SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1788

PRILAGODBE ALGORITMA ZA DINAMIČKO UPRAVLJANJE SPREMNIKOM

Nenad Merdanović

Sadržaj

Uvod

Zbog sve bržeg razvoja računalne tehnologije i brzine memorijskog spremnika, konstantno se traže novi načini upravljanja spremnikom. Kako bi se što efikasnije upravljalo radnim spremnikom potreban je brz i jednostavan algoritam koji će, uz što manje izgubljenog prostora na zaglavlja, brzo i točno alocirati memoriju, te je po potrebi osloboditi.

Ovdje nailazimo na problem nemogućnosti pronalaska univerzalnog algoritma za sve namjene. Ta nemogućnost proizlazi iz namjene računalnog sustava za koji je algoritam izgrađen. Za primjer možemo uzeti algoritam koji je razvijen za servere s velikom količinom brze memorije. Algoritam razvijen za tu primjenu će se ponašati jako dobro u tom okruženju, ali isti neće biti dobar za ugradbeno računalo vrlo ograničene memorije.

U ovom radu biti će prikazana dva algoritma za dinamičko upravljanje spremnikom. Razvijeni algoritmi su u mogućnosti alocirati i osloboditi dijelove memorijskog spremnika, te efikasno spajati susjedne slobodne blokove u veće slobodne blokove. Prvi algoritam koristi odvojeni dio spremnika za spremanje zaglavlja koji pokazuju na zauzete i slobodne blokove. Drugi algoritam koristi zaglavlja samo u slobodnim blokovima, pa se programer mora brinuti oko lokacije i veličine zauzetih blokova.

1. Primjeri dinamičkog upravljanja spremnikom

1.1. Algoritmi za pronalazak slobodnog bloka [1]

Prilikom alokacije memorijskog bloka, treba odrediti koji će se blok unutar radnog spremnika dodijeliti. Određivanje se vrši pomoću nekoliko vrsta algoritama, od kojih svaki ima svoje prednosti i mane, te područje uporabe.

1.1.1. First fit algoritam

First fit algoritam je najjednostavniji algoritam za određivanje bloka koji će se dodijeliti. Zaglavlja slobodnih blokova su uobičajeno implementirana kao vezana lista. Ta lista može biti jednostruko, dvostruko i ciklički vezana, ovisno o namjeni i implementaciji. First fit algoritam koristi nesortiranu vezanu listu zaglavlja slobodnih blokova koju onda pretražuje. Prilikom pretrage, algoritam provjerava veličinu slobodnog bloka opisanog zaglavljem, te u trenutku kada pronađe dovoljno velik blok, njega zauzima (djelomično ili potpuno). Algoritam se koristi gdje je potrebna vrlo brza alokacija i oslobađanje, jer donosi najmanje složen rad s listama.

1.1.2. Best fit algoritam

Best fit algoritam koristi listu sortiranu uzlazno po veličini. U tom slučaju uvijek se odabire prvi blok koji je dovoljne veličine. Zbog sortiranosti liste, takav blok je veličinom najbliži zahtijevanom bloku (od onih dovoljne veličine). Ovakav način alokacije vrlo rijetko dovodi do razdvajanja bloka na dva dijela (slobodni i zauzeti dio), jer je uobičajeno da je pronađen blok vrlo blizak veličinom onom traženom.

1.1.3. Next fit algoritam

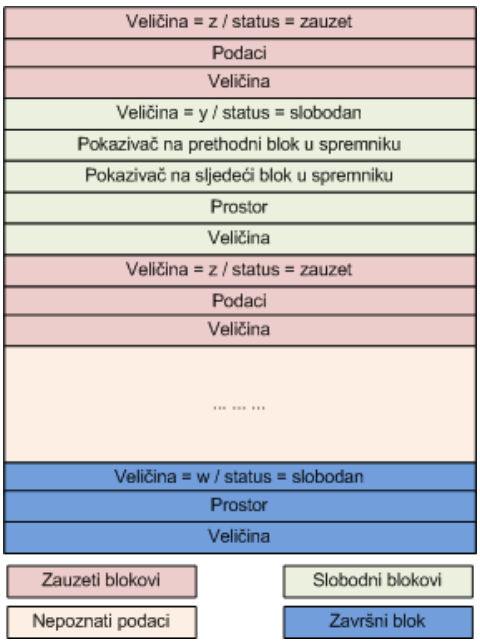
Next fit algoritam koristi pokazivač koji ciklički prolazi listu slobodnih blokova, dok ne nađe prvi blok odgovarajuće veličine, te njega koristi za alokaciju. Pokazivač se tada pomiče na sljedeći blok, te prilikom sljedeće alokacije kreće u prolazak liste s te pozicije. Razlog za korištenje ovakvog algoritma je izbjegavanje stvaranja velikog broja malih slobodnih blokova na početku radnog spremnika koje bi trebalo pretraživati svaki puta. Najčešće se koristi lista zaglavlja sortiranih po adresama blokova koje opisuju.

1.2. Doug Lea's Malloc – dlmalloc [2]

Doug Lea's Malloc predstavlja algoritam i implementaciju sustava za dinamičko upravljanje spremnikom. Nastao je 1987. godine i do danas je ostao najčešće korišteni oblik *malloc()* algoritma i implementacije. Velik broj drugih alokatora i algoritama koristi dlmalloc kao razvojnu bazu i inspiraciju.

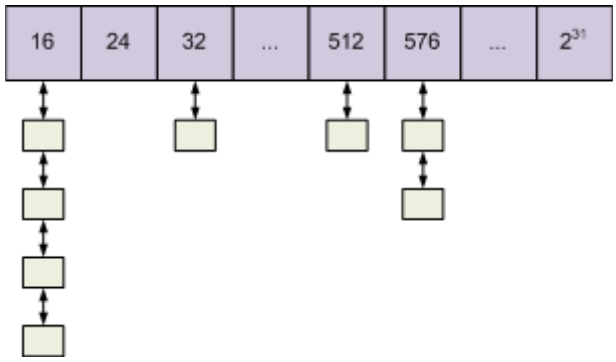
Iako je kroz godine prošao kroz vrlo velik broj poboljšanja i revizija samog algoritma, algoritmi koji čine jezgru ovog alokatora ostali su isti. Radi se o dva algoritma određuju način raspodjele radnog spremnika.

Jedan algoritam se naziva *Boundary tags*, odnosno granične markacije. Radi se o jednostavnoj ideji da svaki blok radnog spremnika sadrži informaciju o svojoj veličini, a slobodni blokovi sadrže pokazivače na prethodni i sljedeći slobodni blok memorije. U prvim verzijama svi dijelovi su sadržavali sve informacije, ali se zbog boljeg iskorištenja prostora, na štetu detekcije grešaka, išlo na izbacivanje dijela informacije iz zauzetog bloka. Ovaj koncept je iskorišten i prilikom izrade jednog od algoritama opisanih u ovom radu, ali izmijenjen kako bi bio upotrebljiviji u ciljanoj okolini.



Sl. 1.1: Granične markacije

Drugi algoritam naziva se *Binning*, a odnosi se na sortiranje slobodnih blokova memorije po njihovoj veličini. Blokovi su raspoređeni u „spremnike“ koji označavaju koliko je blok velik. U trenutnoj implementaciji postoji 128 „spremnika“ koji su logaritamski raspodijeljeni između 8 okteta i 2^{31} okteta. Za blokove manje od 512 okteta razmak između „spremnika“ jest 8 okteta, a neposredno nakon 512 okteta on iznosi 64 okteta, te tako logaritamski raste do 2^{31} okteta.



Sl. 1.2: Slobodni blokovi raspoređeni u spremnike

Kako se u dlmalloc-u koristi *best fit* algoritam za pronalazak slobodnog bloka, ovakav način organizacije lista slobodnih blokova omogućava mu brz pronalazak adekvatnog slobodnog bloka, njegovo zauzimanje i ponovno povezivanje liste unutar spremnika.

Negativne strane su te što svaki slobodan blok mora imati pokazivače na prethodni i sljedeći blok u spremniku, te veličinu slobodnog bloka. Na taj način u 32 bitnom sustavu minimalna veličina slobodnog bloka jest 16 okteta, a u 64 bitnom sustavu jest 24 okteta. Minimalna veličina bloka od 16, odnosno 24, okteta dovodi do vrlo loše iskorištenosti prostora kod alociranja malih količina memorije, npr. kod rada s jednostavnim vezanim listama.

1.3. Jemalloc [3]

Razvojen računalne tehnologije i pojavom višejezgrenih i višeprocesoriskih sustava, došlo je do potrebe razvoja novog alokatora za rad u SMP okolinama. Zbog toga je Jason Evans, član FreeBSD razvojnog tima, krenuo u razvoj jemalloc sustava. Sustav je namijenjen kao zamjena za standardni *malloc()* u SMP sustavima.

Ovaj sustav koristi tzv. arene za razdvajanje memorijskog prostora između dretvi (*threadova*) koji se izvršavaju na različitim procesorima. Na taj način se eliminira potreba za pretjeranim zaključavanjem pristupa memoriji, te ponovno čitanje iz memorije umjesto iz međuspremnika.

2. Zahtjevi prema dinamičkom upravljanju spremnikom

Mjerenje performansi algoritma za dinamičko upravljanje spremnikom je vrlo složen proces. Osim same složenosti procesa, mora se prvo ustanoviti u kakvom okruženju će se koristiti algoritam kojem su mjerene performanse, kako bi se znalo koji podaci su nam ključni, te na temelju kojih će se podataka zaključiti je li algoritam dovoljno dobar za uporabu u takvom okruženju.

Neki od podataka koji se mjere su [3]:

- trajanje (složenost) zauzimanja i oslobađanja memorije
- ukupna zauzeta količina memorije testne aplikacije
- unutrašnja fragmentacija
- vanjska fragmentacija
- skalabilnost

Trajanje zauzimanja i oslobađanja memorije odnosi se na algoritme korištene za povezivanje memorijskih blokova u unutrašnjim strukturama sustava. Te operacije trebaju biti što kraće.

Ukupna zauzeta količina memorije za neku testnu aplikaciju je dobar pokazatelj efikasnosti algoritma za dinamičko upravljanje spremnikom, jer pokazuje, u odnosu na ostale algoritme, koliko se efikasno iskorištava zauzeti dio spremnika.

Unutrašnja fragmentacija jest mjera količina iskoristivog dijela bloka unutar spremnika u odnosu na veličinu bloka. Spremnik, osim korisnog dijela, sadrži i zaglavlje, tj. informacije o samome bloku. Količina tih informacija treba biti što je moguće manja, kako bi unutrašnja fragmentacija bila što manja.

Vanjska fragmentacije jest mjera koja pokazuje iskoristivost memorijskog prostora. Računa se kao odnos zauzetog dijela spremnika i korištenog dijela spremnika.

Skalabilnost jest mogućnost upotrebljavanja istog algoritma na velikom broju sutava, odnosno od slabog jednoprocesorskog sustava s malo memorije, do većih poslužitelja s nekoliko desetaka višejezgrenih procesora i velikom količinom memorije.

Glavni zahtjev u ovome radu bio je da unutrašnja fragmentacija bude što manja, odnosno da bilo kakve informacije koje opisuju zauzeti blok memorije ne smiju biti sadržane u samome bloku. Vanjska fragmentacija može se poboljšati korištenjem *best fit* algoritma, a ne *first fit* kako je implementirano u ovim algoritmima.

3. Realizacija dinamičkog upravljanja spremnikom bez korištenja zaglavlja

Realizacija dinamičkog upravljanja spremnikom bez korištenja zaglavlja unutar blokova zahtjeva drugačiji pristup problematici od pristupa kod uobičajenih algoritama za dinamičko upravljanje spremnikom. Ovaj rad objašnjava dva takva pristupa, daje objašnjenje rada oba algoritma, te uspoređuje njihove efikasnosti.

3.1. Zaglavlja u zasebnom dijelu spremnika

Ovaj pristup jedinstven je po tome što zaglavlja odvaja u poseban dio spremnika. Taj dio nalazi se na početku spremnika i unaprijed je određene veličine. U taj dio se spremaju zaglavlja svih zauzetih i slobodnih blokova. Pri tome se koriste dvije dvostruko povezane liste za manipulaciju zauzetim i slobodnim blokovima. Kako se oslobađanjem memorije oslobađa i jedno zaglavlje, potrebno je imati i listu slobodnih zaglavlja. Ona je realizirana jednostruko povezanom listom iz razloga što se uvijek uzima prvo slobodno zaglavlje.

Ovaj algoritam zauzima memoriju *first fit* algoritmom, odnosno uzima prvi dovoljno velik blok, te ga po potrebi razdvaja na dva dijela: zauzeti blok čija se adresa vraća pozivatelju i slobodni blok koji ostaje u listi slobodnih. Možemo definirati i minimalnu iskoristivu veličinu bloka, kako bismo izbjegli velik broj vrlo malih, a samim time i praktično neiskoristivih blokova. Ukoliko zatražimo blok veličine x, a blok je veličine manje ili jednake od $x + d$, gdje je d minimalna iskoristiva veličina bloka, zauzeti blok će biti veličine $x+d$.

First fit algoritam

```
header *find_block(header *iter, size_t reqsize) {
 while(iter != NULL && iter->size < reqsize)
       iter= iter->next;
 return iter;
}
```

3.1.1. Prikaz rada algoritma s izdvojenim zaglavljima

Rad algoritma započinje inicijalizacijom radnog spremnika. Najprije se odredi veličina prostora za zaglavlja, te se unutar tog prostora zapiše jedno zaglavlje za slobodan blok koje pokazuje na cijeli radni spremnik. Osim tog zaglavlja zapiše se i jedno slobodno zaglavlje kako bi se odredilo koje zaglavlje će se koristiti nakon prve alokacije za spremanje informacija o novonastalim blokovima.

Algoritam se bazira na tri strukture podataka:

- struktura za slobodne blokove - *free_block*
- struktura za zauzete blokove - *res_block*
- struktura za slobodna zaglavlja - *free_header*

Izgled struktura:

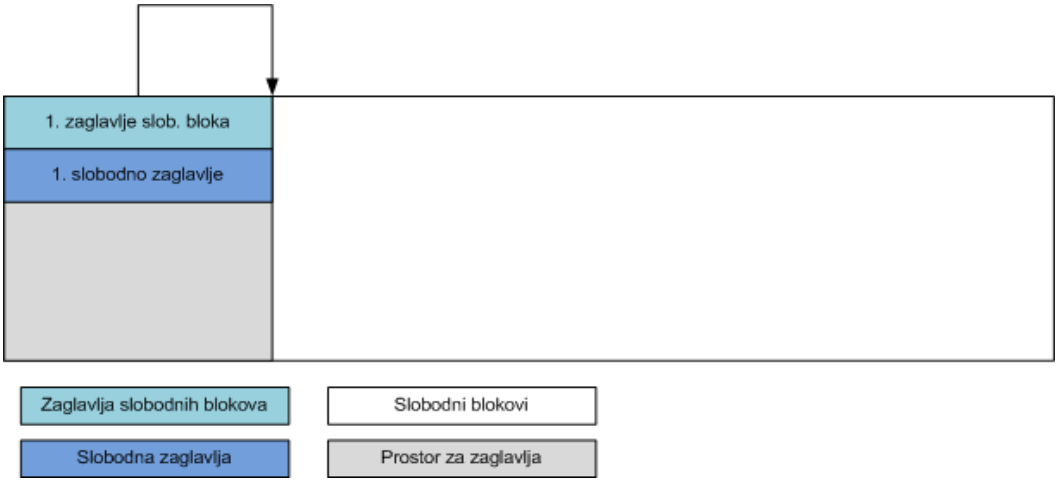
```
struct free_block { /* Ista struktura kao i res_block */
 struct free_block *next;
 struct free_block *prev;
 int adr;
 size_t size;
}


typedef struct free_header {
 struct free_header *next;
 struct free_header *ftemp;
 int temp;
 size_t stemp;
} free_header;
```

Kao što se vidi, strukture za slobodne i zauzete blokove su iste, dok se struktura za slobodna zaglavlja razlikuje. Kako bi se postiglo da su sve strukture jednake veličine u memoriji, struktura za slobodna zaglavlja se morala proširiti „beskorisnim“ varijablama.

Inicijalizaciju radnog spremnika obavlja funkcija *mem_init*, koja kao parametre prima adresu radnog spremnika, te njegovu veličinu.

Prototip funkcije mem_init

```
void mem_init (void *adr, size_t size)
```



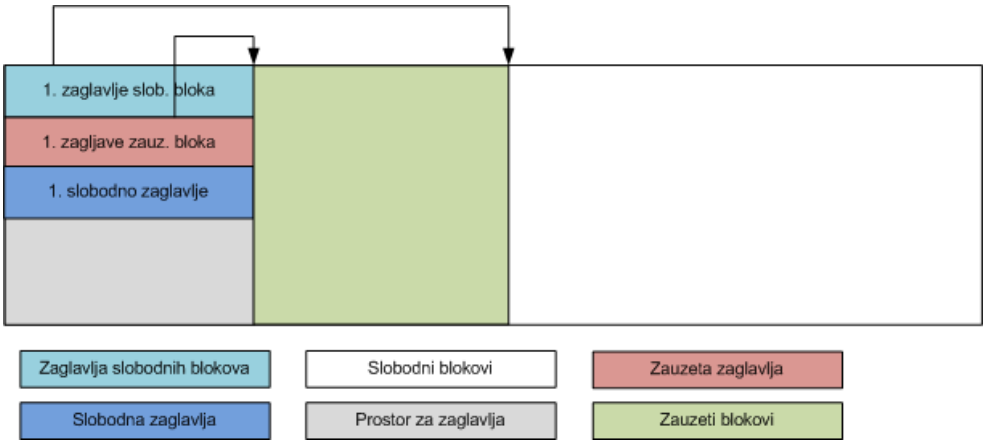
Sl. 3.1: Inicijalizacija radnog spremnika

Prilikom zauzimanja prvog bloka zauzima se dio velikog slobodnog bloka, te se stvara još jedno zaglavlje koje će služiti kao opisnik novostvorenog bloka. Novi blok se stavlja na vrhu liste zauzetih blokova, te se povezuje s ostalim elementima liste. Ukoliko treba zauzeti cijeli slobodni blok, ne stvara se novo zaglavlje, već se koristi postojeće. U tom slučaju trebamo prebaciti to zaglavlje iz liste slobodnih blokova u listu zauzetih blokova (Vidi sliku).

Alokaciju bloka memorije obavlja funkcija *mem_alloc* koja kao parametar prima željenu veličinu memorije i pokazivač na lokaciju gdje će se upisati stvarna veličina, a vraća adresu alociranog bloka memorije.

Prototip funkcije mem_alloc

```
void *mem_alloc (size_t size, size_t *rsize)
```



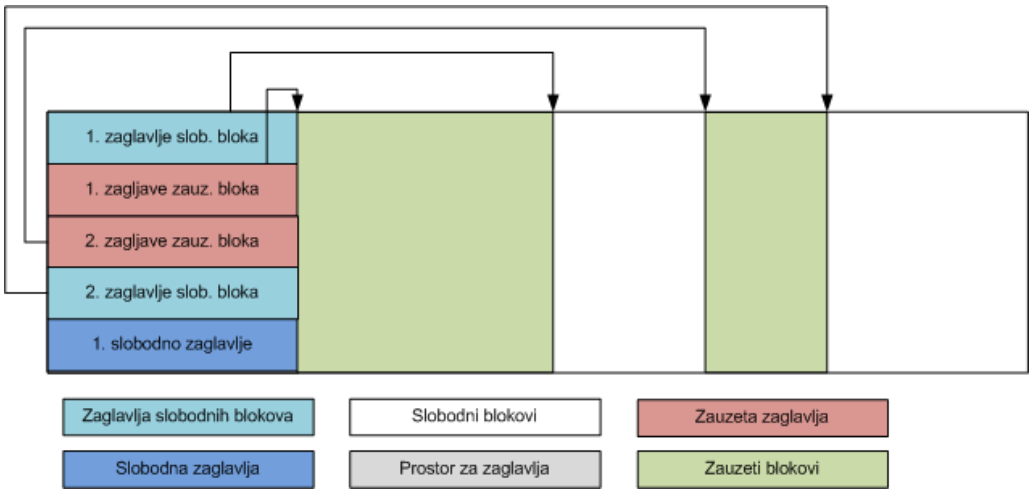
Sl. 3.2: Alokacija bloka u radnom spremniku

Kod oslobađanja memorije, iteriramo kroz listu slobodnih blokova dok ne dođemo do opisnika bloka koji treba osloboditi. Zatim provjeravamo možemo li spojiti taj blok s susjednima, te ga spajamo ovisno o mogućnostima.

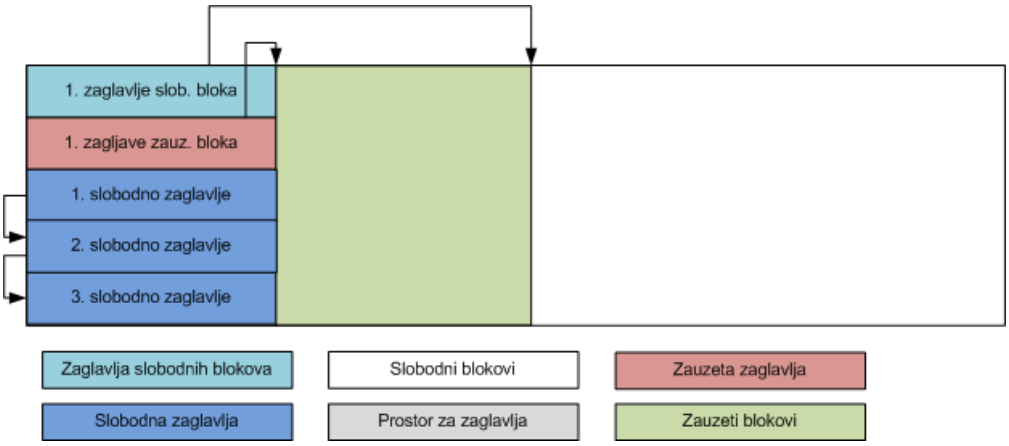
Oslobađanje bloka memorije obavlja funkcija *mem_free* koja kao parametar prima adresu slobodnog bloka koji treba osloboditi. Funkcija prima relativnu adresu u odnosu na pocetak bloka.

Prototip funkcije mem_free

```
int mem_free (int adr)
```



Sl. 3.3: Stanje radnog spremnika prije oslobađanja bloka



Sl. 3.4: Stanje radnog spremnika nakon oslobađanja i spajanja susjednih blokova

3.1.2. Ograničenja algoritma s izdvojenim zaglavljima

Najveće ograničenje algoritma jest ograničen broj blokova na koje radni spremnik može biti podijeljen. Kada dođe do popunjenja dijela spremnika za zaglavlja, nemoguće su daljnje alokacije koje bi trebale razdvojiti blok na dva dijela. Zbog toga ovaj algoritam je primjenjiv samo u okolinama gdje se može predvidjeti ukupan broj potrebnih blokova memorije.

3.2. Zaglavlja samo u slobodnim blokovima

Ovaj algoritam koristi zaglavlja samo u blokovima koji su slobodni, dok zauzeti blokovi ne sadrže nikakve podatke o sebi. Svaki slobodan blok na svom početku i na kraju sadrži zaglavlje koje opisuje taj blok. To zaglavlje se sastoji od pokazivača na prethodni i sljedeći slobodni blok, te veličine tog bloka. Pokazivači uvijek pokazuju na zaglavlje koje se nalazi na početku slobodnog bloka. Svako zaglavlje sadrži i magični broj pomoću kojeg je moguća identifikacija da se radi o slobodnom bloku.

```
typedef struct header {
  struct header *next;
  struct header *prev;
  int magic_no;
  size_t size;
} header;
```

Magični brojevi za početak i za kraj slobodnog bloka se razlikuju, kako bismo pokušali spriječiti da se slučajno magični broj za početak bloka nađe na kraju bloka nehotičnim zapisivanjem.

```
#define MAGIC_START 0x12345678 /* Pocetak bloka */
#define MAGIC_END 0x90123456 /* Kraj bloka */
#define MEMORY_START 0x78901234 /* Pocetak radnog spremnika */
#define MEMORY_END 0x56789012 /* Kraj radnog spremnika */
```

3.2.1. Prikaz rada algoritma s zaglavljima samo u slobodnim blokovima

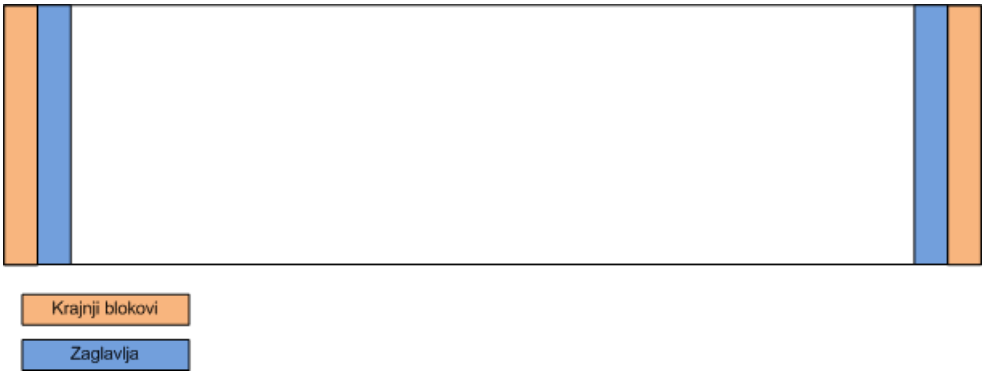
Prilikom inicijalizacije radnog spremnika, zapisuju se prva zaglavlja. Ta zaglavlja se nalaze na početku i na kraju radnog spremnika i ona označavaju njegove krajeve, te se ona nikada neće mijenjati. Ona se koriste kako bi se definirala veličinu radnog spremnika i kako u niti jednom trenutku stanje memorije izvan radnog spremnika ne bi bilo čitano ili zapisivano.

Unutar tih zaglavlja definiramo korisno područje radnog spremnika i u njega zapisujemo zaglavlja koja ga opisuju. Veličina u zaglavlju se postavlja na veličinu korisnog dijela radnog spremnika, a njegovi pokazivači se postavljaju kao *NULL* pokazivači.

Funkcija koja inicijalizira memoriju naziva se *mem_init* i kao parametre prima adresu i veličinu radnog spremnika.

Prototip funkcije mem_init

```
int mem_init(void *adr, size_t size)
```



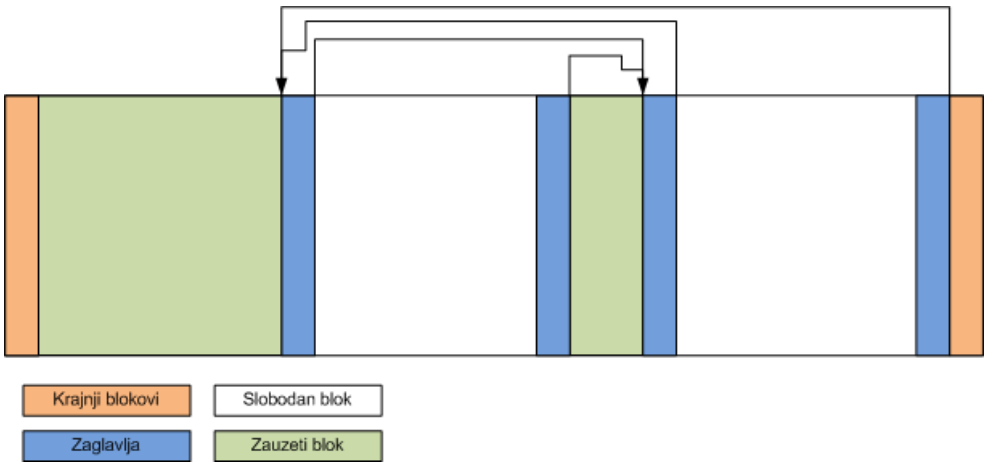
Sl. 3.5: Inicijalizacija radnog spremnika

Prilikom alokacije memorije pregledava se lista svih slobodnih blokova kako bi se našao adekvatan slobodan blok za alokaciju. Algoritam koji se ovdje koristi je jednak kao i kod algoritma s odvojenim zaglavljima, odnosno *first fit* algoritam. Tim algoritmom pronalazi se dovoljno velik blok, te ga se po potrebi razdvaja na slobodni dio i dio koji ćemo alocirati. U ovom koraku se opet pregledava ostaje li dovoljno prostora u slobodnom bloku za praktično iskorištenje. Ukoliko je blok premalen za razdvajanje, a veći od zahtijevane veličine, alocira se cijeli blok i programeru se preko pokazivača vraća ukupna alocirana veličina.

Alokacija se obavlja pozivom funkcije *mem_alloc* koja kao parametre prima veličinu željenog bloka i pokazivač na lokaciju gdje će se upisati stvarna veličina, a vraća adresu alociranog bloka memorije.

Prototip funkcije mem_alloc

```
void *mem_alloc(size_t size, size_t *rsize)
```



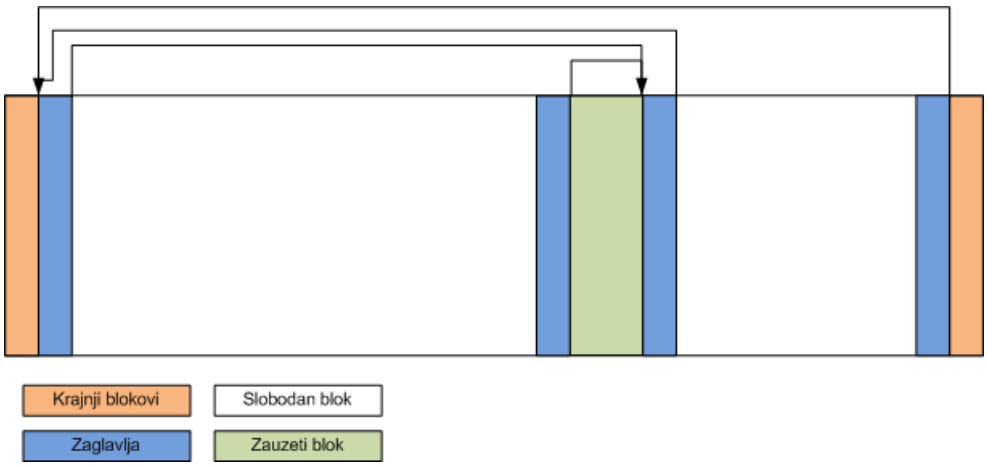
Sl. 3.6: Alokacija bloka u radnom spremniku

Oslobađanje memorije zahtijeva provjeru susjednih blokova kako bi se utvrdilo može li se trenutno oslobođeni blok spojiti s susjednima. Ukoliko se blok ne može spojiti, pokazivačem se prolazi kroz cijelu listu slobodnih blokova kako bi se otkrilo koja se dva bloka nalaze oko trenutno oslobođenog bloka. Zatim se taj blok umeće u listu slobodnih blokova, te je oslobađanje uspješno završeno.

Oslobađanje memorije obavlja se pozivom funkcije *mem_free* koja kao parametre prima adresu bloka i veličinu bloka

Prototip funkcije mem_free

```
void mem_free(void *adr, size_t size)
```



Sl. 3.7: Oslobađanje bloka memorije iz radnog spremnika

Na slici 3.7 vidi se primjer korištenja funkcije *mem_free* za oslobađanje bloka prikazanog na slici 3.6. Oslobođeni blok spojen je sa slobodnim blokom koji se nalazio nakon njega, te je stvoren jedan veliki slobodni blok. Zaglavlja novonastalog slobodnog bloka su promijenjena i ponovno zapisana na njegove krajeve. Pokazivači slobodnih blokova susjednog bloka su pomaknuti kako bi pokazivali na novonastalo zaglavlje.

3.2.2. Ograničenja algoritma s zaglavljima u slobodnim blokovima

Postoji više ograničenja i problema s ovim algoritmom, koji se moraju nekako kompenzirati u pozivajućem programu. Jedna od stvari koja se kompenzira u glavnom programu jest pamćenje veličine za svaki alocirani blok. Kako blok u sebi ne sadrži informacije o svojoj veličini, koja mora biti poznata da bi se oslobodila memorija, potrebno je funkciji za oslobađanje slobodnih blokova predati, osim adrese, i veličinu slobodnog bloka.

Potencijalno opasna situacija je kada se prilikom regularnog rada programa u zauzeti blok zapišu takve informacije da njegov početak ili kraj izgledaju kao zaglavlje slobodnog bloka. U tom slučaju, prilikom oslobađanja nekog od susjednih blokova moguće je da dođe do spajanja zauzetog bloka sa slobodnom i proglašavanjem cijelog novonastalog bloka slobodnim. Ovaj problem je realni nedostatak ovog načina upravljanja spremnikom, ali kako ne postoje dodatne informacije unutar zauzetog bloka, nemoguće je provjeriti radi li se stvarno o slobodnom bloku ili ne. Kako bismo pokušali izbjeći ovaj scenarij, koristimo dva različita magična broja za zaglavlje na početku i zaglavlje na kraju bloka.

3.3. Usporedba algoritama

Iako u zamisli algoritmi kreću od istog zahtjeva, vidljivo je da se algoritmi u mnogočemu razlikuju. Algoritam s odvojenim zaglavljima je jednostavniji za implementirati i manja je mogućnost pogreške u radu algoritma. Algoritam s zaglavljima samo u slobodnim blokovima zahtjeva više rada s vezanim listama i zbog toga je sporiji, ali je njegova unutrašnja fragmentacija praktično jednaka nuli, dok je kod algoritma s odvojenim zaglavljima ona fiksna.

Vanjska fragmentacija oba algoritma je jednaka, zbog toga što je za oba algoritma definirana minimalna veličina iskoristivog prostora slobodnog bloka, te ukoliko bi nakon razdvajanja bloka ona bila manja od definirane, algoritam bi zauzeo cijeli blok.

3.4. Moguća poboljšanja algoritama

Najveće poboljšanje prema vanjskoj fragmentaciji bila bi uporaba *best fit* algoritma za pronalazak slobodnog bloka. Time bismo smanjili vanjsku fragmentaciju zbog toga što bi se zauzimali blokovi koji su najbliže veličini zahtijevanog bloka. Loša strana je povećanje kompleksnosti pretrage liste slobodnih blokova, te samim time sporija alokacija radnog spremnika.

U slučaju korištenja *best fit* algoritma implementacija nečeg nalik na spremnike kod Doug Lea's malloc algoritma bi uvelike povećala brzinu pretraživanja slobodnih blokova. Broj spremnika bi se mogao dinamički određivati ovisno o veličini raspoloživog radnog spremnika, kako bi se dobile optimalne performanse.

U slučaju korištenja sustava s vrlo ograničenom memorijom, moguće je koristiti i jednostruko povezane liste za algoritam s odvojenim zaglavljima kako bi se smanjio dio spremnika koji se koristi za zaglavlja, a samim time ostalo više prostora za podatke.

4. Zaključak

U ovome radu opisano je nekoliko načina dinamičkog upravljanja spremnikom.

Prikazan je rad najčešće korištenih načina dinamičkog upravljanja spremnikom, te su prikazane prednosti i nedostatci svakog od njih. Svi opisani načini nisu zadovoljili potrebe upravljanja spremnikom bez korištenja zaglavlja. Iz tog razloga opisana su dva algoritma koji omogućavaju upravo takvo upravljanje. Uz njihov opis, priložena je i programska implementacija algoritma.

Dva načina su: upravljanje spremnikom uz odvojena zaglavlja i upravljanje spremnikom s zaglavljima samo u slobodnim blokovima. Prikazane su prednosti i nedostatci oba algoritma, te njihova područja primjene.

Na kraju rada opisane su mogućnosti poboljšanja za oba algoritma.

Literatura

[1] Ravenbrook, The Memory Management Reference, 2001, http://www.memorymanagement.org/ - 16.05.2011

[2] Doug Lea, A Memory Allocator by Doug Lea, travanj 2001., http://g.oswego.edu/dl/html/malloc.html – 03.06.2011

[3] Jason Evans, A Scalable Concurrent malloc(3) Implementation for FreeBSD, travanj 2006., people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf – 28.05.2011

[4] Robert Sedgewick, Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms, 3. izdanje, Addison-Wesley Professional, 2001

[5] Daniel P. Bovet, Marco Cesati Ph.D., Understanding the Linux Kernel, 3. izdanje, O'Reilly Media, 2005

5. Sažetak

Naslov: **Prilagodbe algoritma za dinamičko upravljanje spremnikom**

U ovome radu opisani su algoritmi za dinamičko upravljanje radnim spremnikom bez korištenja zaglavlja. Razvijena su dva algoritma: Algoritam s zaglavljima u zasebnom dijelu spremnika i algoritam s zaglavljima samo u slobodnim blokovima. Razvijeni algoritmi su opisani po komponentama: inicijalizacija radnog spremnika, te zauzimanje i oslobađanje blokova unutar spremnika. Algoritmi su u potpunosti implementirani koristeći C programski jezik.

Dodatno, prikazani su i neki konvencionalni sustavi za upravljanje radnim spremnikom, kao što je npr. dlmalloc. Konvencionalni sustavi i algoritmi korišteni su za usporedbu s razvijenim algoritmima, te su istaknute dobre i loše strane svih navedenih algoritama.

Ključne riječi: upravljanje spremnikom, memorija, dlmalloc, zaglavlja, first fit, best fit, C jezik

Summary

This paper describes algorithms for dynamic memory management without using headers. Two algorithms that do not use headers were developed: Algorithm with headers in separate part of the memory space and an algorithm with headers only in free blocks. These algorithms are described by components: Initialization of memory space and allocating/freeing of blocks inside that space. Algorithms are fully implemented using the C programming language.

Also, some conventional systems and algorithms for dynamic memory management are also described, e.g. dlmalloc. A few conventional systems are described and compared to developed algorithms. Both upsides and downsides of all algorithms are shown.

Keywords: memory management, memory, dlmalloc, headers, first fit, best fit, C language