# Predavanje 6

Upravljanje memorijom

## 6.1. Uvod

Poznavanje sistema za upravljanje memorijom (*memory management system*) znatno olakšava dizajn aplikacija u okviru embedded sistema. Na primer, u većini embedded aplikacija, dinamičko alociranje memorije korišćenjem ***malloc*** rutine je često korišćeno. Kao rezultat, dolazi do neželjenog efekta koji se naziva fragmentacija memorije (*memory fragmentation*). Ova generička rutina za alokaciju memorije, u zavisnosti od same implementacije, može značajno da utiče na performanse aplikacije. Dodatno, moguće je da ne zadovolji potrebe za alokacijom same memorije u skladu sa potrebama aplikacije.

Veliki broj embedded uređaja ima veoma limitiran broj procesa koji mogu da se izvršavaju u paraleli u datom trenutku, pri čemu imaju na raspolaganju veoma male količine fizičke memorije. Nasuprot ovakvim uređajima, veći embedded sistemi (kao naprimer mrežni ruteri ili web serveri) imaju znatno više fizičke memorije na raspolaganju, ali i oni takođe preferiraju dinamičku raspodelu memorijskih resursa. Nazavisno od tipa embedded sistema, sistem za upravljanje memorijom ima pred sobom zahteve da obezbedi:

1. minimalnu fragmentaciju memorije
2. minimalnu dodatnu aktivnost (*overhead*) kao rezultat upravljanja memorijom
3. determinističko vreme alokacije memorije

## 6.2. Fragmentacija memorije i sabijanje memorije

Programski kod, podaci i sistemski stek okupiraju deo fizičke memorije nakon što se završi inicijalizacija sistema. Preostali deo memorije koristi ili RTOS ili kernel tipično za dinamičko alociranje memorije. Ovaj opseg memorije se naziva *heap*. Blok za upravljanje memorijom sadrži internu strukturu koja čuva informacije o heap memoriji i naziva se *kontrolni blok*. Ove interne informacije tipično uključuju:

1. početnu adresu bloka fizičke memorije korišćenu za dinamičku alokaciju memorije
2. ukupno veličinu ovog bloka fizičke memorije
3. tabelu alokacija koja daje informacije koje zone memorije su u upotrebi, koje su slobodne i kolika je veličina svakog regiona memorije

Kao primer, posmatramo heap koji je podeljen na više malih blokova fiksne veličine. Svaki blok ima veličinu koja je stepen broja 2 kako bi se olakšalo transliranje zahtevane veličine u odgovarajući broj zahtevanih blokova. U ovom primeru, posmatramo blok koji je veličine 32. Dinamička alokacija memorije, odnosno funkcija malloc, dobija ulazni parametar koji specificira

veličinu zahtevane memorije u bajtima. Malloc uvek alocira veći blok, koji se sastoji od jednog ili više manjih blokova fiksne veličine. Veličina ovog memorijskog bloka je barem velika koliko je bilo zahtevano parametrom *malloc* funkcije, a vrlo često veća od njega. Na primer, ako je zahtevano 100 bajtova, veličina vraćenog bloka je 128 (4 bloka po 32 bajta). Kao rezultat, modul koji je zahtevao memoriju neće koristiti 28 bajtova alocirane memorije, što se naziva fragmentacija memorije. Ovaj specifičan slučaj fragmentacije se dodatno zove interna fragmentacija jer se odnosi na fragmentaciju unutar jednog bloka.

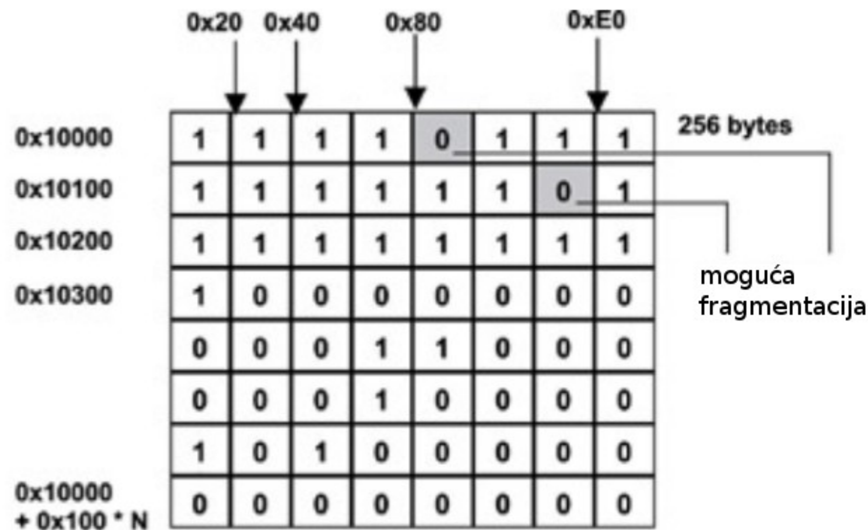
Tabela alokacija može biti predstavljena kao bit-mapa, u kojoj svaki bit reprezentuje 32-bajtni blok. Slika 1 prikazuje stanje tabele alokacija nakon serije pozivanja *malloc* i *free* funkcija. U ovom primeru, heap je veličine 256 bajtova.

*Slika 1: Primer heap memorije veličine 256 bajtova*

U koraku 6 u sistemu postoje dva slobodna bloka veličine 32 bajta svaki, dok nakon koraka 7 umesto tri odvojena slobodna bloka, sistem za upravljanje memorijom prijavljuje jedan slobodan blok veličine 128 bajtova.

Slika 2 prikazuje još jedan primer tabele alokacija memorije. Dva 32-bajtna bloka postoje u sistemu, jedan se nalazi na adresi 0x10080, dok je drugi na adresi 0x101C0. U ovom slučaju nemoguće je dodeliti memoriju ukoliko je zahtevano više od 32 bajta. Ovakva fragmentacija se naziva eksterna fragmentacija jer se odnosi na samu tabelu, a ne na delove memorijskih blokova kao u slučaju od ranije (malloc alocira 128 bajtova od kojih se 28 ne koristi). Jedan način da se reši ovaj problem je sabijanje memorije koja se nalazi između ova dva bloka. Opseg memorije koji počinje na adresi 0x100A0 (neposredno nakon prvog slobodnog bloka) do adrese 0x101BF (koji neposredno prethodi drugom slobodnom bloku) se pomera 32 bajta niže u memorijskom prostoru u opseg adresa 0x10080-0x1019F, što efektivno omogućava kombinovanje dva slobodna bloka u

jedan blok veličine 64 bajta. Ovaj novi slobodan blok se i dalje smatra za fragmentaciju memorije ukoliko buduća alokacija bude zahtevala više od 64 bajta. U skladu sa tim, sabijanje memorije se nastavlja dok svi slobodni blokovi nisu spojeni u jedan veliki blok memorije.




*Slika 2: Primer eksterne fragmentacije i sabijanja memorije*

Postoji nekoliko problema koji mogu da nastanu tokom sabijanja memorije:

- Transfer bloka memorije sa jedne na drugu memorijsku lokaciju je vremenski zahtevan
- Cena kopiranja bloka memorije zavisi od veličine blokova
- Procesi koji koriste premeštane memorijske blokove ne mogu da ih koriste dok se transfer ne završi
- Sabijanje memorije se ranije gotovo nikada nije vršilo u praksi na embedded sistemima, a slobodni memorijski blokovi su se mogli grupisati samo ukoliko su susedni, kao što je pokazano na slici 1
- Sabijanje memorije je dozvoljeno samo ukoliko procesi koji poseduju memorijske blokove koji se premeštaju koriste virtualne adrese. Sabijanje memorije nije dozvoljeno ukoliko procesi koriste fizičke adrese za pristup alociranim memorijskim blokovima.

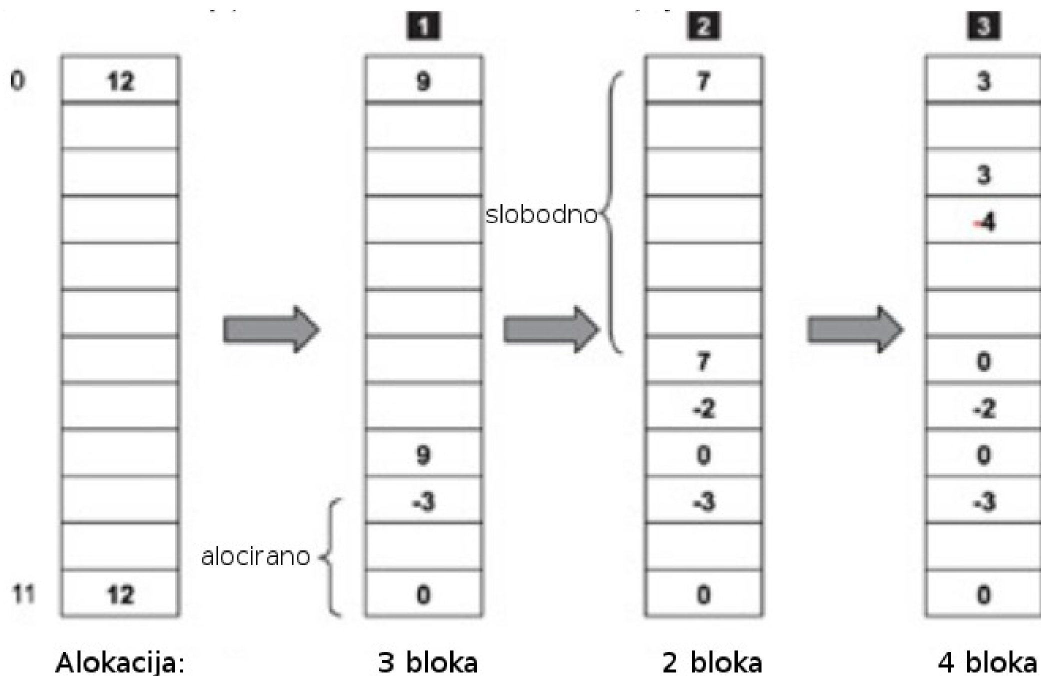
U većini slučajeva, sistem za upravljanje memorijom mora da vodi računa o arhitekturnim specifičnostima u vezi sa memorijskim poravnavanjem. Memorijsko poravnanje (*memory alignment*) se odnosi na restrikcije u pogledu korišćenih adresa u okviru memorijskog prostora. Većina embedded sistema ne podržava pristup višebajtnim podacima na proizvoljnim adresama. Na primer, neke arhitekture zahtevaju višebajtne podatke (npr. integer, i long integer) koji su smešteni na adresama koje su stepen broja 2. Neporavnata memorijska adresa, u ovom smislu, rezultuje izuzetkom memorijskog pristupa (*memory access exception*).

Kao zaključak, efikasan sistem za upravljanje memorijom mora da brzo izvrši sledeće operacije:

1. Utvrdi da li postoji dovoljno veliki blok memorije da zadovolji zahtev za alokacijom memorije. Ova funkcionalnost je deo *malloc* funkcije.
2. Ažurira interne informacije u vezi sa upravljanjem memorijom. Ovo je obaveza ne samo *malloc* već i *free* funkcije.
3. Utvrdi da li upravo oslobođen blok može da bude kombinovan sa susednim blokovima u cilju formiranja većeg bloka memorije koji se može koristiti u narednim alokacijama memorije. Ovo je zadatak *free* operacije.

## 6.3. Primer malloc i free funkcija

Primer koji sledi prikazuje implementaciju algoritma alokacije malloc funkcije. Statički niz celobrojnih vrednosti, koji se naziva *niz alokacija* (*allocation array*), se koristi kao implementacija mape alokacija. Osnovna uloga ovog niza je da odluči da li susedni slobodni blokovi mogu biti spojeni sa postojećim u cilju formiranja većeg slobodnog memorijskog bloka. Svaki podatak ovog niza predstavlja odgovarajuši fiksni blok u okviru memorije. U ovom smislu, niz alokacija je sličan mapi prikazanoj na slici 2, ali ovaj niz koristi drugu šemu enkodovanja. Broj unosa sadržanih u okviru niza je broj blokova fiksne veličine koji je na raspolaganju u okviru upravljanog memorijskog prostora. Na primer, 1MB memorije može biti predstavljen kao 32 768 blokova koji su veličine 32 bajta. U ovom slučaju niz alokacija ima 32768 članova.




*Slika 3: Primer implementacije niza alokacija*

Da bi se pojednostavio primer u cilju boljeg razumevanja algoritma, samo 12 blokova memorije se koriste kao na slici 3.

Neka niz alokacija počinje sa indeksom 0. Pre nego što je ikakva memorija alocirana, jedan veliki blok slobodne memorije postoji koji se sastoji od 12 elementarnih blokova. Niz alokacija koristi jednostavnu šemu enkodovanja kako bi pratio alocirane i slobodne blokove u memoriji. Da bi pokazala na niz slobodnih blokova, pozitivan broj je upisan na mesto prvog i poslednjeg elementa niza koji odgovaraju elementarnim blokovima slobodne memorije. Vrednost ovog pozivnog broja je zapravo broj slobodnih blokova. U našem primeru, u prvom nizu sa leve strane, broj slobodnih elementarnih blokova na početku je 12 i ovaj broj je upisan na indekse 0 i 11 u nizu alokacija.

Upisivanje negativnog broja na mesto prvog elementa u nizu i nule na mesto poslednjeg predstavlja opseg rezervisanih blokova. Broj upisan na prvu lokaciju je jednak umnošku broja -1 i broja rezervisanih elementarnih blokova. U primeru sa slike 3, prva alokacija zahteva 3 bloka. U levom nizu na slici se vidi da je broj -3 upisan na mesto sa indeksom 9, a vrednost 0 na mesto sa indeksom 11, kako bi se označio opseg rezervisanih blokova u skladu sa gore opisanim pristupom. Veličina slobodnog bloka je sada smanjena na 9. Korak tri na slici 3 pokazuje stanje niza alokacija nakon tri alokacije memorije. Ovakav način označavanja (enkodovanja) slobodnih i zauzetih blokova će olakšati spajanje blokova tokom *free* operacije kao što će biti objašnjeno kasnije.

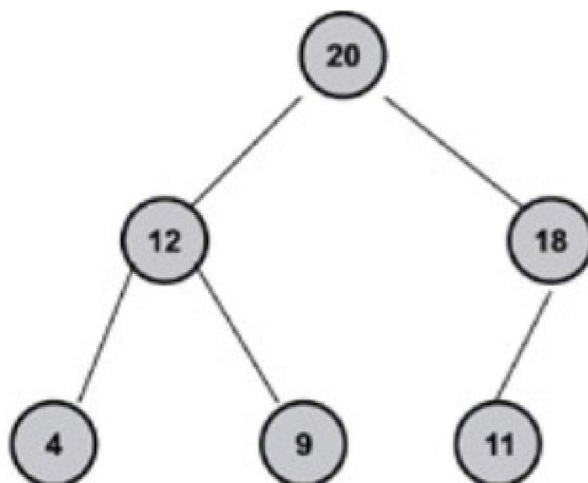
Ne samo da ovakav način označavanja indicira koji su blokovi zauzeti a koji ne, već takođe implicitno olakšava i početnu adresu:

```
start_address = offset + unit_size*index
```

Prilikom alokacije memorije, malloc koristi ovu formulu kako bi izračunao početnu adresu bloka koji se dodeljuje. Na primeru sa slike 3, prva alokacije tri bloka počinje na indeksu 9. Ukoliko je ofset u ovom slučaju 0x10000 a veličina elementarnog bloka 0x20 (32 bajta), adresa izračunata za indeks 9 je

0x10000 + 0x20 * 9 = 0x10120

## 6.4. Brzo pronalaženje slobodnih blokova




*Slika 4: Heap struktura sa podacima o slobodnim blokovima memorije*

Postoji više algoritama za alokaciju memorije:

- *Best Fit Allocation* algoritam: odabira najmanji slobodan blok koji će omogućiti zahtevanu alokaciju
- *Worst Fit Allocation* algoritam: uvek odabira najveći slobodan blok za alokaciju memorije
- *First Fit Allocation* algoritam: odabira prvi slobodan blok koji omogućava zahtevanu alokaciju memorije
- *Next Fit Allocation* algoritam: slično kao First Fit algoritam ali pretragu nastavlja od mesta na kome se stalo prilikom prethodne pretrage

Svaki od gore nevedenih algoritama ima svoje prednosti i mane.

Bez gubitka na opštosti, ovde ćemo razmatrati šemu upravljanja memorijom po kojoj *malloc* uvek pokušava da alocira iz najvećeg dostupnog bloka slobodne memorije.

Ukoliko se memorija alocira na ovaj način, *niz alokacija* prikazan ranije nije adekvatna struktura koja pomaže *malloc* funkciji u cilju pronalaženja najvećeg slobodnog bloka memorije. Ulazi u nizu koji predstavljaju slobodne blokove nisu nikako sortirani po veličini. Pronalaženje najvećeg bloka među njima zahteva uvek prolazak kroz kompletan niz. Zbog ovoga, drugačija struktura podataka se koristi u ovu svrhu, kako bi se ubrzala ova pretraga. Veličine slobodnih blokova iz niza alokacija se čuvaju u okviru *heap strukture podataka* (*heap data structure*), kao što je prikazano na slici 4. Ova struktura je *kompletno binarno stablo*, što podrazumeva:

- svaki nivo binarnog stabla je u potpunosti popunjen osim eventualno poslednjeg;
- ukoliko poslednji nivo nije popunjen, čvorovi se dodaju sa leva na desno.

Osim činjenice toga što je kompletno binarno stablo, heap struktura podataka ima još jednu dodatnu osobinu: vrednost sadržana u nekom čvoru (*ključ*) ne može biti manja od vrednosti koje se nalaze u bilo kojem od njegovih čvorova-dece (*child nodes*). Ovakva struktura se takođe u literaturi naziva i *Max-heap* struktura (*Min-heap* je slična struktura sa osobinom da se u čvoru-detetu ne može naći vrednost koja je manja, za svaki čvor iz stabla). U procesu dodenljivanja memorije, veličina svakog slobodnog bloka je ključ koji se koristi prilikom organizovanja heap-a. Kao rezultat, najveći blok je uvek na vrhu heap strukture, zbog čega je ovakva struktura izuzetno pogodna za implementaciju *Worst Fit* algoritma, koji uvek dodeljuje memoriju iz najvećeg slobodnog bloka. Malloc algoritam koristi ovaj najveći blok slobodne memorije, koji se nalazi na vrhu heap strukture, prilikom alociranja memorije. Memorija koja preostane nakon alokacije se upisuje u heap strukturu u zavisnosti od toga koliko je zaista memorije alocirano. Heap struktura je rearanžirana kao poslednji korak u procesu alokacije memorije.

Iako je veličina svakog opsega memorije ključ koji se koristi prilikom formiranja i rearanžiranja heap strukture, svaki čvor u binarnom stablu osim veličine bloka sadrži i dodatni podatak koji se odnosi na početni indeks u okviru niza alokacija, o kojem je bilo reči u prethodnoj sekciji. Malloc operacija podrazumeva sledeće korake:

1. Pretraži heap strukturu kako bi utvrdio da li postoji slobodan blok koji će omogućiti zahtevanu alokaciju memorije - zapravo samo pogledaj vrh *heap* strukture;
2. Ukoliko takav blok ne postoji, vrati grešku procesu koji je pozvao malloc;
3. Preuzmi početni indeks iz niza alokacija sa vrha heap strukture;
4. Ažuriraj niz alokacija označavanjem novo-alociranog bloka, kao što je prikazano na slici 3;
5. Ukoliko je ceo blok iskorišćen za alokaciju memorije, ukloni vrh iz heap strukture (dole će biti rečeno kako). U suprotnom, ažuriraj ključ vrha heap strukture;
6. Rearanžiraj heap strukturu ako je potrebno (npr. nakon alociranja, na vrhu se više ne nalazi najveći blok slobodne memorije).

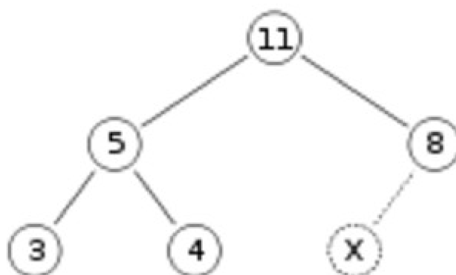
Prilikom alociranja i oslobađanja memorije, operativni sistem vrši tri operacija nad heap strukturom:

- dodavanje čvora u strukturu (nakon nastanka novog bloka slobodne memorije);
- uklanjanje čvora sa vrha strukture (nakon što se kompletan najveći slobodni blok memorije alocira kao odgovor na malloc)
- ažuriranje čvora na vrhu strukture (nakon što nije kompletan najveći slobodni blok memorije alocira kao odgovor na malloc, već samo njegov deo)
- uklanjanje čvora iz strukture (ne sa vrha) je neophodno kada nakon oslobađanja jednog bloka, on može spojiti sa oba susedna bloka
- ažuriranje čvora u strukturi (ne na vrhu) je neophodno kada nakon oslobađanja jednog bloka, on može spojen sa susednim blokom

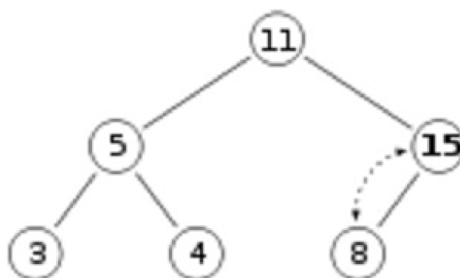
**Dodavanje čvora** u heap strukturu se vrši u skladu sa sledećim algoritmom:

1. Dodaj element u najniži nivo kompletnog binarnog stabla (vodeći računa o pravilu da se elementi dodaju sa leva na desno)
2. Uporedi ključ dodatog elementa sa njegovim roditeljem, ako je sve u redu u skladu sa osobinom da je svaki ključ svakog roditelja veći od ključeva njegove dece, stani ovde
3. Ukoliko nije zadovoljeno, zameni poslednje dodati element sa njegovim roditeljem i vrati se na korak 2.

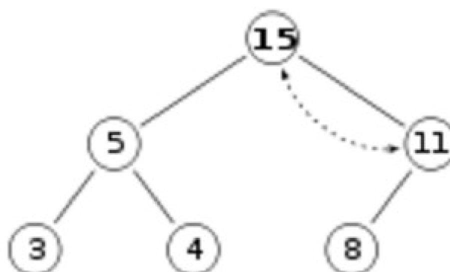
Kao primer pogledajmo sliku ispod. Pretpostavljamo da se dodaje čvor sa ključem 15.




Zatim upoređujemo vrednost ključa dodatog čvora sa njegovim roditeljem i pošto je 15 veće od 8, vršimo zamenu:




Nastavljamo ovaj postupak dalje sve dok ne bude zadovoljeno da poslednje dodati čvor sadrži ključ koji je manji od ključa sadržanog u njegovom roditelju:



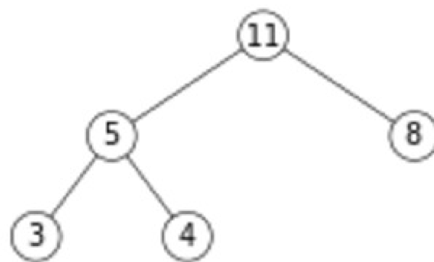

Treba primetiti da nema potrebe porediti sa levim detetom nakon poslednjeg koraka. Obzirom na činjenicu da je heap struktura bila validna i pre poslednje zamene čvorova, to je značilo da je ključ sadržan u okviru levog čvora deteta manji od čvora na vrhu heap strukture (11>5). Samim tim, kada

se zameni 11 sa većim brojem 15, i dalje će, zbog tranzitivnosti, biti zadovoljeno da je 15 > 5.

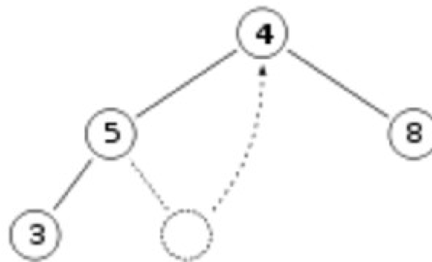
**Brisanje čvora** sa vrha heap strukture se vrši u skladu sa sledećim algoritmom:

1. Postavi na vrh strukture poslednji element poslednjeg reda heap strukture
2. Uporedi novo postavljeni čvor sa svojom decom, ako su u koretnom poretku stani
3. Ukoliko nisu korektno poređani, zameni čvor sa detetom koji sadrži veći ključ i vrati se na korak 2.

Na primeru iste heap strukture od malopre:




nakon uklanjanja čvora sa vrha strukture dobijamo:




nakon čega poredimo 4 sa 5 i 8 i zamenjujemo sa većim od ta dva, te je rezultat:

**Ažuriranje vrha** heap strukture je neophodno kada je samo deo najvećeg slobodnog bloka memorije alociran, i vrši se na sledeći način:

1. Uporedi novo-ažurirani čvor na vrhu strukture sa svojom decom, ako su u koretnom poretku stani
2. Ukoliko nisu korektno poređani, zameni čvor sa detetom koji sadrži veći ključ i vrati se na

korak 1.

**Brisanje čvora** iz strukture se vrši u skladu sa sledećim algoritmom:

1. Postavi na mesto obrisanog čvora strukture poslednji element poslednjeg reda heap strukture (*x*)
2. Uporedi čvor *x* sa svojim roditeljem, ako nisu u koretnom poretku predji na 5
3. Uporedi čvor *x* sa svojim čvorovima-decom, ako su u koretnom poretku stani
4. Zameni čvor *x* sa svojim detetom koji sadrži veći ključ i vrati se na korak 3
5. Zameni čvor *x* sa svojim roditeljem
6. Uporedi *x* sa roditeljem, ako su u koretnom poretku stani, u suprotnom vrati se na 5

Suština ovog algoritma leži u činjenici da kada se jednom postavi umesto obrisanog čvora, čvor *x* se može "kretati" ili na dole ili na gore, dok ostatak strukture nema potrebe premeštati (ne može se desiti da *x* bude istovremen veći od svog roditelja i manji od svog deteta jer tada heap struktura koja je postojala pre operacije brisanja ne bi bila validna).

**Ažuriranje čvora** iz strukture se vrši u skladu sa sledećim algoritmom:

1. Uporedi ažurirani čvor *x* sa svojim roditeljem, ako nisu u koretnom poretku predji na 4
2. Uporedi čvor *x* sa svojim čvorovima-decom, ako su u koretnom poretku stani
3. Zameni čvor *x* sa svojim detetom koji sadrži veći ključ i vrati se na korak 2
4. Zameni čvor *x* sa svojim roditeljem
5. Uporedi *x* sa roditeljem, ako su u koretnom poretku stani, u suprotnom vrati se na 4

Suština ovog algoritma opet leži u činjenici da, nakon ažuriranja, čvor *x* može da se "kreće" ili na dole ili na gore, dok ostatak strukture nema potrebe premeštati. Ovo je rezultat činjenice da se ne može desiti da *x*, nakon ažuriranja, bude istovremeno veći od svog roditelja i manji od svog deteta jer tada heap struktura koja je postojala pre operacije ažuriranja ne bi bila validna.

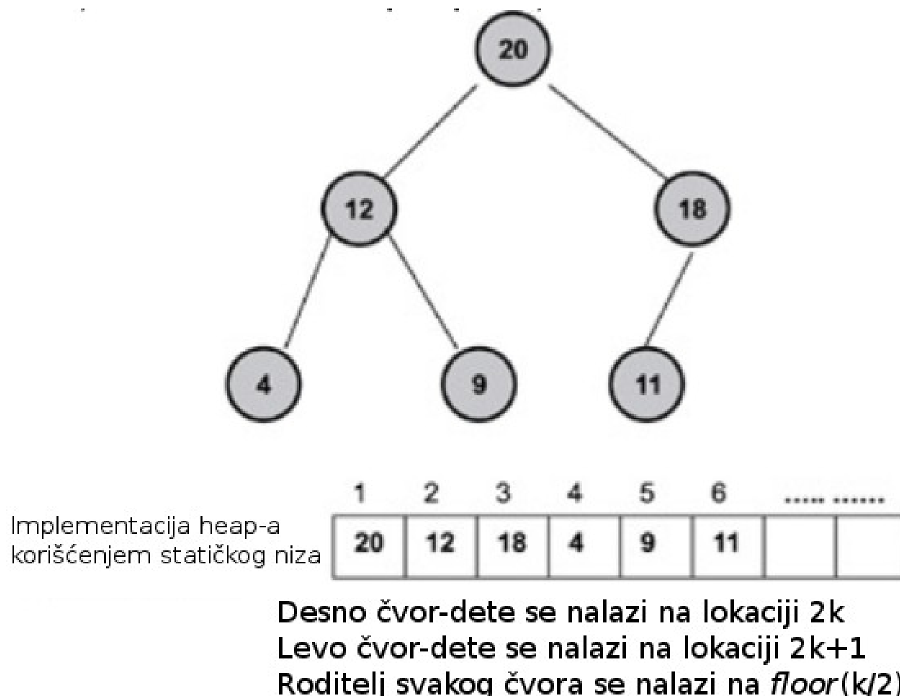
Pre nego što je memorija alocirana, heap ima samo jedan čvor, što označava da je kompletan memorijski prostor dostupan kao jedan veliki slobodan blok. Heap će i dalje imati samo jedan čvor sve dok se memorija alocira bez oslobađanja memorije (samo se menja ključ, tj. vrednost upisana u čvor), ili u slučaju kada se svaki oslobođeni blok memorije može spojiti (eng. *merge*) sa susednim slobodnim blokom.

Osobine heap strukture omogućavaju jednostavnu implementaciju u vidu korišćenja statičkog niza koji se naziva *heap niz* (*heap array*) kao što je prikazano na slici 5. Indeks niza počinje od 1 umesto od 0 kako bi pojednostavio implementaciju. U ovom primeru, šest slobodnih blokova veličina 20, 18, 12, 11, 9 i 4 su dostupni. Sledeća alokacija memorije koristi blok veličine 20 bez obzira na

veličinu zahtevanog bloka. Heap niz je kompaktan način zapisa *kompletnog* binarnog stabla, koji ne koristi pokazivače na decu čvorove, već se roditelj-dete relacija oslikava kroz indekse unutar niza:

- levi čvor-dete čvora na indeksu $k$ se nalazi na indeksu $2k$
- desni čvor-dete čvora na indeksu $k$ nalazi na $2k+1$

Sa druge strane za svaki čvor na indeksu k, roditelj se nalazi na poziciji *floor*(k/2), gde *floor*($x$) funkcija mapira realan broj $x$ na najveći celobrojni broj manji od $x$.



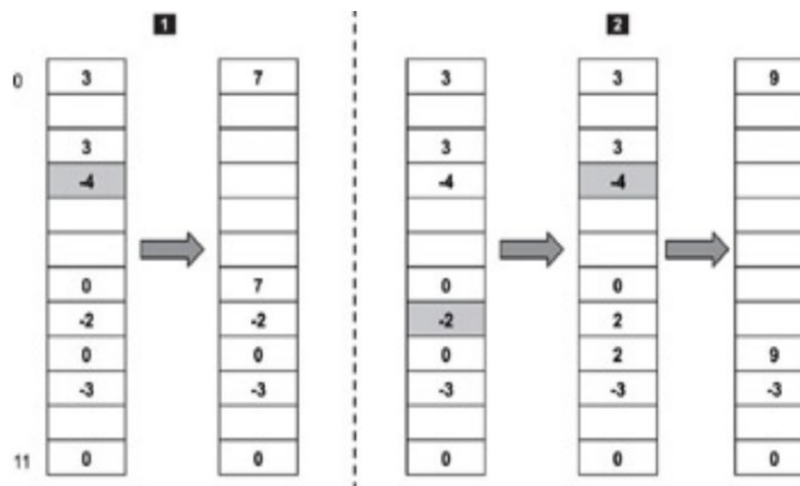

*Slika 5: Implementacija heap-a korišćenjem statičkog niza*

## 6.5. Free operacija

Osnovna funkcija free operacije je da odredi da li blok oslobođene memorije može biti spojen sa svojim susedima. Pravila spajanja su:

1. Ukoliko je *index* početni indeks oslobođenog bloka i različit je od 0, proveri vrednost niza alokacija sa indeksom (*index*-1). Ako je ova vrednost pozitivan broj, ovaj sused može biti spojen;
2. Ako (*index+number_of_blocks*) ne prelazi preko maksimalnog indeksa niza, proveri vrednost upisanu na indeksu (*index+number_of_blocks*). Ako je ova vrednost pozitivna, sused može biti spojen.

Ova dva pravila su ilustrovana na primeru sa slike 6.



*Slika 6: Free operacija i prispajanje slobodnih blokova*

Slika prikazuje dva scenarija vredna diskusije. U prvom scenariju, blok koji počinje na indeksu 3 je blok koji se oslobađa. U skladu sa pravilom 1 od gore, proverava se vrednost upisana u nizu na indeksu 2. Ova vrednost je 3, što znači da susedni blok može biti prispojen. Vrednost 3 dodatno indicira da je veličina ovog bloka 3 veličine elementarnog bloka. Blok koji je oslobođen počinje na indeksu index=3 i veličine je 4 (zbog broja -4 koji je upisan). Dakle, po pravilu 2, treba proveriti vrednost sa indeksa 7. Ova vrednost je -2, što znači da je ovaj susedni blok u upotrebi i da ne može biti prispojen. Rezultat free operacije u ovom scenariju je prikazan drugim nizom sleva na slici 6.

U drugom scenariju, blok sa indeksom 7 se oslobađa. Prateći pravilo 1, gleda se vrednost na indeksu 6, koja je 0. To znači da je susedni blok u upotrebi i ne može se prispojiti. Prateći pravilo 2, proverava se vrednost sa adrese 9, koja je -3, što opet znači da ovaj susedno blok ne može biti prispojen. Novo oslobođeni blok ostaje nezavisan slobodan blok u memorijskom prostoru. Nakon još jednog oslobađanja bloka koji se nalazi na indeksu 3, rezultat niza alokacija je prikazan sa kao prvi niz sa desne strane na slici 6.

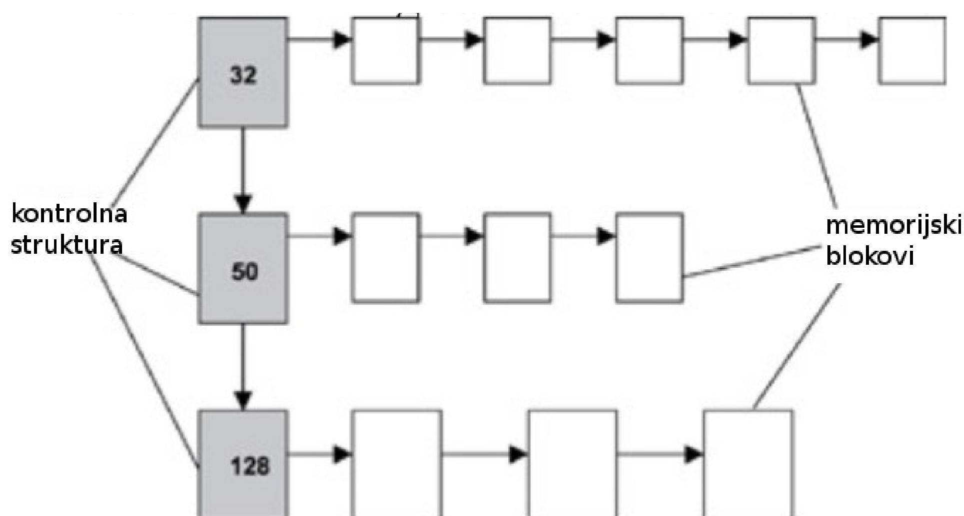
Osim ažuriranja niza alokacija, nakon svakog bloka koji je oslobođen, heap mora biti takođe ažuriran. U skladu sa tim, *free* operacija podrazumeva i sledeće korake:

1. Ažuriraj niz alokacija i spoj susedne blokove ako je moguće;
2. Ako novo-oslobođeni blok ne može biti spojen sa nekim od suseda, dodaj još jedan unos u heap niz (još jedan čvor u heap strukturu);
3. Ukoliko novo-oslobođeni blok može biti spojen sa jednim od suseda, element u heap nizu koji predstavlja tog suseda mora biti ažuriran, a sam heap niz rearanžiran u skladu sa novom veličinom slobodnih blokova;
4. Ako se novo-oslobođeni blok može spojiti sa oba suseda, element heap niza koji predstavlja jednog od suseda mora biti uklonjen, dok se element koji predstavlja drugog suseda mora ažurirati i sam heap niz rearanžirati u skladu sa novom veličinom slobodnih blokova.

## 6.6. Upravljanje memorijom sa fiksnim veličinama blokova

Drugačiji pristup u vezi sa upravljanjem memorijom koristi metod domena sa fiksnim veličinama memorijskih blokova (eng. *fixed-size memory pools*).

Kao što je prikazano na slici 7, raspoloživ prostor memorije je podeljen na domene različitih veličina. Svi blokovi u okviru jednog domena su iste veličine. Na primeru sa slike 7, memorijski pristor je podeljen na tri domena blokova veličine 32, 50 i 128, respektivno. Svaki domen ima svoju kontrolnu strukturu koja održava informacije kao što su veličina bloka u okviru domena, ukupan broj raspoloživih blokova i broj slobodnih blokova. U ovom primeru, domeni memorijskih blokova su povezani međusobno i sortirani po veličini. Pronalaženje najmanjeg adekvatnog bloka za alokaciju podrazumeva pretragu liste domena, nakon čega se proverava kontrolna struktura odgovarajućeg domena u potrazi za prvim adekvatnim blokom.




*Slika 7: Fiksne veličine blokova za alokaciju memorije*

Uspešna alokacija rezultuje blokom koji je uklonjen iz domena, a uspešna dealokacija rezultuje blokom koji je vraćen nazad u domen memorijskih blokova. Sam domen memorijskih blokova je jednostruko povezana lista. Dakle, alokacija i dealokacija se vrše na početku liste.

Ovaj metod nije fleksibilan kao algoritam predstavljen ranije kada je bilo reči o dinamičkom alociranju memorije. U real-time embedded sistemima, zahtevi za memorijom od pojedinačnih procesa često zavise od uslova u kojima se proces izvršava, a ovo okruženje je vrlo često vrlo dinamično. Metod sa fiksnim veličinama blokova neće obezbediti dobre performanse u ovakvom slučaju, obzirom da je gotovo nemoguće anticipirati veličine memorijskih blokova koje će proces najčešće zahtevati. Ovakav problem rezultuje uvećanjem interne memorijske fragmentacije po svakoj alokaciji. Dodatno, broj blokova koji su na raspolaganju za alociranje za svaki od domena je gotovo nemoguće predvideti. U većini slučajeva, memorijski domeni su konstruisani u skladu sa dodatnim pretpostavkama. Rezultat ovoga su domeni koji su uglavnom nekorišćeni, dok su drugi domeni preopterećeni.

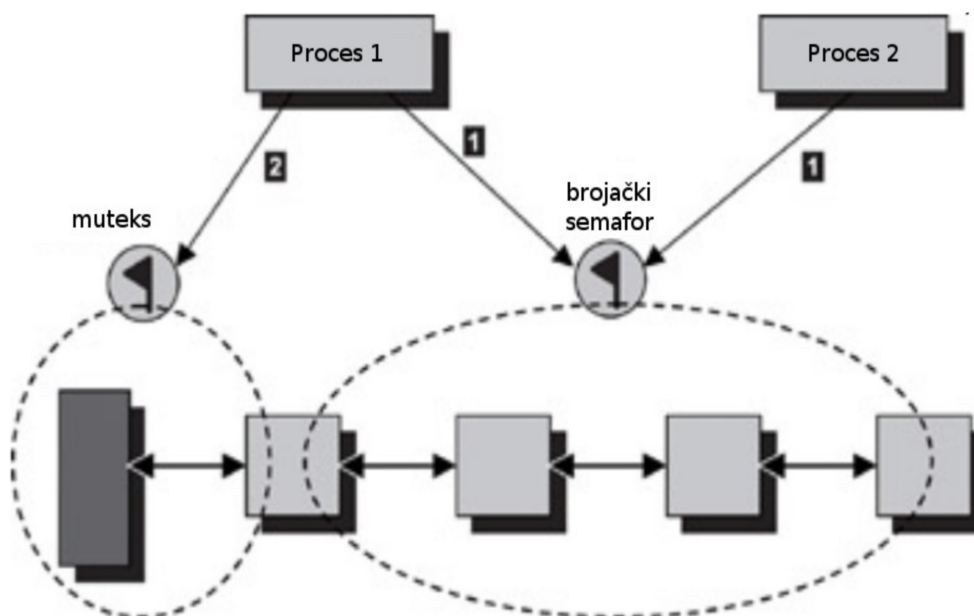
Sa druge strane, ovakva šema alokacije memorije može u određenim slučajevima redukovati interne fragmentacije i obezbediti visoku iskorišćenost u slučaju statičkih embedded aplikacija. Ove aplikacije su aplikacije sa predvidljivim okruženjem, poznatim brojem procesa koji se izvršavaju na početku pokretanja aplikacije i inicijalno poznatim zahtevanim veličinama memorijskih blokova.

Još jedna prednost ovakvog pristupa se odnosi na činjenicu da je više deterministički u poređenju sa heap metod algoritmom. U slučaju heap metode, svaki *malloc* i *free* mogu potencijalno izazvati rearanžiranje heap-a. U slučaju domena memorijskih blokova, uklanjanje i dodavanje blokova iz domena (liste blokova) zahteva konstantno vreme, jer ovakav pristup sa domenima ne zahteva restruktuiranje.

## 6.7. Blokirajuće i ne-blokirajuće alociranje memorije

*Malloc* i *free* funkcije ne omogućavaju pozivajućim procesima da blokiraju čekajući na memoriju da postane dostupna. U većini real-time embedded sistema, procesi se takmiče ravnopravno za limitirane memorijske resurse. Vrlo često je nedostatak memorije u datom embedded sistemu samo privremen. Za neke procese kada zahtev za memorijom ne uspe, proces mora da se vrati korak nazad u izvršavanju i vrlo često da se restartuje. Ovakav ishod može biti veoma skup i ako procesi imaju u vidu da je nedostatak memorije možda samo privremen, efikasnije bi bilo da proces sačeka na raspoloživu memoriju pre nego da se restartuje.

U praksi, dobro dizajnirana alokacija memorije treba da omogući alokaciju koja dozvoljava blokiranje zauvek, blokiranje sa tajm-autom ili neblokirajući poziv za alokaciju memorije. Ovde će na primeru domena memorijskih blokova fiksne veličine biti demonstrirano kako implementirati blokirajuću alokaciju memorije.




*Slika 8: Blokirajuće pristupanje memoriji*

Kao što je prikazano na slici 8, blokirajuća memorijska alokacija može biti implementirana korišćenjem brojačkog semafora i muteks-a. Ovi primitivni objekti za sinhronizaciju su kreirani za svaki domen memorijskih blokova i čuvaju se u okviru kontrolne strukture. Brojački semafor se inicijalizuje sa brojem ukupnog broja dostupnih memorijskih blokova prilikom kreiranja domena. Memorijski blokovi su alocirani i oslobođeni sa početka liste.

Više procesa može da pristupa slobodnim blokovima u okviru domena. Kontrolna struktura je ažurirana svaki put kada se izvrši alokacija ili dealokacija memorije. Muteks garantuje da će proces imati ekskluzivno pravo pristupa listi slobodnih blokova i kontrolnoj strukturi. Proces može da čeka na blok da postane dostupan, zauzme blok i onda nastavi sa izvršavanjem. U ovom slučaju brojački semafor se koristi.

Da bi alokacija bila uspešna, proces mora najpre uspešno da zauzme brojački semafor, nakon čega sledi uspešno zauzimanje muteksa.

Uspešno zauzimanje brojačkog semafora, rezerviše dostupne blokove iz domena. Proces najpre pokušava da zauzme semafor. Ako nema dostupnih blokova, proces će biti blokiran čekajući brojački semafor. Ukoliko je resurs dostupan, proces zauzima semafor uspešno, a brojač semafora se umanjuje za jedan. U ovom trenutku proces je rezervisao blok iz domena ali još uvek nije dobio blok na raspolaganje.

Nakon toga, proces pokušava da zaključa muteks. Ukoliko drugi proces u istom trenutku preuzima blok memorije ili oslobađa prethodno alocirani blok memorije, muteks je zaključan i proces mora da blokira čekajući na muteks. Nakon što se muteks otključa, proces preuzima resurs iz liste. Brojački semafor je oslobođen (brojač uvećan za jedan) svaki put kada proces završi sa korišćenjem memorijskog bloka.

Pseudo kod koji opisuje alokaciju memorije je:

```
Acquire(Counting_Semaphore)
Lock(mutex)
Retrieve the memory block from the pool
Unlock(mutex)
```

Pseudo kod za dealokaciju memorije je:

```
Lock(mutex)
Release the memory block back to into the pool
Unlock(mutex)
Release(Counting_Semaphore)
```

## 6.8. Hardverska jedinica za upravljanje memorijom (*Hardware Memory Management Unit*)

Upravljanje virtualnom memorijom do sada nije bilo diskutovano. Virtualna memorija je tehnika koja omogućava da se masivna memorija (na primer hard-disk) prikaže aplikaciji kao da je deo RAM memorije. Virtualni adresni prostor (takođe nazvan *logički adresni prostor*) je veći od fizičkog memorijskog adresnog prostora. Ova karakteristika omogućava programima koji su veći od raspoložive fizičke memorije da se izvršavaju. Jedinica za upravljanje memorijom (Memory Management Unit - MMU) obezbeđuje sekoliko funkcija. Prvo, MMU translira virtualne adrese u fizičke adrese pri svakom pristupu memoriji. Drugo, MMU obezbeđuje zaštitu memorije.

Translacija adresa je funkcija koja se razlikuje od jedne MMU do druge. Većina komercijalnih RTOS-a ne podržava implementaciju virtualnih adresa, tako da će ovde biti pomenuta protekcija memorijskih blokova, koju većina RTOS-a omogućava.

Ukoliko je MMU omogućen u okviru embedded sistema, fizička memorija je tipično podeljena u blokove koji se zovu *strane* (eng. *pages*). Skup atributa je asociran sa svakom memorijskom stranom. Informacije u okviru atributa najčešće uključuju:

- da li strana sadrži kod (izvršne instrukcije) ili podatke
- da li stranica dozvoljava čitanje, pisanje, izvršavanje koda, ili kombinaciju ova tri
- da li se strani može pristupiti kada CPU nije u privilegovanom modu izvršavanja, u slučaju kada je CPU u privilegovanom modu, ili u oba slučaja.

Svaki pristup memoriji se vrši kroz MMU. U skladu sa tim, hardver forsira pristupe memoriji u skladu sa atributima asociranim svakoj pojedinačnoj stranici. Na primer, ukoliko proces pokuša da upiše sadržaj u region memorije koji je predviđen samo za čitanje, operacije se smatra ilegalnom i MMU je neće dozvoliti. Rezultat ovakve operacije će aktivirati izuzetak pristupa memoriji (*memory access exception*).