 iznosi 1936 nanosekundi, a na operacijskom sustavu Ubuntu 10.04 2520 nanosekundi. Međutim, najkraće vrijeme izvođenja na operacijskom sustavu Ubuntu 10.04 je manje nego na operacijskom sustavu Windows 7. Na histogramima se vidi da je odstupanje trajanja izvođenja funkcije od prosječnog trajanja na operacijskom sustavu

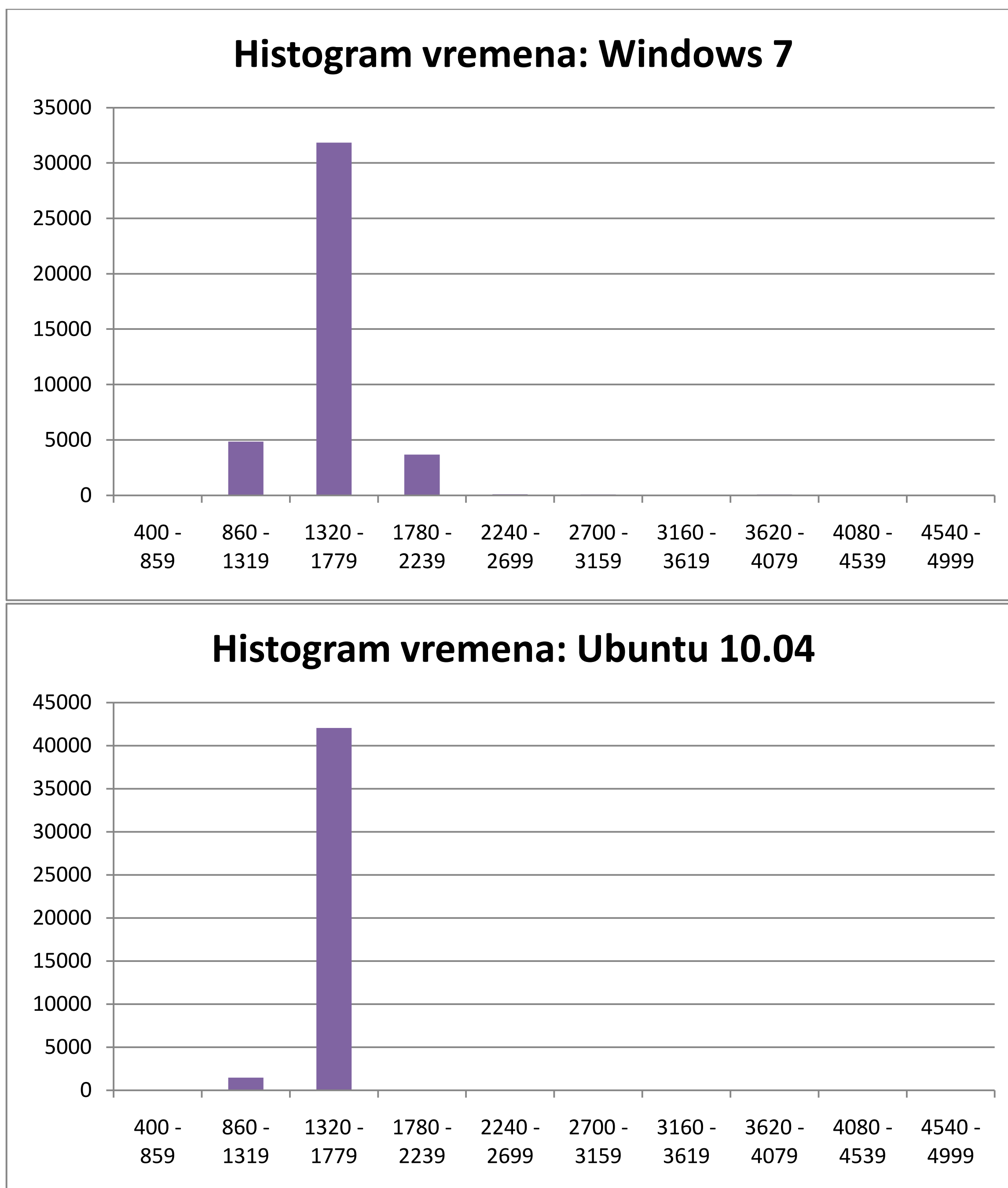
Windows 7 manje, jer na operacijskom sustavu Ubuntu 10.04 veliki broj izvođenja funkcije trajao je oko 3700 nanosekundi, kao što je prikazano na sedmom i osmom stupcu drugog histograma na slici.




Slika 8: Histogrami vremena za funkciju *malloc*

Histogrami vremena za funkciju *free* prikazani su na slici 9. Prosjek izvođenja funkcije na operacijskom sustavu Windows 7 iznosi 1375 nanosekundi, a na operacijskom sustavu Ubuntu 10.04 iznosi 1960 nanosekundi. Na histogramu se

vidi da trajanje izvođenja funkcije na operacijskom sustavu Ubuntu 10.04 ima manje odstupanje od prosječnog vremena izvođenja.




Slika 9: Histogrami vremena za funkciju *free*

# 5. Zaključak

Svojstva algoritama za dinamičko dodjeljivanje radnog spremnika koja se mogu mjeriti su vrijeme izvođenja i fragmentacija. Ponekad dobra vremenska svojstva algoritama uzrokuju veliku fragmentaciju, a mala fragmentacija uzrokuje loša vremenska svojstva.

Bez znanja o implementaciji funkcije za dodjelu dinamičkog spremnika se ne mogu skupljati podaci o unutrašnjoj fragmentaciji.

Odvojeno skupljanje podataka o funkciju za zauzimanje prostora i funkciji za oslobađanje prostora se pokazalo lošom implementacijom analize, jer se tako ne mogu rekonstruirati podaci u vanjskoj fragmentaciji.

Ne postoji najbolji algoritam za dodjelu dinamičkog spremnika. Brzina izvođenja i fragmentacija ne ovise samo o implementaciji algoritma, nego i o prirodi problema koji rješava program koji koristi funkcije za dinamičko dodjeljivanje spremnika. Neki algoritmi lakše rade sa manjim zahtjevima veličine, a neki sa većim zahtjevima veličine. Zato većina operacijskih sustava koristi hibridne algoritme, koji zahtjeve za različite klase veličina rješavaju na drugačiji način.

Potpis:

______________________________

# 6. Literatura

[1] *Buddy memory allocation*
http://en.wikipedia.org/wiki/Buddy_memory_allocation, lipanj 2011

[2] *Inside memory management*
http://www.ibm.com/developerworks/linux/library/l-memory/, lipanj 2011.

[3] *Memory Allocators*
http://www.cs.nmsu.edu/~ekerriga/presentation/index2.html, lipanj 2011.

[4] Puaut. I. *Real-Time Performance of Dynamic Memory Allocation Algorithms.* 14th Euromicro Conference on Real-Time Systems, Beč, Austrija, lipanj 2002, stranice 41-49

# 7. Sažetak

**Naslov:** Statistička analiza algoritama za dinamičko upravljanje spremnikom

**Sažetak:** Dinamički spremnik je dio memorije koji program koristi za spremanje podataka čija se veličina sazna tek prilikom izvođenja programa. Postoje razni načini za upravljanje dinamičkim spremnikom. Osnovni algoritmi za dodjelu dinamičkog spremnika su: *najbolji odgovarajući*, *najgori odgovarajući*, *prvi odgovarajući*, *sljedeći odgovarajući* i *odvojeno poklapanje*. Osim dodjeljivanja radnog spremnika, svaki algoritam ima i funkciju oslobađanja dinamičkog spremnika. Algoritmi ovise o strukturi podataka pomoću koje je organiziran radni spremnik. Osnovne strukture podataka za organizaciju radnog spremnika su liste, stabla i bit mape. Performanse algoritama koje se mogu mjeriti su brzina i fragmentacija. Postoje dvije vrste fragmentacije, unutarnja i vanjska. Parametri koji se mogu skupljati za analizu performansi su vrijeme izvođenja, veličina prostora pojedinog zahtjeva za memorijom i adresa u radnom spremniku koji je algoritam vratio za traženi zahtjev.

**Ključne riječi:** alokacija, analiza, radni spremnik

## Summary

**Title:** Statistical analysis of algorithms for dynamic memory management

**Summary:** Dynamic memory is a part of memory used for storage of data with unknown size until runtime. There are various ways of managing dynamic memory. Basic algorithms for dynamic memory allocation are: best-fit, worst-fit, first-fit, next-fit and segregated-fit. Aside from allocating space, each algorithm has a function for freeing allocated space. Algorithms depend on a data structure used to organize dynamic memory. Basic data structures for organization of dynamic memory are lists, trees and bitmaps. The performances of algorithms which can be measured are speed and fragmentation. There are two types of fragmentation, internal and external. The parameters that can be collected for performance analysis are algorithm runtime, size of requested memory and address in memory returned by the algorithm for the specific request.

**Key words:** allocation, analysis, memory

# 8. Privitak

Radu u tiskanom obliku priložen je CD optički medij. Na njemu se nalaze zaglavlja *s_malloc_win.h* i *s_malloc_unix.h*, programi *free_histogrami, malloc_histogrami* i *testiranje* u izvornom i izvršnom obliku, te ovaj rad u PDF formatu. Programi *free_histogrami* i *malloc_histogrami* su pisani u programskom jeziku C++, a program *testiranje* i zaglavlja *s_malloc_win.h* i *s_malloc_unix.h* u programskom jeziku C.