SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br.1412

PODSUSTAV ZA UPRAVLJANJE SPREMNIKOM UGRADBENOG RAČUNALA

Kornelija Vodanović

Zagreb, lipanj 2010.

SADRŽAJ

1. Uvod

U današnje vrijeme svakodnevnim povećavanjem modernizacije računala se primjenjuju u svim aspektima života. Osim u industriji, primjenjuju se u svakodnevnom životu, obrazovanju, medicini, komunikaciji, igri…

Upravo zbog te raširenosti, računala se moraju prilagoditi pojedinim zahtjevima korisnika, pa tako kod komunikacije korisnici teže malim, prijenosnim računalima, dok kod većih industrijskih postrojenja računala ne moraju nužno biti malena, nego samo procesorski snažna.

U ovom radu prikazati ćemo upravljanje memorijom ugradbenih računala.

Ugradbena računala su računala male veličine, ali usprkos tome zahtijevaju dobre performanse. Kod njih je vrlo bitna veličina, brzina rada i obrade zahtjeva ali i predvidljivost ponašanja u raznim situacijama. Upravo zbog te, zahtijevane, brzine rada, memorija mora brzo obrađivati zahtjeve korisnika ili samog sustava i dodjeljivati mu potrebnu memoriju. Zbog toga u ovom radu obrađujemo nekoliko algoritama za upravljanje memorijom, nudeći programska rješenja za neke od problema koji se pojavljuju u ugradbenim računalnim sustavima.

2. Opis razmatrane okoline

U okviru ovog rada promatrati ćemo ugradbene računalne sustave i upravljanje memorijom u njima. Ugradbena računala su samostalni računalni sustavi posebne namjene. Danas se koriste svugdje, od običnih digitalnih satova i mikrovalnih pećnica, pa sve do vrlo kompleksnih računala koja se koriste u medicini za mjerenja, u automobilima, avionima i nuklearnim elektranama .

Ovi sustavi imaju posebne zahtjeve nad memorijskim sustavom, a to su brzo i dinamičko upravljanje memorijom u vremenski ograničenom periodu.

2.1. Ugradbena računala[1]

Neka ugradbena računala su vrlo jednostavna, a druga pak vrlo složena. Ovi drugi obavljaju, obično, samo mali skup zadataka koji zahtijevaju vrlo snažne procesore i rad u stvarnom vremenu. Uz već navedena ograničenja tu je i problem veličine i težine računala. Naime, kako su to ugradbena računala, to znači da će oni biti postavljeni u neki dio aparata ili stroja, zbog čega oni moraju biti male površine, ali uz stabilne performanse.

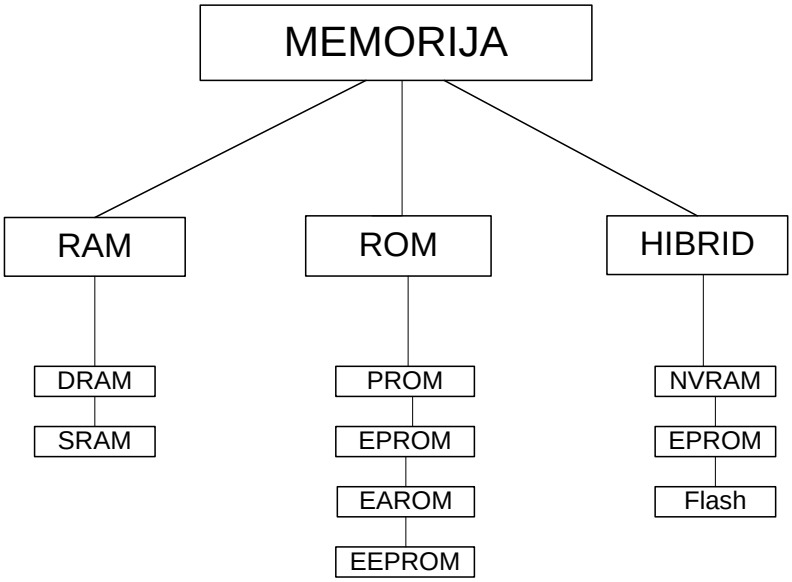
Kod ugradbenih računala svi standardni dijelovi računala su dosta reducirani, tako oni često nemaju ekran ili tipkovnicu, dok je memorija izuzetno mala, a program koji pokreće računala se pohranjuje ili na posebnoj memoriji (eng. *read-only*) ili na *flash* memorijskom čipu.

Ovakva računala zahtijevaju i poseban operacijski sustav koji mora biti izuzetno kompaktan i efikasan i mora izvoditi sve operacije gotovo u stvarnom vremenu što znači da program mora minimizirati kod, i kritične dijelove koda, a komunikacija između procesora i memorije mora biti što brža i pouzdanija.

Zbog sporosti komunikacije između procesora i memorije vrlo je bitan dobar odabir memorije za ugradbena računala.

2.2. Upravljanje memorijom u ugradbenim računalnim sustavima

Kada govorimo o memoriji razlikujemo više vrsti, ovisno o namjeni i načinu pisanja / brisanja podataka. Tako memoriju možemo podijeliti na RAM, ROM i hibridnu vrstu.



Slika 1. Vrsta memorije

RAM (od eng. *Random-access memory*) dalje možemo raščlaniti na dvije podvrste ovisno o tome koliko će se podaci čuvati u memoriji. Kod SRAM (eng. *Static RAM*) memorije podaci bivaju sačuvani sve dok je memorija priključena na struju, dok DRAM (eng. *Dynamic RAM*) čuva podatke tek četvrtinu sekunde, ali uz pomoć posebnog DRAM kontrolera, koji osvježava podatke, možemo produljiti trajanje podataka.

Kod odabira RAM-a vrlo bitno je obratiti pažnju na brzinu komunikacije između memorije i procesora, ali i samu cijenu memorije. Naime SRAM može biti i do četiri puta brži od DRAM-a, ali upravo zbog toga mu je i cijena znatno viša, što kod izrade ugradbenih računala može biti vrlo nepovoljno. Zato se SRAM koristi jedino kada su izuzetno brza komunikacija i odziv memorije bitni.

Često se u ugradbenim računalima mogu pronaći oba tipa, i to tako da postoji jedan manji dio memorije koji je tipa SRAM a koristi se za kritične podatke i komunikaciju, i jednog velikog dijela tipa DRAM koji služi za obavljanje svih drugih poslova.

Na RAM se može pisati i brisati, uvjetno rečeno, neograničen broj puta.

ROM (eng. *Read-only memory*) je memorija iz koje se podaci mogu samo čitati, a razlikujemo ih prema načinima zapisa podataka, ali i prema broju upisivanja podataka na njih. Tako razlikujemo PROM (eng. *Programmable ROM*), EPROM (eng. *Erasable Programmable ROM*), EAROM (engl. *Electrically Alterable Read-Only Memory*), EEPROM (engl. *Electrically Erasable Read-Only Memory*) i CD-ROM.

PROM je memorija koja je kod kupnje u potpunosti prazna (sve lokacije su postavljene na 1). U nju se samo jednom mogu upisivati podaci i to pomoću posebnog uređaja PROM programera, koji upisuje riječ po riječ i time kida unutrašnje veze zbog čega je upisivanje uspješno samo jednom. Ako želimo izmijeniti podatke koji su zapisani na PROM-u moramo zamijeniti cijeli memoriju.

EPROM je vrlo sličan PROM-u, jer se podaci na njega upisuju isto kao i na PROM, ali ovakva memorija ima mogućnost brisanja podataka i ponovnog zapisivanja. Ako jednom zapisane podatke želimo obrisati, cijelu memoriju moramo izložiti jakom ultraljubičastom zračenju, koja resetira sva polja, nakon čega se opet pomoću PROM programera mogu upisati novi podaci. Ovakva brisanja oštećuju memoriju, stoga nije moguće neograničeno puta brisati i upisivati nove podatke.

EAROM je memorija kojoj se može mijenjati sadržaj, ali nije dizajniran za česte izmjene pa uglavnom se koristiti kao ROM.

EEPROM električno izbrisiva programibilna ispisna memorija koja služi kao *flash* memorija. Može se obrisati cijela, ili samo pojedini dijelovi, s tim da se ne mora vaditi van iz računala kako bi se obrisala. Ova memorija ima vrlo dugo vrijeme upisivanja i dohvaćanja podataka stoga nije poželjna u ugradbenim računalima.

Još je dobro spomenuti i starije verzije memorije kao što su diodni matični ROM u koji su se upisivali podaci tako da se na poseban način slažu poluvodičke diode na tiskanoj pločici. Takva memorija je izuzetno jeftina pogotovo kada ju je potrebno ugraditi u nekoliko stotina ugradbenih računala.

2.3. Memorijski zahtjevi kod ugradbenih računalnih sustava

Kao što smo već napomenuli, ugradbeni računalni sustavi su u pravilu malih dimenzija, zbog čega sve komponente moraju biti male, pa tako i memorija. Memorija ne smije biti velika i obično je ugrađena direktno u sam čip zbog čega nije proširiva. Upravo zbog toga memorija se dijeli na manje dijelove i ugrađuje u dijelove sustava gdje se onda koristi samo za pojedinačnu namjenu. Tako možemo razlikovati memoriju namijenjenu samo za instrukcije, memoriju za podatke i memoriju za ulazno izlazne jedinice.

Kako bi se, što efikasnije, iskoristila u startu mala memorija u ugradbenim računalnim sustavima koristi se:

- statički pre-alocirana memorija
- dinamičko upravljanje memorijom

Vrlo često se straničenje i ne koristi zbog zahtjeva za posebnim sklopovljem i dodatnom strukturom podataka (tablice prevođenja).

3. Algoritmi upravljanja memorijom

Upravljanje memorijom je postupak dodjeljivanja dijelova memorije procesima, na njihov zahtjev, te oslobađanja istih, po završetku korištenja, kako bi se mogli ponovo koristiti.

Kada se govori o upravljanju memorijom potrebno je posebnu pažnju dati:

- *alociranju memorije* – dodjeljivanje dijela memorije točno određene veličine procesu,
- *realociranju memorije* – ponovnom dodjeljivanju dijela memorije nakon što se neki proces ponovo aktivirao,
- *oslobađanju memorije* – sva zauzeta memorija se mora moći i osloboditi kako bi se novi nadolazeći procesi mogli uopće izvršavati,
- *zaštiti dodijeljene memorije* – bitno je da drugi procesi ne mogu koristiti taj dio memorije i mijenjati zapisane podatke,
- *dijeljenju memorije* – iako je, prije, navedeno da se memorija štiti od drugih procesa, ponekad će biti potrebno da više procesa pristupa istom dijelu memorije,
- fizičkoj i logičkoj organizaciji memorije.

Isto tako potrebno je napomenuti da postoji više slojeva upravljanja memorijom.

Najniža razina upravljanja memorijom je sklopovska razina, a obuhvaća samu memoriju (RAM, diskove, registre procesora, *cache* memoriju, i MMU), ali i potrebno sklopovlje koje će fizički zapisivati u memoriju, na točno određene lokacije.

Te lokacije određuje druga razina upravljanja, a to je razina operacijskog sustava.

Ova razina se bavi upravljanjem postojeće memorije, stvaranjem virtualne memorije za procese, ali i zaštitom i sigurnošću memorije i njenih dodijeljenih fragmenata.

Najviša razina upravljanja jest razina aplikacijskog upravljanja memorijom. Aplikacija po potrebi traži od operacijskog sustava potrebnu memoriju, te kada ju i dobije, ona se brine kako će taj prostor iskoristiti njeni procesi.

U ovom projektu ćemo se baviti samo razinom operacijskog sustava, te postojećim algoritmima za dodjeljivanje memorije, a to su:

- *Buddy* alociranje memorije
- *Slab* alociranje memorije
- *Memory pool*

3.1. Buddy upravljanje memorijom

Buddy algoritam za alociranje memorije je osmislio Harry Markowitz 1963, a prvi ga je opisao Kenneth C.Knowlton 1965. godine.

Spada u dinamičko upravljanje memorijom, jer se mogu rezervirati blokovi raznih veličina, ovisno o potrebama procesa.

Ovaj algoritam je jedan od jednostavnijih i ne zahtijeva posebno sklopovlje kako bi se realizirao, a može se primjenjivati kod vrlo jednostavnih procesora, što uvelike pomaže kod izgradnje ugradbenih računala.

Temelji se na principu dijeljenja memorije na dijelove koji su veličine potencija broja 2, ili na dijelove koji odgovaraju brojevima Fibonaccijevog niza, gdje je svaki sljedeći broj zbroj prethodna dva broja.

Ovdje ćemo proučavati podjelu na potenciju broja dva, s tim da je maksimalna veličina memorije određena od strane programera (maksimalna veličina memorije je 2^x, gdje x, gornju granicu, određuje programer sustava).

ALGORITAM ALOCIRANJA MEMORIJE

Prvi korak u ostvarivanju algoritma jest definiranje veličine memorije (2^x) tj. njene gornje granice (broja x koji ujedno i definira veličinu najvećeg mogućeg bloka memorije), i najmanje veličine bloka koji je isto potencija broja dva.

Nakon definiranja blokova krećemo sa algoritmom.

Kako bi ostvarili ovaj algoritam potrebno je ostvariti niz listi za svaku dopuštenu veličinu bloka.

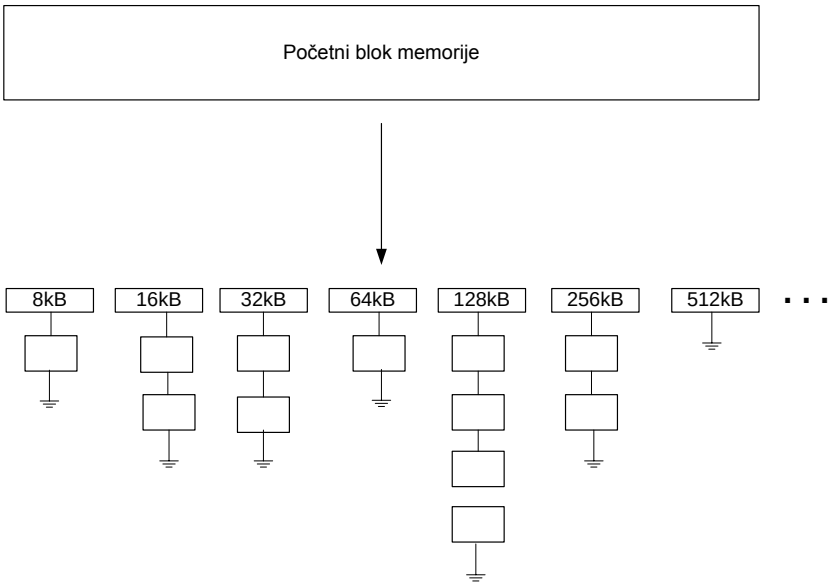
Dopuštene veličine bloka mogu biti bilo koje veličine. Ako hoćemo poboljšati performanse sustava, veličine blokova bi trebale biti što bliže jedna drugoj kako bi imali što manju fragmentaciju. Npr. ako imamo blokove veličine 8B i 16B, tada ako sustav zatraži blok veličine 7B imamo unutarnju fragmentaciju od 1B što i ne predstavlja toliki problem. Ako, pak, sustav zatraži blok veličine 9B, tada ćemo morati zauzeti blok od 16B čime dobivamo fragmentaciju od 7B, što uvelike povećava fragmentaciju.

Jedan način dobivanja veličina blokova je tako da se uzme postojeća prazna memorija i podjeli na pola, a zatim jednu polovicu dalje na pola, sve dok se ne dođe do željene veličine, ili do definiranog najmanjeg segmenta memorije. Tako će se niz može sastojati od listi za veličine 8kB, 16kB, 32kB,…, ovisno o tome kolika je definirana veličina najmanjeg bloka.

U listama blokova će se bilježiti koji blokovi memorije su prazni, a liste mogu biti organizirane proizvoljno (LIFO, FIFO, prema početnim i završim adresama blokova koje predstavljaju, ...) s tim da je najbrže da se uvijek uzima prvi element liste, a da se na kraj liste dodaju oslobođeni blokovi.

Kada imamo definirane i popunjene liste kao što je prikazano na slici 2 možemo krenuti s alociranjem memorije.

Kada neki program zatraži od operacijskog sustava blok memorije veličine n operacijski sustav prvo određuje koju veličinu bloka mora pretraživati. Veličina bloka 2^y se računa tako da ona bude veća ili jednaka željenoj veličini n.



Slika 2. Podjela memorije u *buddy* algoritmu

Nakon toga počinje pretraživanje niza listi. Ako u nizu postoji lista za 2^y veličinu bloka i lista sadrži u sebi zapise o praznim blokovima, programu se dodjeljuje prvi blok zapisan u listi.

Ako, pak, ne postoji u nizu zapis za tu veličinu bloka onda algoritam mora od slobodne memorije napraviti odgovarajući blok i to tako da ponavlja sljedeće korake dok ne ostvari željenu veličinu:

1. *Algoritam uzima prvi blok memorije koji je veći od željenog i slobodan, te ga dijeli na pola.*
2. *Ako je dobivena polovica veličine 2^y onda se algoritam prekida, i program dobiva taj dio memorije. Druga polovica se uvrštava u odgovarajuću listu u nizu.*
3. *Ako nije, a dobivene polovice su još uvijek veće od željene, algoritam ponovo dijeli jednu polovicu na pola i skače na točku 2, u suprotnom algoritam skače na točku 1.*

Blokovi koji se dobivaju dijeljenjem blokova na pola, a koji se ne dodijele programu se uvrštavaju u pripadajuće liste.

OSLOBAĐANJE MEMORIJE

Ako je neki proces završio sa radom, ili je prekinut, onda se njemu dodijeljena memorija mora osloboditi. To se radi tako da se dodijeljeni blok vraća natrag u niz listi. Prije nego se uvrsti u odgovarajuću listu, algoritam gleda da li su susjedni blokovi tog bloka slobodni, ako jesu, onda spaja te blokove, u novi, veći. To ponavlja sve dok postoje susjedni blokovi koji su u listama nealociranih blokova, tj. koji su prazni.

Jednom kada se spoje svi susjedni prazni blokovi, ili kada se dođe do maksimalne veličine bloka (tj. kada sva memorija bude prazna) algoritam se prekida, te se dobiveni novi blok uvrštava u odgovarajuću listu u nizu.

Spomenuta fragmentacija se uvelike može smanjiti, tako da se prilikom oslobađanja memorije uvijek spajaju susjedne rupe u memoriji, u veće dijelove.

Ovaj algoritam može biti izuzetno dobar i brz, ako svi procesi imaju standardizirane potrebe za memorijom, te ako su uvijek blokovi koje procesi traže unutar početnih definiranih blokova. U suprotnom, ako svaki proces ima svoju posebnu veličinu potrebne memorije, ovaj algoritam nije optimalan jer za svaki novi proces mora računati novu veličinu bloka.

3.2. Slab upravljanje memorijom

Za vrijeme rada računala, procesi ili operacijski sustav neke tipove podataka često koriste, često stvaraju i uništavaju nakon uporabe. Za svako zauzimanje memorije za te podatke, operacijski sustav mora pregledati memoriju i pronaći blok koji bi najbolje odgovarao željenim podacima, što oduzima puno vremena. Često uništavanje tih objekata uzrokuje unutarnju fragmentaciju, što nikako nije dobro za upravljanje memorijom.

Upravo zbog toga bi bilo dobro da se za te jezgrene objekte, koji se često koriste, unaprijed alociraju posebni blokovi, koji će se uvijek koristiti samo za tu

vrstu podataka. Podaci o tim unaprijed alociranim blokovima se čuvaju u posebnoj memoriji (eng. *cache memory*) kako bi se zahtjevi mogli brže (trenutno) obrađivati.

Kada neki proces zatraži blok memorije, operacijski sustav pogleda kojeg su tipa podaci koji će se pohraniti unutra, i ako su podaci, podaci posebne vrste, operacijski sustav iz *cache* memorije vraća procesu alocirani blok memorije.

Kada proces završi sa radom, blok se ne briše, nego se samo njegove adrese vraćaju u *cache* memoriju (oslobađaju), nakon čega neki drugi proces može opet koristiti taj dio memorije.

Ime algoritma potječe od engleske riječi *slab* koja ovdje predstavlja blok memorije koji se može sastojati od jedne ili više povezanih stranica memorije. Oni se svrstavaju u posebnu *cache* memoriju.

Slab se može nalaziti u tri stanja – prazan, polupun i pun. Ako je u stanju prazan onda znači da su svi objekti tog slaba označeni kao slobodni, te se mogu dalje koristit. Ako je u stanju polupun onda se unutar slab-a nalaze i zauzeti i oslobođeni objekti. Ako je pak u stanju pun onda su svi njegovi objekti zauzeti.

Na početku su svi slab-ovi označeni kao prazni, i svi su spremni na korištenje.

Cache memorije je mala, ali vrlo brza memorija. Podijeljena je na dijelove od kojih svaki dio predstavlja memoriju za jedan tip objekta (jedan tip slab-a).

Kada neki program krene sa izvođenjem on stvori *cache* memoriju, i podijeli ju na određen broj dijelova i alocira potreban broj slab-ova.

Algoritam radi tako da kada neki proces zatraži poseban jezgreni objekt, operacijski sustav pokušava naći slobodan slab tog tipa u svojoj *cache* memoriji. Ako postoji, dodjeljuje ga procesu, ako ne postoji, onda sustav pokušava alocirati novi slab iz niza stranica memorije i dodjeljuje ga *cache* memoriji.

3.3. Memory pool

Ovaj algoritam je vrlo dobar kod sustava za rad u realnom vremenu zato jer koristi fiksne veličine blokova, koji su, obično, svi iste veličine. Naime, takvom podjelom memorije smanjuje se vrijeme računanja veličine odgovarajućeg bloka te pretraživanja slobodnih blokova, jer se sve svodi samo na pronalaženje prvog praznog bloka. A takvo ubrzanje je potrebno kod sustava za rad u realnom vremenu.

Ideja ovog algoritma je da se memorija podijeli na jednake dijelove, ili na nekoliko različitih veličina, te se oni smještaju u „bazene“ (eng. *pool*). Kada proces zatreba memoriju on zatraži operacijski sustav za memoriju, na što mu operacijski sustav trenutno vraća adrese koje može koristiti.

Prednost ovog algoritma je brzina, jer ako imamo samo jedan bazen, sa samo jednom veličinom blokova memorije, kada proces zatraži memoriju, on može odmah dobiti blok memorije. Isto tako ako mu treba još, on zatraži novi blok i on ga odmah dobiva, bez posebnog računanja veličine potrebnog bloka i bez pretraživanja listi blokova.

Ako postoji više bazena, sa različitim veličinama blokova, onda je algoritam malo složeniji. U tom slučaju, kada procesor zatraži blok memorije, algoritam mora izračunati iz kojeg bazena bi mogao uzeti najprikladniji blok memorije. Kada pronađe veličinu bloka, koja je jednaka ili veća od željenog, pretražuje taj bazen za slobodnom memorijom. Ako u tom bazenu nema slobodnih blokova, algoritam pretražuje sljedeći veći bazen. I tako sve dok ne pronađe blok memorije za proces.

Otpuštanje zauzete memorije je isto tako vrlo jednostavno. Kada proces završi, njegov blok memorije se vraća natrag u bazen kojem pripada i može se koristiti. S tim da prije nego se oslobodi, algoritam provjerava susjedne bazene da li su slobodni, ako jesu spaja ih sa elementom kojeg želimo oslobodi kako bi se stvorio veći blok i riješio problem fragmentacije.

Detaljniji opis, ostvarenje i razrada ovog algoritma slijedi u sljedećem poglavlju.

4. Ostvareni sustav upravljanja memorijom

Za potrebe ovog rada ostvaren je *memory pool* algoritam za upravljanje memorijom u ugradbenim računalnim sustavima. Ovaj algoritam se razlikuje od prijašnje definicije, te je prilagođeniji stvarnom radu sustava.

U početku moramo definirati kolika je memorija koju koristimo i koje je strukture koristimo, što je prikazano na slici 3.

Slika 3. Struktura memorije nakon njezine inicijalizacije

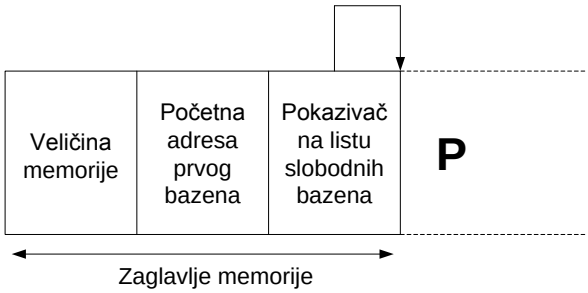
Struktura memorije (M)

Memorija je kao i u osnovnom algoritmu podijeljena na bazene (eng. *pool*), a svaki bazen je podijeljen na više blokova (eng. *chunk*).

U početku, odmah nakon stvaranja memorija će se sastojati od samo jednog bazena, koji je veličine cijele memorije umanjene za zaglavlje memorije. U njemu se nalazi samo jedan blok koji je veličine cijelog bazena umanjenog za zaglavlje (P) i marker bazena (Mp).

Osnovni dijelovi memorije su zaglavlje memorije (na slici prikazano slovom M) u kojem je pohranjena veličina same memorije, adresa početka prvog bazena (P), te pokazivač na listu slobodnih bazena. U listi slobodnih bazena se nalaze svi bazeni koje sustav ne koristi u tom trenutku. Lista je dvostruko povezana, i složena je po adresama bazena kako bi se pretraživanje i umetanje u listu odvijalo sa složenošću $\Theta(1)$. U početku lista praznih bazena pokazuje upravo na ovaj prvi početni bazen. (Slika 4.)

Ograničenje ovog sustava je da se unutar memorije mogu stvarati samo novi bazeni, a samo unutar bazena se mogu stvarati novi blokovi. Novi bazeni se ne mogu stvarati unutar blokova.



Slika 4. Struktura zaglavlja memorije

Sami podaci koji se nalaze unutar bazena, odnosno bloka, se nalaze u dijelu označenom slovom D (eng. *data*). To je ujedno i efektivna veličina samog bloka, jer sustav ne može iskoristiti zaglavlja kako bi u njima pohranio potrebne podatke, zbog kojih stvara blok.

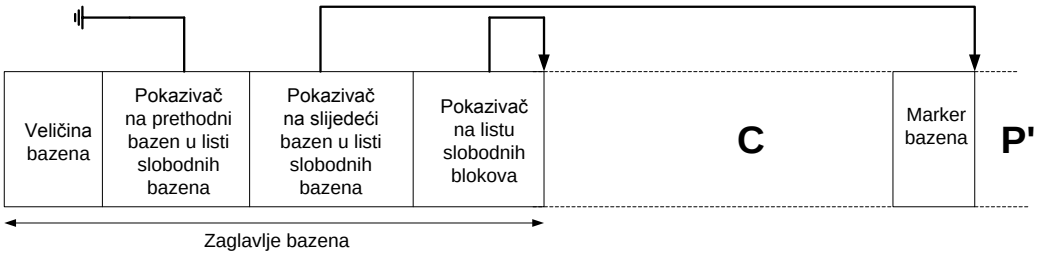
Struktura bazena (P)

Bazen je posebna struktura podataka koja ima svoje zaglavlje, a u sebi korištene i oslobođene blokove.

Zaglavlje bazena je drugačije građe nego zaglavlje same memorije, a prikazano je na slici 5. U zaglavlju se pohranjuje veličina cijelog bazena, a to je veličina koju zahtjeva korisnik i ona uključuje i veličinu zaglavlja i podataka i markera. Uz to nalaze se i dva pokazivača koja se koriste samo kada je bazen slobodan. Tada oni pokazuju na susjedne bazene u listi slobodnih bazena. Pomoću ta dva pokazivača ostvaruje se dvostruko povezana lista, koja je složena po početnim adresama bazena.

Uz ta dva pokazivača postoji i treći koji pokazuje na listu slobodnih blokova, jer svaki bazen se sastoji od jednog ili više blokova. Sustav nakon što je ostvario bazen, od njega traži nove blokove koje onda koristi za svoje potrebe.

Lista praznih blokova je isto organizirana kao i lista slobodnih bazena.



Slika 5. Struktura bazena (u početku stvaranja memorije)

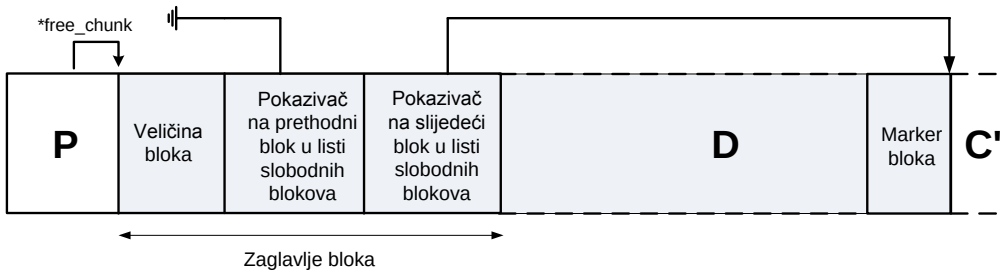
Uz zaglavlje na kraju samog memorijskog područja bazena se nalazi posebna oznaka, marker (Mp) koji označava da li je bazen slobodan ili zauzet. Bazen je zauzet čim sustav zatraži novi bazen. Marker u sebi sadržava veličinu veću od nule, ako je bazen slobodan ili veličinu manju od nule, ako je zauzet.

Vrijednost markera može biti ili -1 za zauzeće ili veličina bazena umanjena za veličinu markera za slobodan bazen. Važnost takvog dodijeljivanja vrijednosti bit će objašnjena kasnije.

Struktura bloka (C)

Blok je, po strukturi, vrlo sličan bazenu. Isto se sastoji od varijable u kojoj je pohranjena veličina bloka, dva pokazivača, na prethodni i sljedeći blok u listi praznih blokova. Kako ne postoji manja jedinica memorije od bloka, onda blok nema pokazivač na listu slobodnih elemenata.

Na kraju bloka je također marker bloka (Mc) koji služi kao oznaka za provjeru da li je blok slobodan ili zauzet. Ako je broj na toj memorijskoj lokaciji veći od nula, tada je blok slobodan, ako je manja od nule tada je blok zauzet.



Slika 6. Struktura bloka

Iako markere uvijek moramo postavljati i kod stvaranja i kod oslobađanja bazena i blokova, markeri će tek doći do izražaja kod oslobađanja elemenata, kada ćemo pomoću markera provjeravati susjedne elemente, da li su slobodni kako bi ih mogli spojiti sa elementom koji želimo osloboditi. Takvim spajanjem sa susjednim slobodnim elementima, povećavamo slobodne bazene, i smanjujemo fragmentaciju memorije.

Vrijednosti markera mogu biti ili -1 za zauzeće ili veličina bloka ako je blok slobodan.

Unutar bloka, u području označenom slovom D, sustav pohranjuje svoje podatke.

Veličina efektivne memorije označena sa D je umanjena za veličinu zaglavlja bloka `sizeof(struct chunk)` i za veličinu markera `sizeof(int)`.

Dodjeljivanje bazena

Kada sustav zatraži bazen on mora pozvati funkciju

```
struct list *create(struct list **list,int size,short type);
```

i poslati joj listu slobodnih bazena, željenu veličinu bazena i tip liste koji šalje. *Type* može biti 0 ili 1, ovisno da li se hoće napraviti novi bazen ili novi blok.

Dodjeljivanje bazena se odvija prema sljedećem algoritmu:

1) *Pretražuj listu praznih bazena sve dok ne naiđeš na kraj liste i dok su elementi manji od zahtijevane veličine*

   2a) ako nisi pronašao niti jedan bazen vrati NULL vrijednost

   2b) čim pronađeš prvi bazen koji je veće ili jednake veličine pogledaj da li je bazen točno željene veličine

      3a) ako je, onda stvori novi bazen, željene veličine, postavi marker na vrijednost -1, stvori mu prvi blok i uvrsti ga u listu slobodnih blokova. Novom bazenu stavi pokazivače na NULL.

4a) Stari bazen izvadi iz liste slobodnih i prespoji susjedne bazene. Ako je taj bazen bio jedini element u listi, pokazivač na listu praznih elemenata postavi na NULL. Vrati novi bazen sustavu.

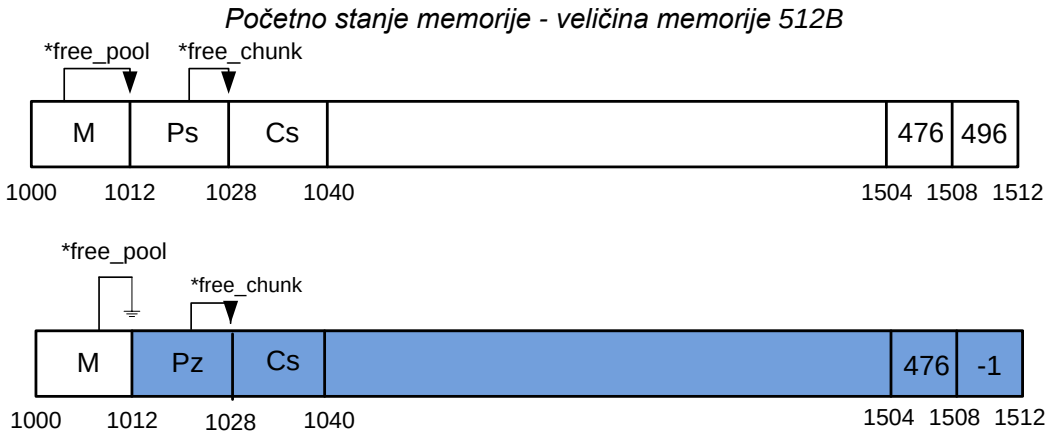
3b) ako je bazen veći od željenog, onda napravi korak 3a s tim da novi bazen dobiva početnu adresu pronađenog bazena, a pronađenom bazenu promijeni početnu adresu i veličinu. Novostvoreni bazen vrati sustavu.

4b) ako je pronađeni bazen bio na početku liste, prespoji početak liste na novu adresu pronađenog bazena

4c) ako nije bio na početku liste onda prethodni bazen spoji na novu adresu pronađenog elementa.

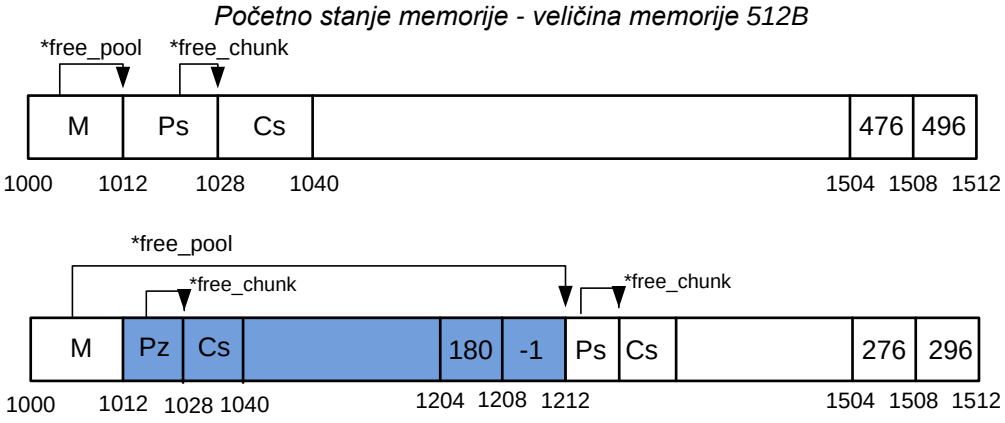
U primjeru na slici 7. vidimo što se događa ako je zahtjev za bazenom, točno veličine cijele memorije tj. točno veličine cijelog bazena. Tada se sustavu vrati novi bazen željene veličine, a lista praznih bazena se uzemljuje (postavlja na NULL).

Plavom bojom je obilježen zauzeti bazen. Ako bi sustav nakon ovog bazena tražio novi, algoritam bi mu vratio NULL vrijednost, jer ne postoji više memorije za stvaranje novog bazena. Na slici vidimo i promjenu pokazivača i markera bazena. Marker prvog bloka, ostaje postavljen na veličinu veću od nule, jer iako je bazen zauzet, on sadržava prvi blok koji je u početku slobodan. Tek zauzimanjem blokova se mijenjaju markeri u blokovima.



Slika 7. Stvaranje bazena veličine memorije

U drugom slučaju opisanom u koracima 3b i 4b algoritma i slikom 8., sustav traži bazen veličine 200B, a u listi slobodnih bazena se nalazi veći bazen. Onda algoritam dijeli postojeći bazen na potrebne veličine, postavlja marker zauzetom bazenu (Pz). U tom bazenu stvara prvi slobodan blok (Cs) i postavlja njegov marker na veličinu koja je smanjena za veličinu zaglavlja bazena i veličinu markera bazena.

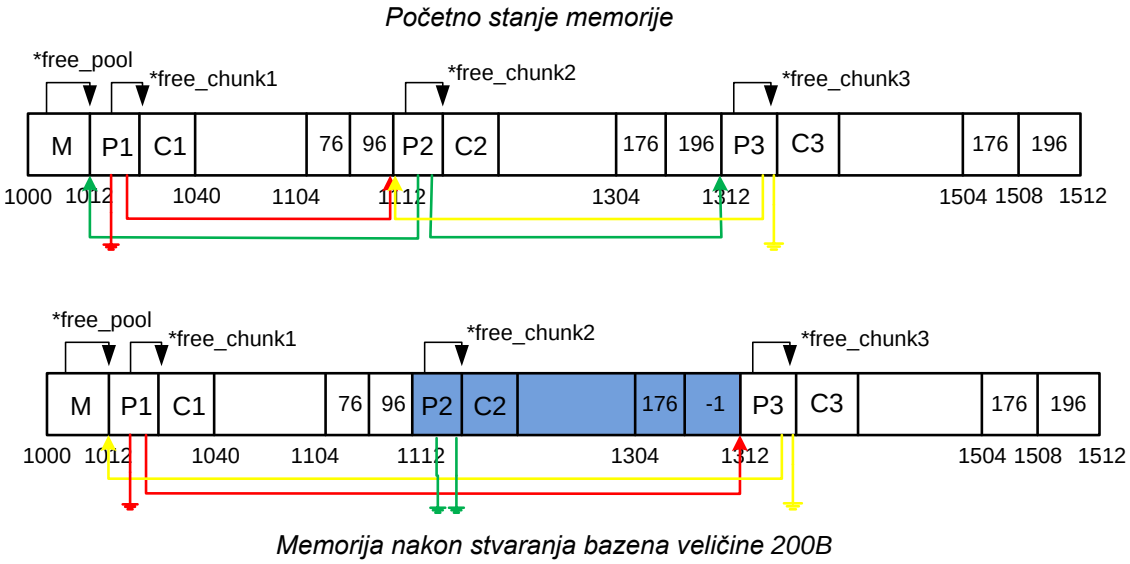


Slika 8. Stvaranje bazena na početku liste

Preostalom dijelu bazena postavlja novo zaglavlje (Ps), mijenja postojeći marker i stvara mu prvi slobodan blok. Ovaj bazen i dalje ostaje u listi slobodnih bazena, samo pošto mu se promijenila početna adresa, što vidimo ispod samog crteža, moramo promijeniti i adresu na koju pokazuje pokazivač slobodnih bazena (**free_pool*).

U ovom slučaju kao ni u prvom nismo promatrali pokazivače bazena na prethodne i sljedeće elemente u listi zato jer su to bili slučajevi kada u listi imamo samo jedan element pa bi stoga oba pokazivača pokazivala na NULL. Zanimljivije ponašanje pokazivača slijedi u trećem primjeru.

Treći slučaj opisan algoritmom javlja se kada smo pronašli odgovarajući bazen, ali se on nalazi u sredini liste slobodnih. To je slučaj opisan koracima 3b i 4c, i slikom 9.



Slika 9. Zauzimanje bazena u sredini liste

Algoritam pretražuje listu do prvog većeg ili veličinom jednakog bazena (eng. *first fit*) kako bi se smanjila složenost pretraživanja liste. Na ovoj slici bitna razlika je u pokazivačima bazena.

Prvi bazen (P1) ima pokazivače crvene boje i na početku pokazivač na prethodnog pokazuje na NULL, jer je on prvi u listi, a pokazivač na sljedećeg pokazuje na drugi bazen. Drugi pak bazen (P2) ima pokazivače zelene boje i oni pokazuju na prvi bazen, za prethodni element liste, i na treći bazen za sljedeći element liste. Treći bazen ima pokazivače žute boje i oni pokazuju na drugi bazen, za prethodni element i na NULL za sljedeći, jer je treći bazen zadnji u listi. Time vidimo dvostruku povezanost liste slobodnih bazena.

Kod ovog slučaja opet se mogu dogoditi dvije situacije, da je pronađeni element točne veličine, ili manji zbog čega se postojeći bazen mora dijeliti na manje dijelove. Ponašanje pokazivača prethodnog i sljedećeg bazena smo prikazali na slici 8 pa ćemo to na ovom primjeru zanemariti.

U ovom slučaju bitno je vidjeti kako se mijenjaju crveni i žuti pokazivači. Vidimo da nakon što izvadimo drugi bazen iz liste, prvi i treći bazen se moraju spojiti, tako da sljedeći element prvog bazena postane treći, a prethodni element bazena tri, postane prvi bazen.

Ovdje kao i u prethodna dva slučaja promatramo slučajeve kada je memorija u potpunosti prazna, i nema zauzetih bazena. Kada bi u memoriji i bilo zauzetih bazena, algoritam se ne bi mijenjao, jer on radi samo nad listom praznih bazena, dok ga se zauzeti bazeni uopće ne tiču.

Dodjeljivanje bloka memorije iz bazena

Za zauzimanje bloka sustav poziva istu funkciju, ali joj šalje drugačije parametre. Ovdje mora poslati listu praznih blokova onog bazena za kojeg želi stvoriti novi blok, željenu veličinu bloka i tip elementa kojeg želi stvoriti.

Opis algoritma dodjele bloka memorije:

1) *Pretražuj listu slobodnih blokova dok su blokovi manji od željene veličine i dok postoji blokova u listi*

2a) ako nisi pronašao odgovarajući blok, vrati vrijednost NULL

2b) ako je pronađeni blok točne veličine, napravi novi blok, postavi mu veličinu, početnu adresu. Postavi marker da više ne sadržava veličinu nego vrijednost -1, tj. oznaku da je zauzet. Pokazivače na prethodni i sljedeći slobodan blok postavi na NULL.

3b) izbaci iz liste slobodnih blok kojeg si pronašao te prespoji listu slobodnih blokova, tako da prethodni blok sada pokazuje na sljedeći blok ovog bloka kojeg smo izbacili

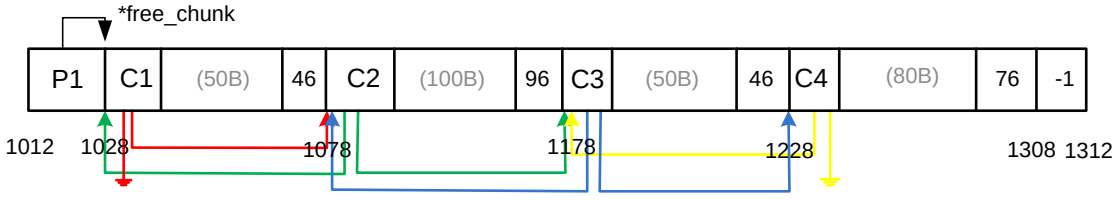
3c) ako je izbačeni blok bio prvi u listi slobodnih, promijeni pokazivač na listu da pokazuje na sljedeći blok na koji je pokazivao izbačeni blok

2c) ako je pronađeni blok veći od željenog, napravi korak 2a, a bloku koji smo podijelili, promijeni iznos markera na novu veličinu, te promijeni početnu adresu

3c) Promijeni pokazivač prethodnog bloka na novu adresu starog bloka

4) vrati novi blok sustavu

Ovdje je bitno napomenuti da makar je sustav zauzeo bazen veličine npr. 300B, što vidimo u sljedećem primjeru na slici 10. Veličina koju mogu iskoristiti blokovi je smanjena za veličinu zaglavlja i markera bazena, što znači da blokovi dobiju 280B memorije za rad.



Početno stanje bazena veličine 300B

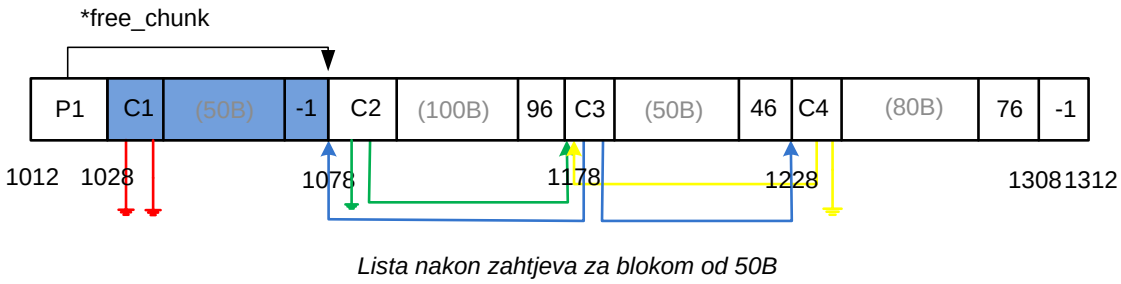
Slika 10. Prikaz bazena sa 4 bloka

Na slici je prikaza bazen veličine 300B koji u sebi već ima stvorena četiri bazena sa veličinama 50B, 100B, 50B i 80B, respektivno. Veličine su ovdje zapisane u zagradama radi lakšeg praćenja zahtjeva.

Sam bazen je zauzet, što se vidi po njegovom markeru koji ima vrijednost -1.

Kako su svi bazeni slobodni vidimo i pokazivače na prethodne i sljedeće pomoću kojih stvaramo listu slobodnih.

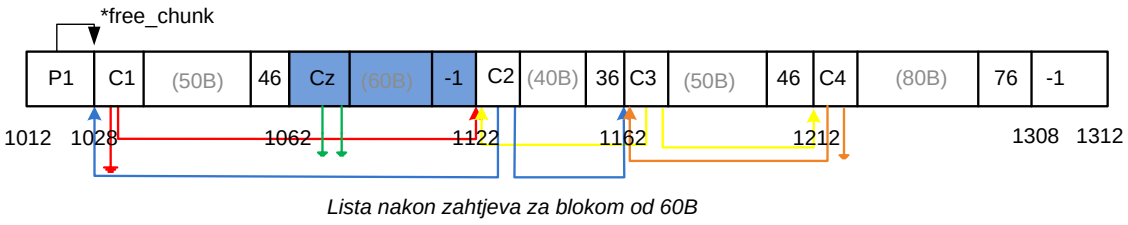
Ako sustavu treba blok veličine 50B on prema algoritmu odmah može iskoristiti prvi bazen. Nakon toga će memorija izgledati kao na slici 11. Radi jednostavnosti prikaza uzeti ćemo da je željena veličina bloka točna veličini prvog bazena.



Slika 11. Memorija bazena nakon prvog zahtjeva

Ovaj slučaj je opisan koracima 2b i 3c.

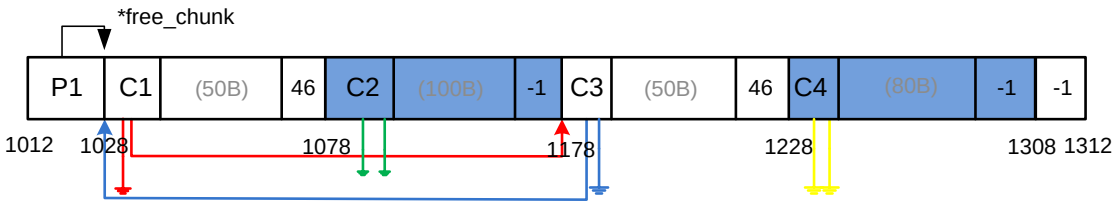
Ako uzmemo da je sustav prvo zatražio blok veličine 60B, onda prvi blok ne odgovara zahtjevu, te algoritam dolazi do drugog bloka, koji je veći od zahtjeva. Pošto je taj drugi blok veći dijeli ga na odgovarajuće veličine i mijenja pokazivače. Taj slučaj je prikazan na slici 12. gdje nastaje novi blok kojeg vračamo a u listi ostaju još uvijek četiri prazna bloka.



Slika 12. Memorija bazena nakon zahtjeva od 60B

Vidimo da je veličina drugog bloka smanjena jer smo uzeli 60B za potrebe sustava.

U sljedećem primjeru, na slici 13., vidimo stanje memorije bazena nakon nekog vremena, nakon što je sustav prvo zatražio blok veličine 100B, a zatim je tražio blok veličine 80B.



Slika 13. Stanje bazena nakon nekoliko zahtjeva

Oslobađanje bazena i bloka memorije

Blokovi i bazeni se mogu i oslobađati. Za to postoji funkcija:

```
int free_element(struct list **list, struct list *element);
```

koja prima listu praznih ili bazena ili blokova, i sam element koji želimo osloboditi. Ovisno o elementu kojeg želi osloboditi sustav šalje ili listu praznih bazena, ili listu praznih blokova.

Kako bi složenost oslobađanja bila što manja, prilikom svakog oslobađanja elementa, bilo to bazen ili blok, on pogleda svoje susjede. Pomoću funkcija `prev_element` i `next_element` provjeri da li su mu susjedni elementi slobodni. Za to koristimo marker. Ove dvije funkcije čitaju marker i ako je marker veći od nula vraćaju u funkciju free_element cijeli element. Ako je marker postavljen na -1 znači da su susjedni elementi zauzeti i funkcije vraćaju NULL vrijednost.

Ova funkcija radi potpuno jednako i prema istom algoritmu i za oslobađanje bazena i za oslobađanje blokova.

Opis algoritma:

1) *Pogledaj da li postoji lijevi susjed i da li je on slobodan*

    2a) ako postoji i ako je slobodan spoji taj element sa elementom kojeg želimo osloboditi.

        3a) Prethodnom elementu povećaj veličinu i promijeni veličinu zapisanu u markeru.

        4a) postavi element da pokazuje na taj prethodni element

    2b) ako ne postoji pogledaj da li postoji i da li je slobodan element koji u memoriji slijedi nakon ovoga kojeg želimo osloboditi

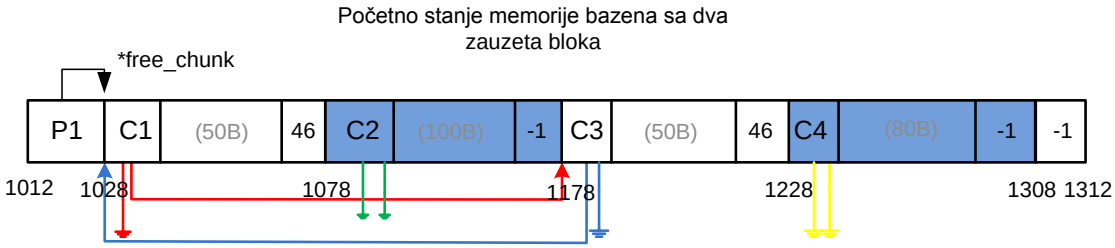
        3b) ako postoji i slobodan je ponovi korak tom slobodnom elementu promijeni početnu adresu na adresu elementa kojeg želimo osloboditi, promijeni vrijednost markera, i veličinu tog već umetnutog sljedećeg elementa.

    2c) ako ni lijevi ni desni susjed nisu slobodni onda samo umetni taj element u listu slobodnih na odgovarajuće mjesto kako bi lista zadržala slaganje po adresama. S tim da se moraju promijeniti pokazivači elementa kojeg želimo umetnuti i njegovih susjeda.

        3c) promijeni marker novo umetnutom elementu na vrijednost njegove veličine

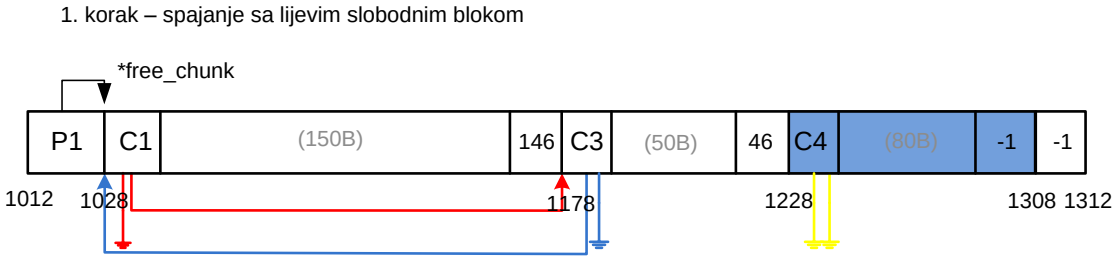
Rad algoritma prikazati ću na oslobađanju blokova u nekoliko primjera opisanih slikama.

Recimo da se nakon nekog vremena u bazenu pojavilo stanje prikazano na slici 14. Vidimo da je bazen veličine 300B, da ima dva zauzeta bazena veličina 100B i 80B i listu slobodnih blokova u kojoj se nalaze dva bazena veličina 50B.



Slika 14. Stanje bazena prije oslobađanja blokova

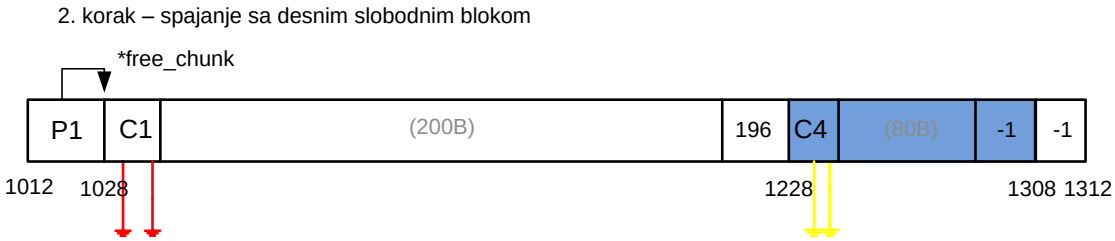
Prvi korak u algoritmu je provjera da li su susjedni elementi slobodni. Ako sustav želi osloboditi blok C2 onda poziva funkciju `prev_element` za C1. Ako mu funkcija vrati vrijednost različitu od NULL, algoritam spaja ta dva bloka i to tako da prethodnom poveća veličinu i promijeni veličinu u markeru kao što je prikazano na slici 15.



Slika 15. Memorija bazena nakon spajanja sa lijevim susjedom

Kada se spaja sa prethodnim elementom pokazivači na susjedne elemente ostaju isti, jer mijenjamo taj prethodni element. C1 će pokazivati i prije i nakon povećanja na iste elemente, jer mi listu nismo mijenjali, samo smo promijenili svojstva jednog elementa.

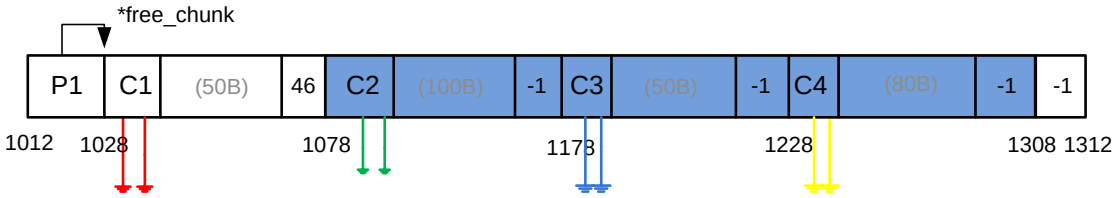
Naravno ako lijevi blok nije bio slobodan ovaj korak se preskače i provjerava se da li je desni susjed slobodan. U ovom slučaju desni susjed je blok C3 i on je slobodan, pa se može obaviti spajanje. Nakon spajanja, promjene veličine i markera izgled memorije bazena prikazan je na slici 16.



Slika 16. Memorija bazena nakon spajanja sa desnim susjedom

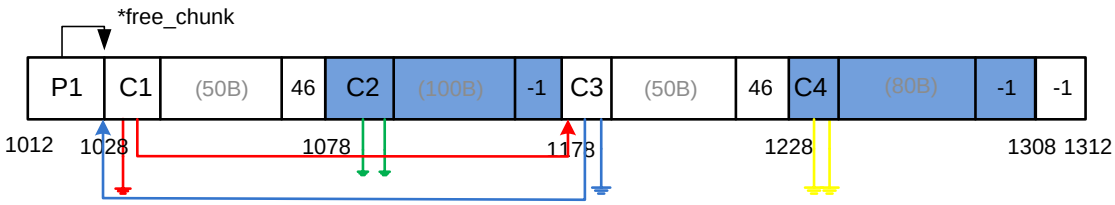
Složenost ovakvog umetanja prilikom oslobađanja je Θ(1), jer svaki element kada se oslobađa pogleda svoje susjede i eventualno se spoji s njima. Tu ne moramo pretraživati listu svih slobodnih kako bi se mogli pozicionirati na pravo mjesto u listi.

Ovaj slučaj je poseban jer nakon što smo provjerili sve susjede i spojili sve moguće prazne blokove, element C1 je ostao jedini blok u listi pa mu moramo promijeniti pokazivače na susjedne elemente na NULL. Naravno ovo je idealni slučaj koji se neće uvijek moći ostvariti. Već se može dogoditi situacija za želimo osloboditi blok koji je okružen zauzetim blokovima kao na slici 17. gdje sustav želi osloboditi blok C3.



Slika 17. Memorija bazena u nekom trenutku izvođenja

Tada oslobađamo taj blok, tako da mu promijenimo marker na vrijednost veličine bloka umanjenu za `sizeof(int)`. Nakon toga namještamo pokazivače liste slobodnih blokova, i njegovih susjeda kao što je prikazano na slici 18.



Slika 18. Oslobađanje bloka koji nema slobodnih susjeda

U ovom slučaju složenost nije Θ(1) zato jer ako element nije okružen slobodnim elementima, potrebno je pronaći mjesto u listi slobodnih, što znači da moramo pretraživati listu slobodnih po adresama dok ne dođemo do dva elementa od kojih je jedan manji a sljedeći veći od adrese elementa kojeg želimo umetnuti.
U najgorem slučaju ovdje složenost može dostići i Θ(n).

Rad funkcija prev_element i next_element

Kao što je prije objašnjeno te dvije funkcije provjeravaju da li su susjedni elementi, bilo bazen ili blok u listi slobodnih, slobodni. Ako jesu onda vrati funkciji adresu tog elementa, a ako nisu onda vraća vrijednost NULL. Tek u tim funkcijama dolaze do izražaja markeri. Funkcije primaju element za koji želimo provjeriti susjeda. Zatim pročitaju što se nalazi na mjestu markera susjeda.

```
void *prev_element(struct list *element);
```

Ako koristimo ovu funkciju ona će od adrese poslanog elementa, pročitati što se nalazi na adresi točno ispred nje, jer zna da tamo mora biti marker.

Tako ako želimo osloboditi element C2 sa slike 18. funkcija `prev_element` će pogledati što se nalaziti na adresi (`1078 - sizeof(int)`) tj. na adresi 1074. Ako je tamo vrijednost -1 znači da je element zauzet i funkcija vraća vrijednost NULL. Ako je tamo vrijednost veća od nule, tada znači da je na tom mjestu upisana veličina sljedećeg elementa umanjena za veličinu markera. Ta veličina je tamo upisana kako bi se funkcija mogla pomaknuti na početnu adresu prethodnog elementa U ovom slučaju na adresi 1074 se nalazi vrijednost 46, što znači da ako

se pomaknemo za 46 adresa u desno ćemo doći do početka prethodnog elementa. I tu adresu će funkcija vratiti funkciji koja ju je pozvala.

Ovdje postoji poseban slučaj, a to je kada pozivamo ovu funkciju za prvi element memorije, bilo to bazena ili čitave memorije. Tada ne možemo gledati adresu ispred jer to bi značilo da će pročitati neku adresu iz zaglavlja, a to nije točno. Stoga u tom slučaju funkcija vraća NULL, jer to nije slobodna memorija, barem ne što se tiče pozivatelja.

```
void *next_element(struct list *element, int size);
```

Poziv ove funkcije je drugačiji jer osim elementa za kojeg provjeravamo susjeda, dobivamo i veličinu tog elementa. Ta veličina nam je potrebna kako bi mogli doći do sljedećeg elementa u memoriji. Početnoj adresi elementa (na slici 18. je to adresa 1078) dodajemo veličinu tog elementa (100 u ovom slučaju) Nakon što je funkcija došla do početne adrese sljedećeg elementa, potrebno je pročitati što se nalazi u njegovom markeru. Kada se pozicioniramo na mjesto markera (na adresu 1224) pomoću naredbe `(char*)next + next->size - sizeof(int);` možemo pročitati sadržaj markera. I onda sljedi isto ispitivanje kao u i funkciji.

Također i ovdje postoji uvijet da ako je element za koji provjeravamo susjede zadnji u memoriji, bilo memoriji bazena, ili sveukupnoj memoriji, tada funkcija vraća NULL, jer ne možemo više čitati adrese u desno.

Proučavajući ovaj algoritam dolazimo do zaključka da može biti izuzetno efikasan kada su bazeni i svi blokovi jednakih ili sličnih veličina. Poželjno je da te veličine budu prilagođene zahtjevima sustava, kako se ne bi trošilo dodatno vrijeme za pretraživanje listi kako bi se pronašao element odogovarajuće veličine.

U suprotnom dobivamo velike gubitke prilikom dodjeljivanja bazena ili blokova jer se troši dodatno vrijeme za pretraživanje listi slobodnih, a nakon toga i za dijeljenje većih bazena / blokova na odgovarajuću veličinu.

5. Zaključak

Ovim radom opisan je jedan način dinamičkog upravljanja memorijom u ugradbenim računalnim sustavima.

Dan je opis vrsti memorija koje se koriste u ugradbenim računalnim sustavima, sa njihovim svojstvima. Uz to proučavani su i zahtjevi ugradbenih računalnih sustava nad memorijom, te u sklopu toga i algoritmi koji se koriste za dinamičko upravljanje memorijom.

Opisana su tri algoritma, *slab* algoritam *buddy* algoritam i *memory pool*. Na jednom mjestu su dani sami algoritmi, ali i njihove prednosti i nedostatci.

Osim samog opisa, detaljnije je razrađen *memory pool* algoritam za kojeg je dano jedno programsko ostvarenje, te je analiziran njegov rad. Pokazuje se da bi algoritam dao najbolje rezultate kada bi se zahtjevi prema pojedinom bazenu ujednačili tj. kada bi veličine traženih blokova bile slične (ili iste).

6. Literatura

[1] Embedded system, http://en.wikipedia.org/wiki/Embedded_system, travanj 2010.

[2] Slab allocation, http://en.wikipedia.org/wiki/Slab_allocation, ožujak 2010.

[3] The Memory Management Reference, http://www.memorymanagement.org

[4] Michael Barr, Programing Embedded Systems in C and C++, 1st edition, O'Reilly, January 1999

[5] Richard Brooksby, 18. Lipanj 2002, Memory pool system project, http://www.ravenbrook.com/project/mps/

7. Sažetak

U ovom radu analizirane su potrebe nad dinamičkim algoritmima upravljanja spremnikom za ugrađena računala. Prikazani su razni tipovi spremnika koji se u tim sustavima mogu koristiti. Nadalje, prikazani su i analizirani najčešći algoritmi dinamičkog upravljanja spremnikom za ugrađene sustave, u koje spadaju: *slab* algoritam, *buddy* algoritam I *memory pool*.

U okviru rada ostvaren je algoritam *memory pool*, te je prikazan njegov način rada na jednostavnom primjeru dodjeljivanja i brisanja bazena i blokova.

Ključne riječi: upravljanje memorijom, ugradbeni računalni sustavi, dinamičko dodjeljivanje memorije, upravljanje memorijom bazenima, Slab algoritam, Buddy algoritam

8. Summary

Title: Memory management subsystem for embedded computer

This paper analyzes need on dynamic memory management algorithms for embedded systems. It shows various types of memory which can be used in those systems. Furthermore, it analyzes basic algorithms for dynamic memory management in embedded systems, like slab algorithm, buddy algorithm and memory pool.

For this paper memory pool algorithm was fully developed. This program shows on basic example of creating and deleting pools and chunks how this algorithm works.

Keywords: memory management, embedded systems, dynamic memory allocation, memory pool management, Slab algorithm, buddy algorithm

9. Privitak

Ovdje se na jednom mjestu nalaze sve funkcije koje su programski ostvarene, i sve strukture objašnjene u tekstu.

Za stvaranje memorije, bazena i blokova korištene su strukture:

```
// struktura koja predstavlja blok unutar bazena
struct chunk {
      int size;                    // veličina bloka
      struct chunk *next_chunk;    // pokazivač na sljedeći blok
      struct chunk *prev_chunk;    // pokazivač na prethodni
                                   //blok u listi praznih blokova
};

// struktura koja predstavlja bazen
struct pool {
      int size;  // veličina bazena
      struct pool *next_pool;     // pokazivač na sljedeči bazena
      struct pool *prev_pool;     // pokazivač na prethodni bazen
      struct chunk *free_chunk;   // pokazivač na listu slobodnih
                                  // blokova
};

// struktura koja predstavlja cijelu memoriju
struct memory{
      int size;                // veličina ukupne memorije
      void *mem_address;       // početna adresa prvog bazena
      struct pool *free_pool; // pokazivač na listu praznih bazena
};

// univerzalna struktura koja se korisit za ostvarivanje
// funkcija stvaranja i unistavanja chunk-ova i pool-ova
struct list {
      int size;             // veličina elementa liste
      void *previous;       // pokazivač na prethodni element liste
```

```
    void *next;          // pokazivač na sljedeći element liste
};
```

Ostvarene funkcije su:

```
// Funkcija koja stvara početnu memoriju sustava
struct memory *create_memory (int memory_size);
// Argumenti funkcije:
//    - memory_size – ukupna željena veličina memorije
// Provratna vrijednost:
//    - adresa početka memorijskog prostora

// Univerzalna funkcija koja stvara novi chunk i pool
struct list *create(struct list **list,int size, short type);
// Argumenti funkcije:
//    - **list – dvostruki pokazivač na listu koju pretražujemo za
//              slobodni element
//    - size – željena veličina novog bloka ili bazena
//    - type – varijabla koja govori funkciji da li stvaramo novi
//             bazen ili blok
// Povratna vrijednost:
//    -adresa početka memorijskog prostora novonastalog elementa


// Univerzalna funkcija za uništavanje pool-a ili chunk-a
int free_element(struct list **list, struct list *element,
                 int type);
// Argumenti funkcije:
//    - **list – dvostruki pokazivač na listu slobodnih elemenata
//    - *element – adresa elementa kojeg želimo osloboditi
//    - type – varijabla koja definira da li se oslobađa bazen ili
//             blok
// Povratna vrijednost:
//    - vrijednost 1 – ako je uspiješno oslobođen element
// Funkcija koja vraća prethodni element ako je on u listi
slobodnih
```

```
void *prev_element(struct list *element);
// Argumenti funkcije:
//    - *element – adresa elementa za kojeg provjeravamo da li je
//                 prethodni element slobodan
// Povratna vrijednost:
//    - početna adresa prethodnog elementa ako je slobodan ili
//      vrijednost NULL ako je element zauzet


// Funkcija koja vraća sljedeći element ako je on u listi
slobodnih
void *next_element(struct list *element, int size);
// Argumenti funkcije:
//    - *element – adresa elementa za kojeg provjeravamo da li je
//                 sljedeći element slobodan
// Povratna vrijednost:
//    - početna adresa sljedećeg elemeenta ako je slobodan ili
//      vrijednost NULL ako je element zauzet


// Funkcija koja postavlja marker na vrijednost size
void end(struct list *element, int size);
// Argumenti funkcije:
//    - *element – adresa elementa za koji postavljamo vrijednost
//                 markera
//    - size – veličina elementa kojem postavljamo marker
// Povratna vrijednost:
//    - funkcija nema povratne vrijednosti
```

Uz to u programskom ostvarenju postoji i glavni program koji simulira radi svih navedenih funkcija.

Program i njegove funkcionalnosti mogu se pronaći na priloženom cd u koricama rada.