SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1415

ALGORITMI UPRAVLJANJA SPREMNIČKIM PROSTOROM ZA UGRADBENA RAČUNALA

Hrvoje Knežević

Zagreb, lipanj 2010.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1415

ALGORITMI UPRAVLJANJA SPREMNIČKIM PROSTOROM ZA UGRADBENA RAČUNALA

Hrvoje Knežević

Zagreb, lipanj 2010.

SADRŽAJ

1. Uvod

Ugradbeno računalo je računalni sustav dizajniran za izvođenje jedne ili više usko određenih funkcija, često sa specifičnim zahtjevima u pogledu izvođenja u stvarnom vremenu. Ovakvo računalo ugrađeno je u neki uređaj kao dio cjelovitog sustava. Ugrađeni sustavi široko su rasprostranjeni i koriste se u mnogim uređajima.

Budući da se ugradbena računala projektiraju tako da izvršavaju neku specifičnu zadaću u uređaju, optimiziranje njihove veličine i performansi može uvelike smanjiti cijenu proizvoda i povećati pouzdanost i performansu cijelog sustava. Ugradbena računala koriste se u mnogim uređajima, od digitalnih satova i MP3 svirača, preko prometne signalizacije, termostata i kućanskih aparata, sve do uređaja koji vrše neku visoko preciznu operaciju, kao što je upravljanje elektranama.

Ugradbena računala imaju nekoliko specifičnih obilježja:

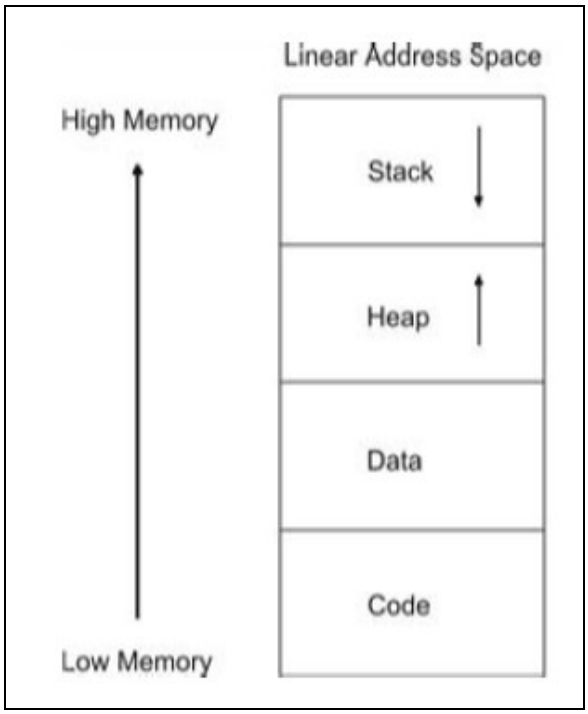
- Projektiraju se za neku specifičnu namjenu, te su rijetko izvedena kao računala opće namjene. Pri projektiranju ovakvih sustava potrebno je usmjeriti pažnju na pouzdanost i performanse sustava, istovremeno zadržavajući jednostavnost izvođenja i korištenja.
- Ugradbena računala mogu biti potpuno neovisni od ostatka sustava, ali često su dizajnirana kao dio neke veće cjeline, odnosno ovise o nekim drugim dijelovima sustava.
- Programske instrukcije ugradbenog računala obično se nalaze pohranjene kao ugrađeni program (*firmware*) u trajnoj (*read-only*) memoriji. Ugradbeni sustavi često imaju ograničene fizičke resurse: raspolažu malim količinama memorije, a mogu (i ne moraju) na raspolaganju imati jednostavne ulazno-izlazne jedinice, kao što su ekran ili tipkovnica.

Upravljanje spremničkim prostorom vrlo je važan segment u radu ugradbenog računala. Kod odabira načina upravljanja potrebno je voditi računa o brzini zauzimanja i oslobađanja blokova memorije, složenosti izvedbe, kao i o pojavi unutarnje i vanjske fragmentacije. Zbog vrlo ograničenih spremničkih resursa, upravljanje spremničkim prostorom mora biti izvedeno tako, da se kod velikog broja zahtjeva ne narušava performansa cijelog sustava.

2. Načini upravljanja spremničkim prostorom

2.1. Adresni prostor i vrste upravljanja spremničkim prostorom

Adresni prostor korisničkih programa obično se dijeli na četiri dijela: dio spremničkog prostora gdje je zapisan programski kod, dio sa podacima koje program koristi, dio koji program koristi **za vrijeme izvođenja** (*gomila, heap*) te *stog* na koji se **prije izvođenja** zapisuje kontekst programa, lokalne varijable, varijable stanja i sl. Primjer adresnog prostora korisničkog programa prikazan je na slici 1.



Slika 1: Korisnički adresni prostor

Upravljanje spremničkim prostorom dijeli se na tri osnovne vrste: statičko, automatsko i dinamičko upravljanje. Svaka vrsta upravljanja odvija se nad drugim dijelom adresnog prostora korisničkih programa.

Statičko upravljanje spremničkim prostorom postupak je koji se koristi kod razvoja neke aplikacije u fazi prije kompilacije. Pritom se određuju specifične lokacije, varijable i resursi koje aplikacija koristi i kojoj za vrijeme izvođenja ne može pristupiti niti jedan drugi programski modul.

Automatsko upravljanje spremničkim prostorom (upravljanje zasnovano na korištenju stoga - *stack-based management*) koristi operacijski sustav za upravljanje internim procesima i pozivima procedura. Kod upravljanja različitim procesima i procedurama, OS koristi dio prostora za pohranu statičkih i dinamičkih varijabli i konteksta programa, po principu LIFO (Last-in-First-out). Također, korisnički programi koriste stog za pohranu vlastitih lokalnih varijabli, deklariranih *prije* izvođenja programa.

Dinamičko upravljanje spremničkim prostorom (*dynamic memory management, heap-based memory management)* postupak je upravljanja dijelom spremničkog prostora koji koriste računalni programi za vrijeme svog izvođenja. Dinamički alociran spremnički prostor daje se na raspolaganje nekom programu, i za vrijeme rada tog programa nedostupan je drugim programima. Po završetku rada programa, dinamički alociran prostor oslobađa se i postaje dostupan drugim programima.

Na početku i tijekom izvođenja, korisnički programi dinamički šalju zahtjeve za memorijom. Uloga operacijskog sustava je da primi zahtjev, te sukladno sa veličinom zahtjeva i raspoloživosti slobodne memorije, pronađe i dodijeli blok odgovarajuće veličine programu. Na kraju rada, program oslobađa zauzeti blok, a operacijski sustav ga prema potrebi spaja sa susjednim slobodnim blokovima i omogućava ponovno korištenje oslobođenog dijela memorije.

2.2. Mehanizmi i svojstva dinamičkog upravljanja memorijom

Dinamičko upravljanje memorijom postupak je, dakle, upravljanja dijelom memorije koji koriste računalni programi za vrijeme svog izvođenja. Zadatak ispunjavanja zahtjeva za alokacijom memorije sastoji se od postupka pronalaženja neiskorištenog memorijskog bloka, čija veličina zadovoljava zahtjev programa. Memorijski blokovi se alociraju iz velikog bloka neiskorištene memorije (gomila, *heap*). Budući da se točna lokacija bloka koji se dodjeljuje ne zna unaprijed, memoriji se pristupa indirektno, putem reference.

Postoji nekoliko mehanizama dinamičkog upravljanja memorijom. Jedna od mogućih podjela (Masmano, 2008) je prema načinu izvedbe:

- jednostruko ili dvostruko povezana lista slobodnih blokova
- razdvojene liste slobodnih blokova iste veličine (segregated free lists)
- *buddy*-blokovi (specijalan slučaj lista slobodnih blokova)
- indeksiranje korištenjem naprednih podatkovnih struktura (indexed fits)
- korištenje bitmapa (bitmap fits)
- hibridni alokatori (kombinacija više navedenih mehanizama)

Korištenje *povezanih lista* jedan je od najosnovnijih mehanizama upravljanja. Slobodni blokovi memorije pretražuju se slijedno od nižih prema višim memorijskim lokacijama, a pokazivači na slobodne blokove pohranjeni su u jednostruko ili dvostruko povezanoj listi. Blokovi se mogu dodjeljivati na nekoliko različitih načina. Primjeri ovakvog mehanizma upravljanja su npr. *first-fit* (alocira se prvi blok memorije koji je veći ili jednak veličini zahtjeva) ili *best-fit* (alocira se onaj slobodni blok čija veličina najviše odgovara traženom zahtjevu).

Mehanizmi zasnovani na *razdvojenim listama slobodnih blokova* koriste liste slobodnih blokova. U svakoj od lista pohranjeni su pokazivači na blokove memorije unaprijed određene veličine ili raspona veličine. Kada se neki blok oslobodi, stavlja se u listu koja odgovara njegovoj veličini. Blokovi su tako logički, ali ne i fizički razdvojeni.

Buddy-block mehanizam (mehanizam susjednih blokova) poseban je slučaj korištenja razdvojenih lista slobodnih blokova. Ako sa H označimo ukupnu veličinu raspoložive memorije, postojat će $\log_2(H)$ lista slobodnih blokova, budući da će blokovi memorije biti raspodijeljeni po veličinama koje su potencije broja 2. Na taj način postiže se velika učinkovitost algoritma kod operacija dijeljenja i spajanja slobodnih blokova, ali se stvara velika unutarnja fragmentacija memorije. Postoje različite varijante ovog mehanizma, od kojih je najčešće korištena varijanta *binarnog buddy-sustava.* Kod ove izvedbe ukupna veličina memorije mora biti potencija broja 2. Kada se pojavi potreba za manjim blokom memorije, veći blokovi podijele se na dva manja susjedna bloka jednake veličine. Kada su oba susjedna bloka slobodna, spajaju se u veći blok. Ako se pojavi zahtjev za manjim blokom memorije, a nema niti jednog slobodnog bloka odgovarajuće

veličine, veći slobodni blok dijeli se jednom ili više puta sve dok se ne dobije slobodan blok odgovarajuće veličine.

Primjeri *indeksiranja memorije korištenjem naprednih podatkovnih struktura* su algoritmi koji koriste Adelson-Velskiijeva i Landinova stabla (*AVL trees*) ili *Fast-Fit* algoritam koji koristi kartezijanska stabla.

Korištenje *bitmapa* za upravljanje memorijskim prostorom omogućava veliku brzinu alociranja i oslobađanja memorije, ali uz kompliciraniju tehničku izvedbu, te imaju jediničnu složenost, `O(1)`.

Hibridni alokacijski mehanizmi kombiniraju različite mehanizme upravljanja kako bi se poboljšale neke karakteristike sustava (vrijeme alokacije/oslobađanja blokova, fragmentacija i sl.). Poznati primjer hibridnog mehanizma upravljanja je *Doug Lea alokacijski mehanizam (DLalloc)*, koji kombinira razdvojene liste, binarna stabla i dvostruko povezane liste. Svaki mehanizam koristi se za upravljanje jednim dijelom memorijskih blokova, ovisno o njihovoj veličini. Drugi primjer ovakvog mehanizma, koji koristi kombinaciju bitmapa i dvorazinskih razdvojenih lista slobodnih blokova je TLSF mehanizam. Složenost ovog algoritma je `O(1)` za operacije alociranja i oslobađanja blokova.

Od svih navedenih algoritama, u daljnjem tekstu temeljito će biti razrađeni samo oni mehanizmi koji će biti korišteni za ostvarenje sustava.

Kod izvedbe dinamičkog upravljanja memorijskim prostorom javljaju se određeni problemi na koje je potrebno obratiti pozornost, i koji određuju karakteristike ostvarenog sustava upravljanja:

- problemi kod ispunjavanja zahtjeva za alokacijom (ako u nekom trenutku nema raspoloživih blokova odgovarajuće veličine),
- pojava unutarnje i vanjske fragmentacije,
- problemi vezani uz složenost izvođenja algoritma i brzinu alokacije/oslobađanja bloka memorije,
- zauzimanje prostora za dodatne podatke koji se zapisuju u zaglavlje memorijskih blokova, a koje sustav koristi za upravljanje memorijom.

Unutarnja fragmentacija memorije javlja se kada se alocira veća količina memorije od one koju program zahtjeva za izvršavanje. Ova vrsta fragmentacije javlja se kod mehanizama koji dodjeljuju blokove unaprijed određene veličine. Problem unutarnje fragmentacije moguće je riješiti dodatnom podjelom neiskorištenog dijela bloka, kako bi se mogao iskoristiti i taj prostor, ali se pritom povećava složenost izvedbe. *Vanjska fragmentacija* javlja se za vrijeme rada sustava, kada se nakon nekog vremena alocira i oslobodi određen broj memorijskih blokova. Prilikom oslobađanja blokova između zauzetih dijelova memorije stvaraju se slobodni prostori koji se ne mogu spojiti sa drugim slobodnim blokovima. Ukoliko isti blokovi ponovno budu podijeljeni kako bi se jedan dio prostora dodijelio za neke manje zahtjeve, pojavljuju se fragmenti memorije koji ne mogu biti upotrijebljeni zbog vrlo male veličine, te dolazi do smanjenja iskoristivosti memorije. Postoje neki mehanizmi kojima se rješava ovaj problem, tako što neupotrebljivi fragmenti memorije sakupljaju i potom logički spajaju kako bi formirali veći, upotrebljivi blok memorije. Ova tehnika naziva se *sakupljanje otpada (garbage collection)*.

Neki mehanizmi upravljanja memorijom koriste podatke koji se pohranjuju u *zaglavlje* na početku bloka memorije. U zaglavlje bloka obično se stavljaju podaci o dostupnosti i veličini bloka, te pokazivači na prethodni ili slijedeći slobodni blok memorije. Ovo može predstavljati značajan problem u sustavima u kojima je česta pojava malih zahtjeva za memorijom, jer značajan dio memorije biva iskorišten za zaglavlja blokova, tj. ponekad zaglavlje može činiti čak polovicu ukupne veličine zauzetog bloka.

U računarskoj znanosti, *složenost algoritma* predstavlja mjeru učinkovitosti nekog algoritma da se izvrši u konačnom vremenu. Moguće je procijeniti složenost algoritma u najboljem, najgorem ili prosječnom slučaju. U razvoju nekog algoritma, najbolju povratnu informaciju daje *složenost u najgorem slučaju (worst-case complexity)*, jer može imati najveći utjecaj na ukupnu performansu sustava. Budući da je ponekad teško procijeniti najgoru situaciju u radu algoritma, ili postoji više situacija između kojih je teško odlučiti se koja je najgora moguća, moguće je u tim slučajevima koristiti za mjeru performanse i *složenost u lošem slučaju (bad-case scenario)*.

3. Ostvarenje sustava

3.1. Odabir mehanizma u ovisnosti o svojstvima ugradbenih sustava

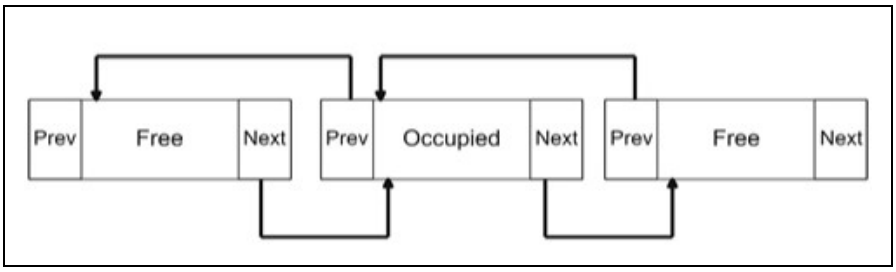
Kod odabira metode upravljanja spremničkim prostorom u ugradbenom računalnom sustavu, potrebno je voditi računa o određenim faktorima: zbog velike ograničenosti raspoloživog spremničkog prostora, potrebno je što je više moguće smanjiti fragmentaciju prostora (unutarnju i vanjsku), istovremeno pazeći da to ne dovede do smanjenja performanse ukupnog sustava. Također, cilj je da ostvarenje sustava istovremeno bude učinkovito i jednostavno izvedeno. Važno je obratiti pozornost i na veličinu zaglavlja bloka, jer u sustavu sa ograničenom količinom spremničkog prostora svako dodatno zauzimanje prostora može dovesti do neučinkovitosti ili usporavanja rada sustava.

Vodeći se po ovim kriterijima, za ostvarenje sustava upravljanja memorijom odabran je mehanizam susjednih blokova sa binarnom raspodjelom (*binary buddy-blocks*), koji koristi razdvojene liste slobodnih blokova memorije. Također, radi usporedbe performanse ostvaren je i mehanizam sa dvostruko povezanim listama (sequential fit), te napravljena modifikacija mehanizma susjednih blokova kako bi se povećala performansa kod oslobađanja slobodnih blokova. U nastavku teksta biti će detaljno objašnjeni navedeni mehanizmi i analizirane performanse tako ostvarenog sustava.

3.2. Sustav sa dvostruko povezanim listama (sequential fit)

Najosnovniji način upravljanja spremničkim prostorom je sustav koji koristi dvostruko povezane liste. Cijela memorija na početku rada tvori jedan memorijski blok. U zaglavlje svakog memorijskog bloka smještaju se podaci koji se koriste za upravljanje memorijom: pokazivač na prethodni memorijski blok, pokazivač na slijedeći memorijski blok, stanje bloka (zastavica koja opisuje je li blok trenutno zauzet ili slobodan za dodjelu), te veličina bloka. Na slici 2[1] prikazan je izgled povezane liste kojom su organizirani slobodni blokovi memorije. Ovakav način dodjele slobodnih blokova vrlo je sličan mehanizmu rada funkcija `malloc()` i `free()` programskog jezika C.

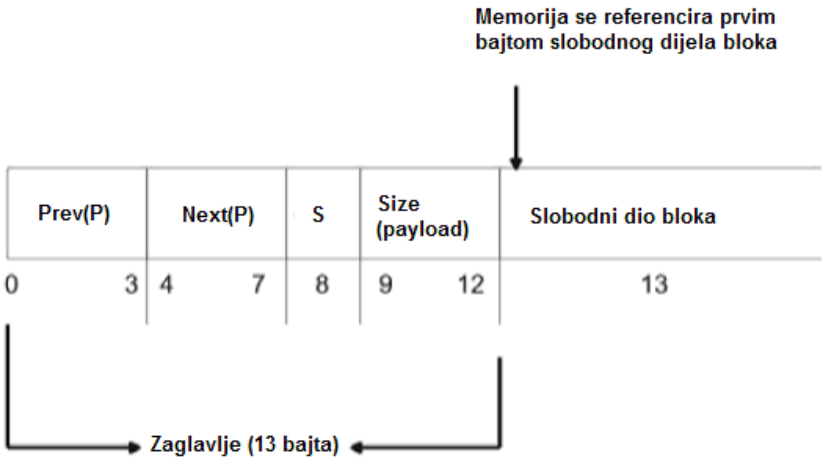
[1] Slika preuzeta iz B.Blunden: *Memory management: algorithms and implementation in C/C++*, 2003.



Slika 2 : Dvostruko povezana lista memorijskih blokova

U ovom ostvarenju navedenog mehanizma, zaglavlje je veličine 13 bajta. Pritom po 4 bajta zauzimaju pokazivači na susjedne blokove i zapis o veličini bloka, a jedan bajt otpada na zastavicu koja opisuje stanje bloka. Ostatak prostora čini iskoristivi dio bloka (payload) Izgled jednog takvog bloka memorije prikazan je na slici 3. Važno je napomenuti da će prvi blok memorije za pokazivač na prethodni blok imati vrijednost 0, kao i posljednji blok za vrijednost pokazivača na slijedeći blok memorije.

Kada se pojavi zahtjev za memorijom, traži se prvi blok odgovarajuće veličine. Ako je takav blok pronađen, alocira se dio bloka koji odgovara veličini zahtjeva, a od preostalog prostora formira se novi blok memorije i dodijeli mu se novo zaglavlje. Pritom će se promijeniti pokazivači unutar zaglavlja tako da novostvoreni blokovi pokazuju jedan na drugog, kao i podaci o veličini blokova i zastavice stanja. Ovdje je važno primijetiti da iskoristivi prostor bloka mora biti dovoljan za ispunjavanje memorijskog zahtjeva, ali pritom mora ostati dovoljno prostora da bi se formiralo zaglavlje slijedećeg bloka. Također, iskoristivi prostor novog bloka mora biti barem veličine zaglavlja, kako bi se taj blok mogao ponovno podijeliti prema potrebi. Ako sa `H` označimo zaglavlje bloka, sa `P` iskoristivi dio prostora u bloku, a sa `Z` veličinu zahtjeva, odabrani blok memorije mora biti veličine najmanje `P = Z + H + H`.



Slika 3: Format zaglavlja memorijskog bloka

Kod oslobađanja memorije, zauzetom bloku memorije promjeni se zastavica stanja, kako bi se označilo da je slobodan, te se po potrebi novooslobođeni blok spaja sa susjednim slobodnim blokovima. Ako niti jedan susjedni blok nije slobodan, ne radi se ništa. Ako je prethodni ili slijedeći blok u listi slobodan (ili oba), tada će oni biti spojeni sa novooslobođenim blokom.

Ovaj mehanizam upravljanja memorijom ostvaren je pomoću dvije osnovne funkcije, `alociraj()` i `oslobodi()`. Za ostvarenje zaglavlja koriste se pokazivači koji su definirani na početku koda:

```
#define PREV(i) (*((unsigned long*)(&memblok[i-13])))
#define NEXT(i) (*((unsigned long*)(&memblok[i-9])))
#define STATE(i) (*((unsigned char*)(&memblok[i-5])))
#define SIZE(i) (*((unsigned long*)(&memblok[i-4])))
#define FREE '0'
#define OCCUPIED '1'
#define MEMSTART 13
#define HEADER 13
```

Ovdje je važno primijetiti da se svaki blok memorije referencira po prvom bajtu iskoristivog prostora iza zaglavlja, pa se kao početak memorije ne uzima adresna lokacija 0, nego 13.

Funkcija `alociraj()` kao argument prima veličinu zahtjeva za memorijom, te pretražuje listu slobodnih blokova kako bi pronašla blok odgovarajuće veličine.

```
unsigned long alociraj(unsigned int prostor){
	unsigned long blok = MEMSTART, novi_blok, pom;
	int flag=0;
	while(flag==0){
	if((SIZE(blok) >= (prostor + HEADER)) && (STATE(blok)==FREE)){
		pom = NEXT(blok);
		novi_blok = blok+prostor+HEADER;
		STATE(blok) = OCCUPIED;
		STATE(novi_blok)=FREE;
		SIZE(novi_blok) = SIZE(blok)-prostor-HEADER;
		SIZE(blok)= prostor;
		NEXT(novi_blok)= NEXT(blok);
		NEXT(blok) = novi_blok;
		PREV(novi_blok)=blok;
		if (pom!=0) PREV(pom)=novi_blok;
		printf ("\n");
		printf ("PREV:%lu NEXT:%lu STATE:%c SIZE:%lu
		PAYLOAD_ADDRESS: %d\n\n",PREV(blok),NEXT(blok),
			STATE(blok),SIZE(blok), blok);
		ispis_mem();
		return blok;
	}
	if (NEXT(blok)==0) flag=1;
	blok = NEXT(blok);
	}
	return 0;
}
```

Ako se pronađe blok odgovarajuće veličine, od preostalog dijela bloka formirat će se novi blok, te će se promijeniti podaci u zaglavlju: zauzeti blok i novostvoreni blok pokazivat će jedan na drugoga, a novi blok na slijedeći blok memorije. Ako je uspješno pronađen blok odgovarajuće veličine, funkcija vraća adresu bloka. U protivnom vraća 0.

Oslobađanje memorijskog bloka ostvaruje se funkcijom `oslobodi()`. Funkcija kao argument prima adresu zauzetog bloka, oslobađa ga promjenom zastavice stanja, te ga po potrebi spaja sa susjednim slobodnim blokovima.

```
void oslobodi(unsigned long blok){
       unsigned long p=PREV(blok),n=NEXT(blok);
       unsigned long pom;
       STATE(blok)=FREE;

       if (p==0 && n!=0){
       if(STATE(n)==FREE){
                     printf("\nSpajanje bloka %lu sa slijedecim slobodnim blokom
                     na adresi %lu", blok, n);
              SIZE(blok)+= SIZE(n) + HEADER;
              NEXT(blok)=NEXT(n);
              if (NEXT(n)!=0){
                     pom=NEXT(n);
                     PREV(pom)=blok;
              }
       }
       }
       else if (n==0){
       if (STATE(p)==FREE){
                     printf("\nSpajanje bloka %lu sa prethodnim slobodnim blokom
                     na adresi %lu", blok, p);
              SIZE(p) = SIZE(p) + SIZE(blok) + HEADER;
              NEXT(p)=NEXT(blok);
       }
       }
       else if(STATE(p)==FREE && STATE(n)==OCCUPIED){
                     printf("\nSpajanje bloka %lu sa prethodnim slobodnim blokom
                     na adresi %lu", blok, p);
              SIZE(p)+= SIZE(blok) + HEADER;
              NEXT(p)=NEXT(blok);
              PREV(n)=p;
       }
       else if(STATE(p)== OCCUPIED && STATE(n)==FREE){
                     printf("\nSpajanje bloka %lu sa slijedecim slobodnim blokom
                     na adresi %lu", blok, n);
              SIZE(blok)+= SIZE(n) + HEADER;
              NEXT(blok)=NEXT(n);
              pom=NEXT(n);
              if (pom !=0) PREV(pom)=blok;
       }
       else if(STATE(p)==FREE && STATE(n)==FREE){
                     printf("\nSpajanje bloka %lu sa prethodnim slobodnim blokom
                     na adresi %lu\n",blok,p);
              printf(" i slijedecim slobodnim blokom na adresi %lu",n);
              SIZE(p)+= SIZE(blok) + SIZE(n) + HEADER + HEADER;
              NEXT(p)=NEXT(n);
              pom=NEXT(n);
              if (pom != 0) PREV(pom)=p;
       }
}
```

Nakon oslobađanja bloka, radi se provjera jesu li susjedni blokovi slobodni, te se ovisno od njihovih zastavica stanja vrši spajanje blokova. Četiri su moguća slučaja: niti jedan susjedni blok nije slobodan, slobodan je prethodni ili slijedeći blok, ili su slobodna oba. Također, provjerava se je li pokazivač na prethodni blok postavljen na 0 (kako bi se ustanovilo radi li se o bloku na početku memorije), kao i je li pokazivač na slijedeći blok postavljen na 0 (radi se o bloku na kraju memorije).

Unutar funkcije `alociraj()` poziva se dodatna funkcija `ispis_mem()` koja prikazuje trenutno stanje zauzeća memorijskih blokova.

Memorijski prostor nad kojim se vrši upravljanje na početku rada programa alociran je funkcijom `malloc()` i po završetku rada oslobađa se funkcijom `free()`, kako bi se omogućio rad ostvarenog sustava na različitim platformama.

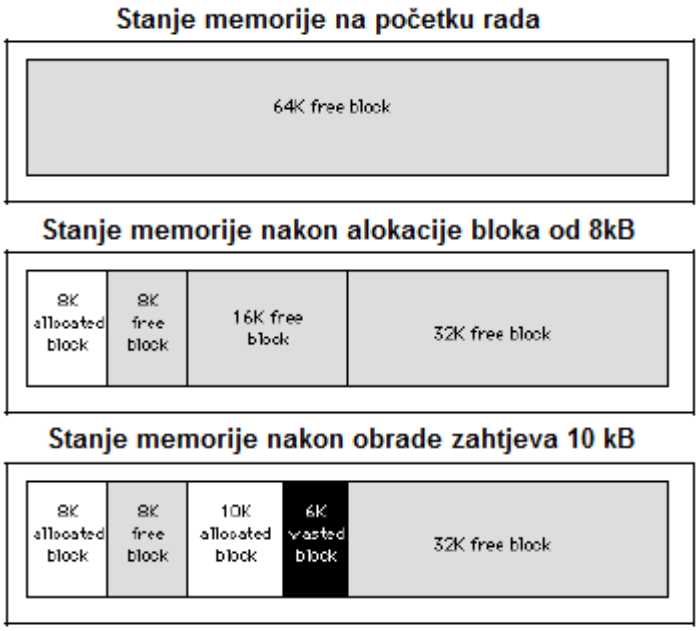
3.3. Sustav susjednih blokova sa binarnom raspodjelom (binary *buddy*-blocks)

Mehanizam rada ovog algoritma objašnjen je djelomično u prethodnom poglavlju. Za ostvarenje sustava koriste se liste slobodnih blokova. Blokovi su u listama smješteni prema unaprijed određenoj veličini, koja je potencija broja 2. Liste slobodnih blokova formiraju se za unaprijed određene veličine blokova. Pritom je maksimalna veličina bloka određena količinom memorije koju sustav ima na raspolaganju, a minimalna veličina bloka postavlja se na neku vrijednost koja je dovoljno velika da zadovolji najmanji mogući zahtjev programa, npr. 16 bajta.

Kada se pojavi zahtjev za memorijom, pretražuju se liste slobodnih blokova: pronađe se lista sa blokovima odgovarajuće veličine, a zatim se putuje po listi i traži slobodni blok. Kada blok bude pronađen, označava se kao zauzet i njegova adresa dodjeljuje se programu koji je postavio zahtjev. Ako u odgovarajućoj listi nema slobodnih blokova, pronalazi se prvi veći slobodni blok i dijeli se na dva manja bloka jednake veličine. Postupak se ponavlja dok se ne dobije slobodni blok odgovarajuće veličine.

Kada program završi s radom, oslobađa se zauzeti blok: pritom se provjerava je li njemu susjedni blok (*buddy*) također slobodan. Ako je to slučaj, tada se ta dva bloka spajaju u veći blok i smještaju u odgovarajuću listu. Postupak se ponavlja dok je god moguće spojiti susjedne blokove u veći blok.

Prednost ovako ostvarenog sustava je *nepostojanje zaglavlja* u blokovima memorije: liste slobodnih blokova sadrže sve potrebne podatke o stanju i veličini bloka, te adrese slobodnih blokova. Također, vrijeme traženja slobodnog bloka manje je nego kod slijednog pretraživanja memorije, jer je dovoljno samo pronaći listu slobodnih blokova i u njoj provjeriti sadrži li slobodni blok odgovarajuće veličine. Nedostatak je to, što se zbog unaprijed određene veličine blokova javlja značajna *unutarnja fragmentacija*, jer se veličina zahtjeva zaokružuje na veličinu dodijeljenog bloka, pa dio prostora unutar bloka ostaje neiskorišten. Slika 4 prikazuje mehanizam rada ovakvog sustava na primjeru memorije veličine 64 KB, kao i primjer pojave unutarnje fragmentacije.



Slika 4: Princip rada *buddy*-block algoritma i pojava unutarnje fragmentacije

Na početku rada pokazivači na memorijske blokove smještaju se u liste slobodnih blokova koje imaju slijedeću strukturu:

```
struct blokovi{
       int slobodno;
       char *p;
       struct blokovi *l_buddy;
       struct blokovi *r_buddy;
}blok;

struct free_list{
       unsigned short int size;
       unsigned short int slobodno;
       struct blokovi b[MAX_BLOK];
} lista[MAX_LISTA];
```

Svaka lista sadrži dodatne podatke: veličinu blokova koji se nalaze u listi, te njihovu veličinu. Za svaki blok u listi sadržani su slijedeći podaci: stanje bloka, pokazivač na adresu bloka, te pokazivači na zapise u slijedećoj listi, a koji sadrže podatke o blokovima koji mogu biti stvoreni od trenutnog bloka. Dakle, ukoliko se pojavi zahtjev za manjim blokom, a nema trenutno raspoloživih blokova odgovarajuće veličine, traži se prvi veći blok. Ako je pronađen takav blok, promijenit će se zastavica dostupnosti tog bloka, a u nižoj listi postavit će se zastavice novodobivenih manjih blokova. Također, promijenit će se i broj raspoloživih blokova u obje liste. Postupak je obrnut u slučaju oslobađanja manjeg bloka: ako je susjedni blok slobodan, oni će biti označeni kao nedostupni, a u višoj listi veći blok nastao spajanjem dva manja bloka označit će se dostupnim.

Za ostvarenje ovog sustava koriste se tri osnovne funkcije: `alociraj()`, `oslobodi()` i `split()`. Funkcija `alociraj()` prima kao argument veličinu zahtjeva za memorijom i pokazivač na strukturu koja sadrži liste slobodnih blokova:

```
char *alociraj(struct free_list lista[], int prostor){

       int i=0,j=0, blok_size=0;
       char *adresa;

       while (prostor<lista[i].size && i<list_num) i++;
       if(prostor>lista[i].size) i--;
       blok_size=lista[i].size;

       if (lista[i].slobodno==0 ){
              if(split(lista, i)==0) return NULL;
       }
       lista[i].slobodno--;
       while(lista[i].b[j].slobodno==0)j++;
       adresa=lista[i].b[j].p;
       lista[i].b[j].slobodno=0;
       printf("Zahtjev za memorijom %dB, odabran je blok velicine %dB na
       adresi %d\n\n", br, prostor, blok_size, adresa - addr);
       ispis_mem(lista);

       return adresa;
}
```

Prvo se pronalazi lista sa blokovima čija veličina odgovara memorijskom zahtjevu. Zatim se provjerava ima li u listi dostupnih blokova. Ako trenutno nema dostupnih blokova, poziva se funkcija `split()` koja dijeli veće blokove. Ova funkcija vraća vrijednost 0 ako nema niti jednog slobodnog bloka koji je moguće podijeliti, i u tom slučaju funkcija alociraj vraća vrijednost `null`. U protivnom, ako postoji slobodan blok odgovarajuće veličine ili je odgovarajući blok dobiven dijeljenjem većeg bloka, funkcija umanjuje vrijednost brojača slobodnih blokova, postavlja zastavicu stanja bloka i vraća

adresu dodijeljenog bloka. Prije vraćanja adrese bloka poziva se funkcija `ispis_mem()` koja ispisuje podatke o slobodnim blokovima memorije.

Funkcija `split()` kao argument prima indeks liste koja sadrži blokove čija veličina odgovara veličini zahtjeva za memorijom. Provjeravaju se liste koje sadrže veće blokove, i ukoliko se pronađe slobodan blok, on se dijeli sve dok se ne dobije blok odgovarajuće veličine. Ako je operacija podjele blokova uspješno napravljena, funkcija vraća vrijednost 1, u suprotnom vraća vrijednost 0.

```
int split(struct free_list lista[], int i){
	int blok_size = lista[i].size;
	int x=i, n=i, j=0;
	int flag=0;

	while(lista[n].slobodno==0 && n>0) n--;
	if (lista[n].slobodno==0) return 0;
	else{
		while (lista[x].slobodno==0){
			lista[n].slobodno--;
			lista[n+1].slobodno+=2;
			while(lista[n].b[j].slobodno==0)j++;
			lista[n].b[j].slobodno=0;
			lista[n].b[j].l_buddy->slobodno=1;
			lista[n].b[j].r_buddy->slobodno=1;
			n++;
		}
	}
	return 1;
}
```

Funkcija `oslobodi()` prima kao argument adresu i veličinu zauzetog bloka. Prvo se pronalazi lista sa blokovima odgovarajuće veličine, a zatim se pronalazi zauzeti blok unutar liste. Mijenja se zastavica dostupnosti i povećava se brojač slobodnih blokova u listi za 1.

Nakon oslobađanja zauzetog bloka, provjerava se postoje li slobodni susjedni blokovi koje je moguće spojiti. Ako postoje, spajaju se u veći blok. Postupak se ponavlja dok god u nekoj listi ima slobodnih blokova koje je moguće spojiti.

```
void oslobodi(struct free_list lista[], char *adresa, int prostor){
	int i=0,j=0,blok_size=0, flag=1;

	while (prostor<lista[i].size && i<list_num) i++;
	if(prostor>lista[i].size) i--;
	blok_size=lista[i].size;
	while(lista[i].b[j].p != adresa) j++;
	lista[i].b[j].slobodno=1;
	lista[i].slobodno++;
	printf("\nOslobodjen blok velicine %dB na adresi %d\n",
		blok_size, adresa - addr);
	ispis_mem(lista);
	i=list_num-2;

	while(i>=0){
		for(j=0;j<=(max_block/lista[i].size);j++){
			if(lista[i].b[j].l_buddy->slobodno==1 &&
			lista[i].b[j].r_buddy->slobodno==1){
				lista[i].b[j].l_buddy->slobodno=0;
				lista[i].b[j].r_buddy->slobodno=0;
				lista[i].b[j].slobodno=1;
				lista[i+1].slobodno-=2;
				lista[i].slobodno++;
				printf("\nSpojena su 2 bloka velicine %d u novi blok
				velicine %d!\n", lista[i+1].size, lista[i].size);
				ispis_mem(lista);
			}
		}
	i--;
	}
}
```

Spremnički prostor nad kojim se vrši upravljanje na početku rada programa alociran je funkcijom `malloc()` i po završetku rada oslobađa se funkcijom `free()`, kako bi se omogućilo korištenje ostvarenog sustavana različitim platformama.

4. Analiza svojstava ostvarenih sustava

Ostvareni sustavi upravljanja memorijom analizirani su prema dva bitna svojstva: složenost algoritma u slučaju najgoreg scenarija, te ukupnoj brzini izvođenja velikog broja alokacija i oslobađanja blokova memorije. Također, bit će komentirana količina memorijske fragmentacije i prostorna lokalnost zahtjeva. Konačno, bit će prikazano jedno od mogućih poboljšanja algoritma kako bi se ubrzao rad sustava izvedenog razdvojenim listama slobodnih blokova.

4.1. Složenosti algoritama

Za analizu složenosti ostvarenih sustava uzimaju se najgori slučajevi kod izvršavanja operacija alokacije i oslobađanja blokova . Kod prikaza složenosti koristit će se slijedeće oznake: H označava ukupnu veličinu memorije, a M veličinu bloka memorije koji zadovoljava zahtjev.

Kod sustava ostvarenog **dvostruko povezanom listom**, najgora situacija kod alociranja blokova je u situaciji kada lista ima maksimalnu dužinu, a blok odgovarajuće veličine nalazi se na kraju liste. Ovakav slučaj moguć je ako se pojavljuju zahtjevi za blokovima minimalne veličine sve dok ne bude iscrpljena sva slobodna memorija, a zatim se oslobodi blok (ili nekoliko blokova koji se spajaju) na kraju liste. U tom slučaju morat će se provjeriti svi blokovi od početka do kraja liste, te će složenost ove operacije je $O\left(\frac{H}{2M}\right)$.

Postupak oslobađanja zauzetog bloka svodi se na promjenu zaglavlja i spajanje sa susjednim blokovima. Pritom je najgora moguća situacija kada su slobodna oba susjedna bloka, pa je potrebno spojiti ukupno 3 memorijska bloka. Složenost ovog dijela algoritma je $O(1)$.

Kod ostvarenja ***buddy*-block** algoritmom, najgori scenarij kod alokacije slobodnog bloka odvija se na početku rada sustava, ako dođe zahtjev za blokom najmanje veličine. U ovom slučaju početni blok memorije morat će se podijeliti maksimalni broj puta kako bi se dobio blok odgovarajuće veličine. Složenost ovog postupka je $O\left(\log_2 \frac{H}{M}\right)$.

Kod operacije oslobađanja bloka memorije, najgori slučaj simetričan je najgorem slučaju kod alokacije bloka. To je situacija kada je alociran samo jedan blok minimalne veličine. Kod oslobađanja tog bloka potrebno je spojiti sve blokove iz svih lista. Cijena ovog dijela algoritma je $O\left(\log_2 \frac{H}{M}\right)$.

U kasnijem dijelu ovog poglavlja bit će objašnjeno poboljšanje potonjeg algoritma, kojim se postiže da se složenost oslobađanja bloka svede na O(1), a pritom se ne promijeni složenost kod operacije alociranja bloka.

4.2. Fragmentacija memorije i specifičnosti u adresiranju

Analizom funkcionalnosti ostvarenih algoritama može se zaključiti njihovo ponašanje i stanje memorije nakon dužeg vremena rada.

Kod ostvarenja dvostrukom povezanom listom, postoji nekoliko faktora koji uzrokuju nepotrebne gubitke raspoloživog prostora. Prvo, zaglavlja blokova koja se koriste u radu algoritma zauzimaju određeni prostor. U slučaju sustava u kojem se pojavljuju samo veliki zahtjevi i ima dovoljno memorije, gubitak od 13 bajta na zaglavlje ne predstavlja velik problem. Međutim, u ugradbenom računalu, koje ima vrlo ograničene resurse, taj gubitak je itekako značajan. Nadalje, ako se radi o sustavu u kojem je velik broj zahtjeva u rasponu od npr. 16-64 bajta, vidljivo je da će na zaglavlje otpasti vrlo velik dio bloka. Za zahtjev od 16 bajta bit će potrebno zauzeti blok od 29 bajta, što je gotovo dvostruko od potrebne količine prostora.

Drugi problem kod ove izvedbe je vanjska fragmentacija memorije. Dok unutarnja fragmentacija ne postoji (jer se uvijek zauzima točna količina prostora uz prostor potreban za zaglavlje), nakon dužeg vremena rada moguće je da se oslobađanjem manjih blokova memorije stvori velik broj malih slobodnih fragmenata memorije. Problem je što se ti blokovi ne mogu spojiti, jer je zbog poredanosti adresa moguće spojiti samo susjedne blokove memorije. U slučaju da dođe veći zahtjev za memorijom, on neće moći biti ispunjen, iako je ukupna veličina svih slobodnih blokova dovoljna da zadovolji potrebe zahtjeva. Postoje neki mehanizmi kojima je moguće spojiti ove fragmente u veći upotrebljivi blok, ali se pritom pojavljuju veliki gubitci u ukupnoj performansi sustava i povećava se algoritamska složenost, koja je i bez dodatnih preinaka vrlo velika.

Kod *buddy*-block izvedbe nema gubitaka prostora uzrokovanih zaglavljima jer ona ne postoje, ali se javlja značajan problem unutarnje fragmentacije. Ako uzmemo da je veličina nekog bloka `n` bajta, a pojavi se zahtjev za točno `n+1` bajta memorije, tada je potrebno dodijeliti blok veličine `2n` kako bi se zadovoljio zahtjev. Pritom se gubi gotovo dvostruka količina memorije od one koja je zahtijevana od strane programa. Postoje neki postupci kojima se rješava problem unutarnje fragmentacije tako da se neiskorišteni dio dodijeljenog bloka ponovno podijeli i omogući za korištenje, ali pritom se ponovno javlja problem pada performanse i povećanja algoritamske složenosti.

Neki hibridni alokacijski mehanizmi koriste kombinaciju više tehnika, od kojih se svaka koristi za upravljanje zahtjevima različitih veličina. Omjer složenosti i fragmentacije je kod njih zadovoljavajući, ali su ovakvi mehanizmi primjenjivi samo u sustavima gdje je dostupna veća količina memorije, pa je moguće memoriju podijeliti na zasebne cjeline nad kojima se koriste različiti mehanizmi. Pritom se blokovi memorije dodjeljuju iz neke od cjelina ovisno o veličini memorijskog zahtjeva.

U slučaju dvostruko povezane liste, nema uzorka ponašanja u duljem radu: sustav dodjeljuje adrese blokova koje su mu u tom trenutku dostupne i zadovoljavaju zahtjev, tako da se ne može govoriti o određenom skupu adresa koje se ponovno dodjeljuju. Kod izvedbe *buddy*-blokovima, ovisno od sustava u kojem se koristi, moguće je pratiti određene memorijske lokacije, jer se zbog pravilne raspodjele veličina blokova neke adrese na početku memorijskog spremnika često ponovno koriste. *Buddy*-block mehanizam ima još jedno zanimljivo i korisno svojstvo. Naime, zbog toga što se blokovi dijele po potencijama broja 2, može se primijetiti da će se početne adrese susjednih blokova (lijevog i desnog *buddyja*) uvijek razlikovati u točno jednom bitu adrese, i to na takav način, da će lijevi *buddy* uvijek imati adresu koja sadrži samo jednu jedinicu, a adresa desnog *buddyja* imat će dodatnu jedinicu na prvom nižem bitu. Ovo svojstvo može se iskoristiti za brzo pronalaženje početka susjednog bloka.

Zbog svih ovih razloga, mehanizam ostvaren razdvojenim listama slobodnih blokova (*buddy*-block) čini se kao učinkovita izvedba sustava upravljanja memorijom u ugradbenim računalima. Potvrda ove teze pokušat će se pronaći analizom rezultata testa performanse, u kojem će se ispitivati brzina izvođenja pri uzastopnoj alokaciji i oslobađanju velikog broja blokova.

4.3. Modifikacija *buddy*-block algoritma

U nastojanju da se smanji složenost algoritma i ukupno vrijeme rada *buddy-block* mehanizma, ostvarena je jedna od mogućih modifikacija ovog sustava upravljanja. Naime, u ugradbenom sustavu u kojem je većina zahtjeva za memorijom vrlo mala, moguće je primijeniti izmijenjenu varijantu algoritma, koja će vršiti spajanje slobodnih blokova samo kada se pojavi isključiva potreba za time. Ovakva izvedba sustava upravljanja memorijom ponekad se naziva ***lazy buddy*** algoritam. Ovakav pristup značajno povećava performanse sustava, jer se posao oslobađanja zauzetog bloka svodi na pronalaženje bloka u listi i promjenu zastavice stanja. Složenost tog dijela algoritma za najgori slučaj je O(1).

Istovremeno, kod alociranja memorije ne mijenja se složenost algoritma, jer će se (pod pretpostavkom da je alokacija moguća, tj. da postoji dovoljna količina slobodnog prostora da je moguće dodijeliti memoriju programu) posao spajanja blokova raditi isključivo ako nema dostupnog bloka odgovarajuće veličine. Redoslijed operacija kod alokacije je slijedeći: prvo se traži odgovarajući slobodan blok; ako nema dostupnog bloka, pokušava se spojiti manje slobodne blokove kako bi se formirao blok odgovarajuće veličine. Tek ako ni to nije moguće, pokušat će se podijeliti veći blokovi kako bi se dobio blok odgovarajuće veličine. Pritom najgori scenarij i dalje ostaje isti kao i kod osnovne izvedbe *buddy*-block algoritma, a i složenost ovog dijela algoritma se ne mijenja, $O\left(\log_2 \frac{H}{M}\right)$.

Funkcije koje se koriste u ovoj izvedbi ponešto se razlikuju u odnosu na funkcije korištene u osnovnoj varijanti algoritma. Dodana je nova funkcija, `join()`, koja je po načinu rada ista kao i drugi dio funkcije `oslobodi()` u osnovnoj izvedbi, ali se sada ona poziva unutar funkcije `alociraj()`.

Funkcija `alociraj()` kao argumente prima veličinu memorijskog zahtjeva i pokazivač na strukturu u kojoj se nalaze liste slobodnih blokova. Kada se pronađe odgovarajuća lista slobodnih blokova, provjerava se ima li u njoj slobodnih blokova. Ako nema, poziva se funkcija `join()` koja pokušava spojiti manje slobodne blokove kako bi se stvorio blok odgovarajuće veličine. Ako i nakon toga ne postoji odgovarajući blok, poziva se funkcija `split()` koja pokušava pronaći veći blok i podijeliti ga tako da se dobije blok odgovarajuće veličine. Ako postoji dostupan blok, vraća se adresa bloka, u protivnom funkcija `alociraj()` vraća vrijednost `null`.

```
char *alociraj(struct free_list lista[], int prostor){
	int i=0,j=0, blok_size=0;
	char *adresa;

	while (prostor<lista[i].size && i<list_num) i++;
	if(prostor>lista[i].size) i--;
	blok_size=lista[i].size;

	if (lista[i].slobodno==0){
		join(lista, blok_size);
		if(lista[i].slobodno==0){
			if(split(lista, i)==0){
				return NULL;
			}
		}
	}
	lista[i].slobodno--;
	while(lista[i].b[j].slobodno==0)j++;
	adresa=lista[i].b[j].p;
	lista[i].b[j].slobodno=0;
	return adresa;
}
```

Funkcija `join()` provjerava postoje li slobodni blokovi manji od tražene veličine bloka, a koje je moguće spojiti kako bi se dobio blok odgovarajuće veličine. Kao argumente prima pokazivač na strukturu koja sadrži liste slobodnih blokova i veličinu traženog bloka memorije.

```
void join(struct free_list lista[], int blok_size){
	int i = list_num-2;
	int j = 0;

	while(i>=0 && (lista[i].size<=blok_size)){
		for(j=0;j<=(max_block/lista[i].size);j++){
			if(lista[i].b[j].l_buddy->slobodno==1 &&
			   lista[i].b[j].r_buddy->slobodno==1){
				lista[i].b[j].l_buddy->slobodno=0;
				lista[i].b[j].r_buddy->slobodno=0;
				lista[i].b[j].slobodno=1;
				lista[i+1].slobodno-=2;
				lista[i].slobodno++;
			}
		}
		i--;
	}
}
```

Funkcija `split()` kao argument prima indeks liste koja treba sadržavati slobodan blok. Provjeravaju se liste koje sadrže veće blokove, kako bi se pronašao slobodan blok koji je moguće podijeliti da bi se dobio blok odgovarajuće veličine. Ako ova operacija bude uspješno provedena, funkcija vraća vrijednost 1. Ako nema slobodnih većih blokova, povratna vrijednost je 0.

```
int split(struct free_list lista[], int i){
	int blok_size = lista[i].size;
	int x=i, n=i, j=0;

	while(lista[n].slobodno==0 && n>0) n--;
	if (lista[n].slobodno==0) return 0;
	else{
		while (lista[x].slobodno==0){
			lista[n].slobodno--;
			lista[n+1].slobodno+=2;
			while(lista[n].b[j].slobodno==0)j++;
			lista[n].b[j].slobodno=0;
			lista[n].b[j].l_buddy->slobodno=1;
			lista[n].b[j].r_buddy->slobodno=1;
			n++;
		}
	}
	return 1;
}
```

Konačno, oslobađanje zauzetog bloka vrši se unutar funkcije `oslobodi()`. Ovaj postupak svodi se na pronalaženje bloka u odgovarajućoj listi i postavljanje zastavice dostupnosti.

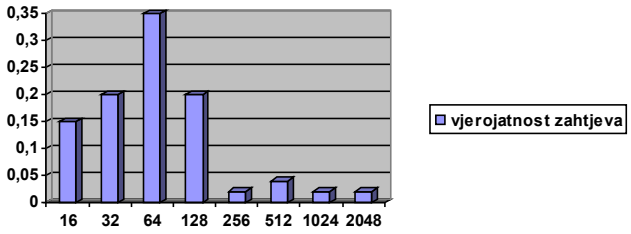
```
void oslobodi(struct free_list lista[], char *adresa, int prostor){
	int i=0,j=0,blok_size=0;

	while (prostor<lista[i].size && i<list_num) i++;
	if(prostor>lista[i].size) i--;
	blok_size=lista[i].size;

	while(lista[i].b[j].p != adresa) j++;
	lista[i].b[j].slobodno=1;
	lista[i].slobodno++;
}
```

Detaljno komentirani programski kodovi svih ostvarenih sustava za upravljanje memorijskim prostorom dostupni su na mediju priloženom uz rad.

4.4. Testiranje performanse

Ranije u poglavlju zaključeno je da je *buddy*-block izvedba sustava primjerenija za korištenje u ugradbenom računalnom sustavu. Kako bi se pokušala potvrditi ova teza ostvaren je test performanse, u kojem se promatra brzina izvođenja pojedinog algoritma. Test je zamišljen tako, da se prvo izvrši velik broj alokacija blokova memorije, a nakon toga se oslobode svi alocirani blokovi, i pritom se mjeri vrijeme potrebno za izvršavanje tog zadatka. Test se ponavlja deset puta, a rezultat je izražen prosječnim vremenom izvršavanja svih testova. Ovaj test primijenjen je na oba ostvarena sustava, kao i na modifikaciji *buddy*-block sustava, sa ciljem da se analizira međusobni odnos brzina, kao i doprinos modifikacije osnovnog algoritma ukupnom poboljšanju performanse.

Kako bi vrsta zahtjeva u testu bila što vjerodostojnija i bliža radu stvarnog sustava, ostvareno je generiranje memorijskih zahtjeva u rasponu od 16-2048 bajta, i to po slijedećoj razdiobi vjerojatnosti zahtjeva:



Slika 5: Razdioba zahtjeva za memorijom

Testiranje je rađeno na primjeru memorije od 32 kB (kako bi se omogućila uspješna alokacija većeg dijela zahtjeva), sa 1000 uzastopnih zahtjeva za alokacijom. Rezultati testiranja su izraženi u milisekundama i mikrosekundama. Uz rezultate testiranja prikazane su i složenosti alokacije i oslobađanja za pojedine izvedbe.

Tablica 1: Test performanse

	Test performanse (100 alokacija i oslobađanja)	Složenost (alokacija)	Složenost (oslobađanje)
Dvostruko povezana lista	92 μs	$O\left(\frac{H}{2M}\right)$	$O(1)$
Buddy-blocks	6,8 ms	$O\left(\log_2 \frac{H}{M}\right)$	$O\left(\log_2 \frac{H}{M}\right)$
Lazy *buddy*-blocks	2,04 ms	$O\left(\log_2 \frac{H}{M}\right)$	$O(1)$

Rezultati testa donekle su neočekivani: dvostruko povezana lista pokazuje se za nekoliko redova veličine brža od *buddy*-block mehanizma. Ipak, ovaj fenomen može se lako objasniti nesavršenošću testa. Naime, test je napravljen tako da gotovo svi zahtjevi za memorijom budu odmah ispunjeni. To znači da će u gotovo svim prolascima kroz listu ovakav sustav pronaći slobodan prostor odmah iza posljednjeg zauzetog bloka. S druge strane, *buddy*-block sustav izvršavat će mnoštvo dijeljenja i spajanja blokova, u pokušaju da dodijeli optimalan prostor za svaki zahtjev. Tu se zapravo može vidjeti prava snaga *buddy*-block algoritma: kada bi ovaj algoritam bio ostvaren na stvarnom sustavu, koji radi u stvarnom vremenu, nakon dužeg rada i velikog broja naizmjeničnih alokacija i oslobađanja pokazao bi se puno učinkovitijim u pronalaženju slobodnih blokova, dok bi

sustav sa dvostrukom listom drastično usporio zbog ogromne vanjske fragmentacije koja bi se pojavila nakon dužeg rada.

Što se tiče drugog dijela testa, ovdje su očekivanja ispala opravdana: modifikacija *buddy*-block algoritma izvršava se gotovo 3,5 puta brže od svoje osnovne varijante, jer se ne gubi vrijeme na spajanje blokova nakon svakog oslobađanja.

Korištenjem naprednijih testova za navedene sustave dobivaju se rezultati koji pokazuju stvarne odnose brzina rada. Kako je pokazano u nekim znanstvenim publikacijama (Masmano,2008), *buddy*-block sustav višestruko nadilazi dvostruko povezane liste u performansi. U tablici 2[2] prikazani su rezultati testiranja algoritama razvijenih u *frameworku* **µlayer,** a test je napravljen tako, da se svaki od ostvarenih sustava dovede u stanje najgoreg mogućeg slučaja, te se zatim izmjeri vrijeme rada svakog algoritma. Kao mjera rada uzeti su broj procesorskih instrukcija i broj ciklusa procesora tijekom izvršavanja najgoreg slučaja.

Tablica 2: Test performanse algoritama za najgori slučaj

		Broj instrukcija procesora	Broj procesorskih ciklusa
First-fit	Alokacija	81995	1613263
	Oslobađanje	115	1241
Buddy-blocks	Alokacija	1403	3898
	Oslobađanje	1379	4774

[2] Rezultati mjerenja preuzeti iz : M.Masmano i dr, *A Constant-Time Dynamic Storage Allocator for Real-Time Systems,*2008

4.5. Završne napomene

Komentirani programski kodovi svih ostvarenih sustava i testova performansi nalaze se na mediju priloženom uz rad. Rezultati testova performanse ovise od platforme na kojoj se izvršavaju, ali omjeri dobivenih vremena ostaju jednaki. Rezultati mjerenja performanse u tekstu su prikazani su u mikrosekundama i milisekundama, a u programskom ostvarenju rezultati se prikazuju kao omjer otkucaja signala takta procesora za vrijeme izvođenja i ukupnog broja otkucaja u jednoj sekundi. Programsko ostvarenje algoritama napravljeno je u alatu *Visual studio 2005 Proffesional Edition* i u razvojnoj okolini *Microsoft .NET Framework 2.0*.

5. Zaključak

Na temelju analize ostvarenih sustava za upravljanje spremničkim prostorom, zaključuje se kako je izvedba pomoću binarnih susjednih blokova i razdvojenih lista puno primjerenija korištenju u ugradbenim računalnim sustavima od izvedbe dvostruko povezanom listom. Unatoč brojnim nedostacima, ovako ostvaren sustav pokazuje se pouzdanim i učinkovitim u okolini sa ograničenim resursima. Vrijeme izvođenja i složenost algoritama potrebnih za izvedbu pokazuje povoljne karakteristike za korištenje u ugradbenim sustavima. Uz dodatna poboljšanja, može se ostvariti veća ušteda spremničkog prostora, ali uz cijenu povećanja složenosti i pada u ukupnoj performansi.

Dvostruko povezane liste mogu se učinkovito koristiti u sustavima sa većim memorijskim resursima, iako danas postoje puno naprednije izvedbe upravljanja memorijom, koje ovaj vrlo rudimentarni način upravljanja istiskuju iz upotrebe.

Proučavanje načina upravljanja spremničkim prostorom vrlo je važno područje računalne znanosti i teorije računarstva, koje se konstantno razvija, proširuje i poboljšava. Učinkovito razvijen sustav upravljanja memorijom doprinosi ukupnom poboljšanju performanse cijelog računalnog sustava.

Potpis:

_____

Popis korištene literature

[1] Bartlett, J.: *Inside Memory Management: The Choices, Tradeoffs, and Implementations of Dynamic Allocation*, 16.11.2004., preuzeto sa http://www.ibm.com/developerworks/br/library/l-memory/ , 16.05.2010.

[2] Blunden, B. : *Memory Management: Algorithms and Implementation in C/C++*, Plano,Texas, SAD: Wordware Publising ,Inc, 2003.

[3] Crespo,A., Ripoll,I., Masmano, M. : *Dynamic Memory Management for Embedded Real-Time Systems*, preuzeto sa stranice: http://www.gii.upv.es, Grupo de Informática Industrial – Sistemas de Tiempo Real, Universidad Politécnica de Valencia, 2007.

[4] Donahue,S.M., Hampton, M.P., Deters,M., Nye, J.M., Cytron,R.K., Kavi,K.M. : 'Storage Allocation for Real-Time, Embedded Systems', rad objavljen u *Embedded Software (EMSOFT)*, Tahoe City, California, SAD, Springer-Verlag, 2001.

[5] Jain, P., Chiou,D., Devadas,S., Rudolph,L. : *Application-Specific Memory Management for Embedded Systems,* Computation Structures Group Memo 427, Massachusetts Institute of Technology (MIT), Laboratory for Computer Science, 1999.

[6] Li, Q., Yao, C.: *Real-Time Concepts for Embedded Systems*, San Francisco, SAD, CMP Books, 2003.

[7] Masmano,M., Ripoll,I., Balbastre,P., Crespo,A.: *A Constant-Time Dynamic Storage Allocator for Real-Time Systems,*, Nizozemska, Kluwer Academic Publishers, 2008.

[8] Masmano, M., Ripoll,I., Real,J., Crespo,A., Wellings,A.J.: *Implementation of a Constant-Time Dynamic Storage Allocator*, Software: Practice and Experience, online izdanje, 2007.

[9] Marinissen, E.J., Prince, B., Keitel-Schulz, D., Zorian, Y. : *Challenges in Embedded Memory Design and Test*, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), 2005.

[10] Michael, M.M.: *Experiences Developing Concurrent Algorithms*, IBM T. J. Watson Research Center, IBM Corporation, online izdanje, 2008.

[11] Serewa,S. : *The Improvement of the Buddy System*, Theoretical and Applied Informatics, ISSN 1896-5334, vol. 18, no.2., pp. 133 – 140, studeni 2006.

Popis slika

Popis tablica

Sažetak

Naslov: Algoritmi upravljanja spremničkim prostorom za ugradbena računala

Sažetak: Ugradbena računala raspolažu vrlo malim količinama memorije. Kod izvedbe upravljanja spremničkim prostorom za ograničene količine memorije potrebno je uzeti u obzir unutarnju i vanjsku fragmentaciju, ukupnu iskoristivost preostalog prostora, vrijeme potrebno za zauzimanje i oslobađanje blokova memorije, kao i ukupnu performansu ostvarenog sustava. Složenost algoritama korištenih u izvedbi vrlo je važan faktor pri ostvarenju takvog sustava. Pri mjerenju performanse i procjeni složenosti ostvarenih algoritama potrebno je promotriti ponašanje pri najgorem mogućem slučaju izvođenja. Postojeće algoritme moguće je modificirati kako bi im se smanjila složenost i količina fragmentacije, te povećala ukupna brzina izvođenja.

Ključne riječi: ugradbena računala, algoritmi za upravljanje memorijom, procjena složenosti algoritma, mjerenje performanse

Summary

Title: Memory management algorithms for embedded computers

Summary: Embedded computers have very small amounts of memory at disposal. In realization of memory management for limited amounts of memory it is necessary to take into consideration internal and external fragmentation, overall usability of remaining memory space, the amount of time needed for allocation and freeing blocks of memory, as well as overall performance of realized system. The complexity of used algorithms is very important characteristic in development of such system. While testing performance and evaluating complexity of managed algorithms, it is necessary to observe the behavior in worst-case execution scenario. The existing algorithms can be modified to decrease complexity and the amount of fragmentation, and also to improve the overall execution time.

Keywords: embedded computers, memory management algorithms, algorithms complexity evaluation, performance measurement

Privitak

Dodatak 1: Test performanse

Programsko ostvarenje testa performanse radi se tako, da se koriste dodatne strukture i funkcije za generiranje zahtjeva prema raspodjeli opisanoj u radu. Na početku programa dodaje se slijedeći programski kod:

```
double p[8] = {.15, .20, .35, .20, .02, .04, .02, .02};
unsigned long x[8] = {16,32,64,128,256,512,1024,2048};
unsigned long s[10000];
```

Prva struktura sadrži vjerojatnosti pojave zahtjeva određene veličine. Iduća struktura sadrži veličine zahtjeva. posljednju strukturu zapisuju se generirane veličine zahtjeva koji će se izvršiti za potrebe testiranja.

Također, koristi se dodatna funkcija `zahtjev()` pomoću koje se generiraju zahtjevi ovisno o zadanoj raspodjeli i zapisuju u strukturu `s[]`.

```
void zahtjev(int i, double p[]){
	int n=0;
	double r;
	for(n=0;n<i;n++){
		r = (((double)rand())/((double)RAND_MAX));
		if(r < p[0]) s[n]=(x[0]);
		else if(r <(p[0]+p[1]))s[n]=(x[1]);
		else if(r <(p[0]+p[1]+p[2]))s[n]=(x[2]);
		else if(r <(p[0]+p[1]+p[2]+p[3]))s[n]=(x[3]);
		else if(r <(p[0]+p[1]+p[2]+p[3]+p[4]))s[n]=(x[4]);
		else if(r <(p[0]+p[1]+p[2]+p[3]+p[4]+p[5]))s[n]=(x[5]);
		else if(r <(p[0]+p[1]+p[2]+p[3]+p[4]+p[5]+p[6]))s[n]=(x[6]);
		else s[n]=(x[7]);
	}
}
```

Funkcija kao argumente prima ukupni broj zahtjeva koje mora generirati, te pokazivač na strukturu koja sadrži vjerojatnosti pojave zahtjeva. Za svaki zahtjev pomoću funkcije `rand()` generira se pseudoslučajna vrijednost između 0 i 1. Ovisno o generiranoj vrijednosti, odabire se veličina zahtjeva i zapisuje se u strukturu s[] na mjesto čiji je indeks jednak indeksu zahtjeva.

Nakon zauzimanja prostora za spremnik nad kojim se izvodi test (za potrebe testa, veličina je unaprijed postavljena na 32 kB), od korisnika se traži da unese broj zahtjeva koje je potrebno generirati. Nakon toga, počinje se izvoditi deset uzastopnih mjerenja vremena potrebih za alokaciju i oslobađanje blokova generiranih veličina. Mjerenje se izvodi pomoću ugrađene funkcije `QueryPerformanceCounter()`, kako bi se postigla veća preciznost rezultata. Navedena funkcija vraća vrijednost brojača otkucaja

procesorskog takta u trenutku poziva. Ova funkcija poziva se prije prve alokacije i nakon posljednjeg oslobađanja zauzetog bloka. Razlika dvije dobivene vrijednosti jednaka je ukupnom proteklom vremenu. Vrijednosti rezultata izražene su kao omjer izračunate vrijednosti broja otkucaja i ukupne frekvencije procesora, koja se dobiva pozivom ugrađene funkcije `QueryPerformanceFrequency()`. Ovaj postupak ponavlja se deset puta, i na kraju se ukupni rezultat izražava kao srednja vrijednost svih izvršenih mjerenja.

Ovisno od izvedbe mehanizma upravljanja memorijom, potrebno je u programski kod testa performanse dodati sve potrebne funkcije koje ostvaruju upravljanje memorijom za taj mehanizam. Između dohvaćanja dviju vrijednosti brojača, pozivaju se funkcije `alociraj()` i `oslobodi()` ostvarenog sustava. Općeniti oblik programskog koda korištenog za testiranje performanse slijedi u nastavku teksta. Također, potrebno je obratiti pozornost na dodatni zahtjev koji se postavlja kod testiranja *lazy-buddy* mehanizma: nakon oslobađanja posljednjeg alociranog bloka, i prije pokretanja novog testa, potrebno je prvo spojiti sve oslobođene blokove kako bi se memorija vratila u početno stanje. Ovo je potrebno napraviti pozivom pripadne funkcije `join()` nakon završetka mjerenja.

Dodatak 2: Programski kod testa performanse (općeniti oblik)

```
#define _CRT_SECURE_NO_DEPRECATE
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <windows.h>

//...
//prostor za definiranje potrebnih struktura za ostvarenje sustava
//...

double p[8] = {.15, .20, .35, .20, .02, .04, .02, .02};
unsigned long x[8] = {16,32,64,128,256,512,1024,2048};
unsigned long s[10000];

//-------------------------------------------------------------------------------/
/FUNKCIJA ZA GENERIRANJE ZAHTJEVA ZA MEMORIJOM
//-------------------------------------------------------------------------------
void zahtjev(int i, double p[]){
	int n=0;
	double r;
	for(n=0;n<i;n++){
		r = (((double)rand())/((double)RAND_MAX));
		if(r < p[0]) s[n]=(x[0]);
		else if(r <(p[0]+p[1]))s[n]=(x[1]);
		else if(r <(p[0]+p[1]+p[2]))s[n]=(x[2]);
		else if(r <(p[0]+p[1]+p[2]+p[3]))s[n]=(x[3]);
		else if(r <(p[0]+p[1]+p[2]+p[3]+p[4]))s[n]=(x[4]);
		else if(r <(p[0]+p[1]+p[2]+p[3]+p[4]+p[5]))s[n]=(x[5]);
		else if(r <(p[0]+p[1]+p[2]+p[3]+p[4]+p[5]+p[6]))s[n]=(x[6]);
		else s[n]=(x[7]);
	}
}

//...
//prostor za definiranje potrebnih funkcija za ostvarenje sustava
//to po potrebi ukljucuje i funkciju za ispis stanja memorije
//...

//-------------------------------------------------------------------------------
--//MAIN
//-------------------------------------------------------------------------------
--int main(){
	int i=0,j=0,k=0,br_zahtjeva=0,n=0;
	char *memblok;//pokazivac na pocetak memorije
	char *mp;//pomocni pokazivac
	size_t size;//velicina memorije u B
	int mem_size = 32768;// velicina memorije postavljena na 32K
	char *adresa[10000];
	LARGE_INTEGER t1,t2,tm,tm1,tm2,tps,t_ukupno;
	t_ukupno.LowPart=0;
	t_ukupno.HighPart=0;

	printf("Velicina memorijskog spremnika: %d\n", mem_size);
//-------------------------------------------------------------------------------
--//Pretvorba kB-->B i alociranje prostora koji ce predstavljati memoriju
//-------------------------------------------------------------------------------
--	if( (memblok = (char *)malloc( (int)(mem_size) * sizeof( char ) )) == NULL
){
		printf("Memoriju nije moguce alocirati!\nZavrsetak programa...\n");
		exit(1);
	}
	addr = memblok;
```

```
      size = _msize( memblok );
//...
//po potrebi se ovdje koriste dodatni mehanizmi za postavljanje početnog stanja
//korištenog sustava
//...

//--------------------------------------------------------------------------
--
//Test performanse: mjerenje vremena potrebnog za alociranje i oslobadjanje nekog
broja zahtjeva
//--------------------------------------------------------------------------
--     printf("Broj zahtjeva za memorijom : ");
    scanf("%d",&br_zahtjeva);
      QueryPerformanceFrequency(&tps);
      for(n=1;n<=10;n++){
            srand( (unsigned)time( NULL ) );
            zahtjev(br_zahtjeva,p);
            printf("\nTest %d: POCETAK\n\n",n);
            printf("-----------------------------------------\n");

            QueryPerformanceCounter(&t1);
            for(i=0;i<br_zahtjeva;i++){
                  //poziv odgovarajuce funkcije za alociranje
                  adresa[i] = alociraj(...);
            }
            for(i=0;i<br_zahtjeva;i++){
                  //poziv odgovarajuce funkcije za oslobađanje
                  if(adresa[i]!=NULL) oslobodi(...);
            }
            QueryPerformanceCounter(&t2);

            tm1.QuadPart=t1.QuadPart % tps.QuadPart;
            tm2.QuadPart=t2.QuadPart % tps.QuadPart;
            tm.QuadPart=tm2.QuadPart-tm1.QuadPart;
            t_ukupno.QuadPart+=tm.QuadPart;

            printf("\nTest %d:KRAJ\n",n);
            printf("-----------------------------------------\n");
            printf("Vrijeme potrebno za alokaciju i oslobadjanje %d blokova
                  memorije: %lu/%lu s\n",br_zahtjeva,tm.LowPart,tps.LowPart);
      }
//--------------------------------------------------------------------------
--
//Kraj programa
//--------------------------------------------------------------------------
--
      printf("\nSvi testovi su zavrsili s radom\n");
      printf("-----------------------------------------\n");
      printf("Prosjecno vrijeme potrebno za alokaciju i oslobadjanje %d blokova
            memorije:\n\n%lu/%lu s\n",br_zahtjeva,(t_ukupno.LowPart/10),
            tps.LowPart);
      free( memblok );
      return 0;
}
```

Dodatak 3: Primjer ispisa testa performanse – buddy block algoritam

```
Velicina memorijskog spremnika: 32768
Stanje memorije na pocetku rada:

|| BLOCK_SIZE || FREE || ADRESS (addr_begin-addr_end)
---------------------------------
32768 || 1 || (0 - 32767) |
16384 || 0 ||
8192 || 0 ||
4096 || 0 ||
2048 || 0 ||
1024 || 0 ||
512 || 0 ||
256 || 0 ||
128 || 0 ||
64 || 0 ||
32 || 0 ||
16 || 0 ||

Broj zahtjeva za memorijom : 100

Test 1: POCETAK
-----------------------------------------

Test 1:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
24011/3579545 s

Test 2: POCETAK
-----------------------------------------

Test 2:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
24144/3579545 s

Test 3: POCETAK
-----------------------------------------

Test 3:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
24362/3579545 s

Test 4: POCETAK
-----------------------------------------

Test 4:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
24715/3579545 s

Test 5: POCETAK
-----------------------------------------

Test 5:KRAJ
-----------------------------------------
```

```
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
24486/3579545 s

Test 6: POCETAK
-----------------------------------------

Test 6:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
24285/3579545 s

Test 7: POCETAK
-----------------------------------------

Test 7:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
24056/3579545 s

Test 8: POCETAK
-----------------------------------------

Test 8:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
18355/3579545 s

Test 9: POCETAK
-----------------------------------------

Test 9:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
18356/3579545 s

Test 10: POCETAK
-----------------------------------------

Test 10:KRAJ
-----------------------------------------
Vrijeme potrebno za alokaciju i oslobadjanje 100 blokova memorije:
18765/3579545 s

Svi testovi su zavrsili s radom
-----------------------------------------
Prosjecno vrijeme potrebno za alokaciju i oslobadjanje 100 blokova
memorije:

22553/3579545 s
```