UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

# UPRAVLJANJE S POMNILNIKOM

SEMINARSKA NALOGA

Avtor: Tikei Radovac

Predmet: Sistemska programska oprema

Mentor: Tomaž Dobravec

Ljubljana, januar 2025

# KAZALO

## POVZETEK

**Naslov**: Upravljanje s pomnilnikom

**Avtor**: Tikei Radovac

Seminarska naloga obravnava upravljanje pomnilnika. Osredotočena je predvsem na upravljanje pomnilnika na programskem nivoju. Predstavlja osnove alokacije in dealokacije pomnilnika za programe, predvsem za programski jezik C, ter pogoste izzive in napake pri le tem. Prikaže tudi najpogostejše algoritme za samodejno upravljanje s pomnilnikom (garbage collection). Na koncu pa še obravnava mojo lastno implementacijo enostavnega alokatorja v programskem jeziku C.

**Ključne besede**: pomnilnik, smetar (garbage collector), kopica, algoritem.

# 1 UVOD

Skozi svoje izkušnje programiranja sem opazil, da se velikokrat sprašujem o delovanju pomnilnika. Od kje sploh ta pomnilnik mojemu programu, zakaj se zdi kot da je pomnilnika za lokalne spremenljivke veliko manj kot za dinamično dodeljene spremenljivke, kdo programu dodeli pomnilnik, kako vse sploh deluje? Takšna vprašanja sem si zastavljal predvsem med programiranjem v C-ju, medtem ko sem programiral v Javi takšnih vprašanj nisem imel, saj Java veliko naredi samodejno, zato me je zanimalo kako to deluje. Odločil sem se, da bom vse podrobneje raziskal skozi to seminarsko nalogo.

V seminarski nalogi se bom predvsem osredotočil na upravljanje s pomnilnikom na programskem nivoju, čeprav bom malo opisal tudi, kako je s tem na strojnem nivoju in na nivoju operacijskega sistema. Raziskal bom kopico in sklad, vendar bom bolj osredotočen na kopico in dinamično dodeljevanje pomnilnika. Opisal bom postopek zahteve po pomnilniku in sproščanje pomnilnika, ter izzive in napake pri tem. Zadevo bom podrobneje pogledal na primeru programskega jezika C. Raziskal bom tudi področje smetarjev oz. garbage collectorjev (GC) in različne algoritme, ki so pri tem uporabljeni, ter kako to izgleda pri različnih programskih jezikih. Na koncu bom še sam poskusil implementirat svoj enostavni alokator v C-ju.

# 2 UPRAVLJANJE POMNILNIKA

V vsakem računalniku se nahaja glavni pomnilnik, ki skrbi za shranjevanje podatkov, do katerih dostopajo različni procesi in naprave. Večina teh procesov se izvaja sočasno, zato jih je treba med izvajanjem hraniti v glavnem pomnilniku za zagotavljanje optimalnega delovanja. Ker si procesi delijo pomnilnik, je z njim treba ustrezno opravljati. Upravljanje pomnilnika je proces nadzora in usklajevanja glavnega pomnilnika računalnika. S tem zagotovimo, da imajo operacijski sistemi, aplikacije in drugi tekoči procesi na voljo dovolj pomnilnika za nemoteno izvajanje svojih operacij. Potrebno je, da se pri tem upošteva omejitve zmogljivosti pomnilniške naprave, da se sprosti pomnilniški prostor, kadar ni več potreben, ali pa se razširi prek navideznega pomnilnika. Upravljanje pomnilnika deluje na treh ravneh: strojna oprema, operacijski sistem in program/aplikacija.

## 2.1 STROJNA OPREMA IN OPERACISJKI SISTEM

Na ravni strojne opreme se upravljanje pomnilnika ukvarja s fizičnimi komponentami, ki shranjujejo podatke. Večino upravljanja na fizični ravni opravi enota za upravljanje pomnilnika (Memory managment unit ali MMU). Ta nadzira procesorjeve operacije nad pomnilnikom in predpomnilnikom. Zelo pomembno vlogo izvaja MMU pri prevajanju logičnih naslovov, ki jih uporabljajo tekoči procesi, v fizične naslove na pomnilniški napravi.

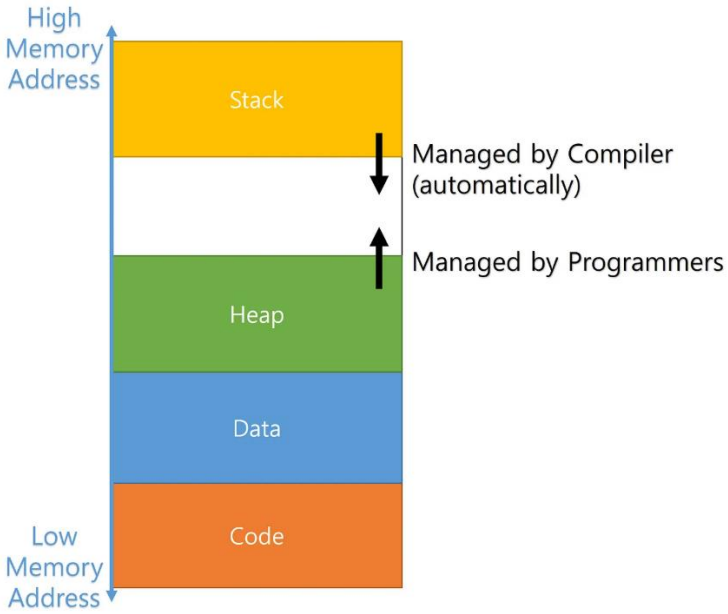
Na ravni operacijskega sistema pa upravljanje pomnilnika vsebuje dodeljevanje in nenehno prerazporejanje določenih pomnilniških blokov posameznim procesom in programom. OS nenehno prestavlja procese med pomnilnikom in napravami za shranjevanje (trdi disk ali SSD), da se zdi kot da je pomnilnika več kot v resnici. Določa tudi, kateri procesi dobijo pomnilniške vire, koliko jih posamezen proces dobi ter kdaj so ti viri dodeljeni.

# 3 URAVLJENJE POMNILNIKA NA PROGRAMSKEM NIVOJU

Na ravni programa se upravljanje pomnilnika izvaja med postopkom razvoja aplikacije, pri tem ga nadzoruje aplikacija sama, namesto da ga centralno upravlja OS ali MMU. Po začetni dodelitvi pomnilnika s strani OS, aplikacija prevzame odgovornost za razporejanje znotraj tega prostora. To pomeni, da bo aplikacija dodeljevala in sproščala pomnilnik za svoje podatkovne strukture, vendar znotraj omejitev, ki jih določa OS. Del tega upravljanja je prepuščen programerju (npr. delo s kopico), medtem ko upravljanje sklada običajno avtomatsko nadzoruje prevajalnik.

## 3.1 POMNILNIK PROGRAMA

Za začetek izvajanja programa mora nalagalnik programov naložiti njegove izvršljive binarne datoteke in vse z njimi povezane vire iz trdega diska v RAM. Po naložitvi programa se PC nastavi na naslov prvega ukaza programa, nakar se program zažene. Vse spremenljivke, funkcije, parametri in podobno se shrani v pomnilnik na sklad ali kopico, ki so dodeljeni segmentom RAM-a znotraj primarnega pomnilnika. Slika 1 na zelo preprost način prikazuje, kako izgleda pomnilnik programa, potem ko je bil naložen v pomnilnik. Na sliki vidimo še podatkovni segment (Data), ki je namenjen shranjevanju globalnih in statičnih spremenljivk.




Slika 1: Prikaz pomnilnika za program

### 3.1.1 Sklad

Sklad je območje pomnilnika, ki je dodeljeno znotraj RAM-a za proces. Uporablja se za shranjevanje začasnih podatkov. Deluje kot LIFO (last-in-first-out) podatkovna struktura za podatke. Če je spremenljivka zadnji element v skladu, bo prva odstranjena. Primeri podatkov, ki so shranjeni v skladu, so: lokalne spremenljivke, parametri funkcij, povratni naslovi... Zmogljivost sklada se izračuna pred izvedbo programa in ostane statična, medtem ko se program izvaja. Običajno ima veliko manjšo zmogljivost od kopice.

### 3.1.2 Kopica

Kopica se uporablja za dinamično dodeljevanje pomnilnika. Tukaj so shranjeni kompleksni podatki, kot so: objekti, podatkovne strukture, veliki podatki... Dodeljevanje in sproščanje pomnilnika lahko ročno opravlja programer, ali pa se to izvaja avtomatsko (npr. smetar). Nima fiksne velikosti. Kopica ima počasnejši dostopni čas kot sklad. To pa zato, ker ima skladovni pomnilnik linearno strukturo, pomnilnik na kopici pa uporablja kazalce.

## 3.2 TEMELJI ALOKACIJE IN DEALOKACIJE POMNILNIKA

Ob zagonu, program prejme nekaj velikih blokov pomnilnika (RAM). Ko program zahteva pomnilnik za objekt ali podatkovno strukturo, se pomnilnik ročno ali samodejno dodeli. Če je to ročno, mora razvijalec to dodelitev izrecno sprogramirati v kodi (npr. malloc). Alokacija lahko uporabi eno od številnih tehnik za dodeljevanje pomnilniških blokov. Ena od metod je prvo prileganje (First-Fit), pri katerem dodeli prvi prosti blok, ki je dovolj velik za zahtevo. Druga metoda je prijateljski sistem (Buddy System), kjer se pomnilnik razdeli na bloke določenih velikosti, ki so potence dvojke (na primer 4 KB, 8 KB ali 16 KB). Če aplikacija potrebuje manjši blok, se večji blok razdeli na manjše "prijateljske" bloke, ki jih je mogoče pozneje ponovno združiti. Še ena takšna metoda je pod-razdeljevalniki (Sub-Allocators), ki aplikaciji dodeli majhne bloke pomnilnika iz večjega bloka, prejetega od sistemskega upravljalnika pomnilnika. Upravljalnik pomnilnika je lahko vgrajen v programski jezik ali na voljo kot knjižnica. Ko program ne potrebuje več pomnilniškega prostora, ki mu je bil dodeljen, se ta prostor sprosti in ponovno dodeli (tj. reciklira). To nalogo lahko ročno opravi programer (npr. free) ali se to samodejno zgodi s procesom, ki se pogosto imenuje smetar (garbage collector - GC).

## 3.3 IZIVI IN NAPAKE

Ena od najpogostejših napak, na katero naletijo razvijalci pri uporabi sklada, je napaka prelivanja sklada (Stack Overflow), ki se pojavi, ko je presežena kapaciteta sklada, na primer zaradi pregloboke rekurzije ali pretirane uporabe lokalnih spremenljivk.

Pri uporabi kopice pa naletimo na drugačne napake. Ena od teh je uhajanje pomnilnika (Memory Leak), kjer dodeljenega pomnilnika ne sprostimo, kar vodi do izčrpanja razpoložljivih virov. Poleg tega se lahko pojavi fragmentacija pomnilnika. To se zgodi ko je skupna količina pomnilnika zadostna, vendar niso na voljo dovolj veliki neprekinjeni bloki.

# 4 C DINAMIČNO DODELJEVANJE POMNILNIKA

V C-ju imamo več funkcij iz standardne knjižnice, ki jih uporabljamo za dinamično dodeljevanje pomnilnika, in sicer malloc, realloc, calloc, aligned_alloc in free. Tukaj se bom predvsem osredotočil na malloc funkcijo. Ta funkcija dodeli oz. rezervira blok pomnilnika, ki je velik toliko bajtov kot je podano v argumentu in vrne kazalec na prvi bajt dodeljenega prostora. Funkcija nam dodeli prostor na kopici, kar upravlja runtime knjižnica, specifična za platformo oz. operacijski sistem. Malloc je v resnici samo posrednik med programom in operacijskim sistemom. V ozadju malloc vključuje več plasti abstrakcije in je odvisen od implementacije v knjižnicah, kot so glibc za Linux ali MSVCRT za Windows.

Ko uporabnik kliče funkcijo malloc ta naprej preveri, ali je na voljo prost blok ustrezne velikosti v že dodeljenem prostoru na kopici. Če je pomnilnik na voljo, ga vrne klicatelju, sicer zahteva več pomnilnika od operacijskega sistema. Linux uporablja sistemske klice za pridobitev pomnilnika (mmapa ali sbrk). Windows uporablja funkcije kot so VirtuallAlloc in HeapAlloc za dodelitev pomnilnika. Ko program sprosti pomnilnik s free, ga knjižnica označi kot prostega in ga lahko ponovno uporabi za prihodnje klice malloc-a.

# 5 GARBAGE COLLECTION

Čeprav se ročno upravljanje pomnilnika zdi preprosto, se lahko zlahka zaplete in je nagnjeno k napakam, ki jih je težko odpraviti. Avtomatsko upravljanje pomnilnika ali AAMM (Automatic application memory managment), je dandanes vse bolj priljubljeno, saj novejši, modernejši in višje nivojski programski jeziki od razvijalcev ne zahtevajo ročnega upravljanja pomnilnika svojih aplikacij. AAMM je največkrat storitev, del ali razširitev programskega jezika. AAMM je po navadi opravljeno s smetarjem (garbage collector). Izvaja se v določenih intervalih, ti intervali pa povzročajo dodatne obremenitve. Ta obremenitev se imenuje čas premora. Smetar reciklira in razdeli bloke, ki niso dosegljivi iz programa. Po definiciji je objekt X dosegljiv, če in samo če:

- register vsebuje kazalec na X ali,
- drug dosegljiv objekt vsebuje kazalec na X.

## 5.1 PREDNOSTI IN SLABOSTI AAMM

Smetar ali GC (garbage collector) programerja osvobodi ročnega sproščanja pomnilnika, kar je prijaznejše programerju ter pomaga preprečiti nekatere vrste napak. Ena takšnih napak so »viseči kazalci«, ki nastanejo, ko se del pomnilnika sprosti, medtem ko nanj še vedno obstajajo kazalci. To lahko privede do nepredvidljivih rezultatov. Prav tako preprečuje napake z dvojno sprostitvijo (double free), ki se pojavijo, ko program poskuša sprostiti že sproščeno in morda ponovno dodeljeno območje pomnilnika. GC tudi zmanjšuje tveganje za »memory leaks«, kjer program ne sprosti pomnilnika, ki ga zasedajo nedosegljivi objekti, kar lahko vodi do izčrpanja pomnilnika.

Ena glavnih slabosti smetarjev pa je, da uporablja računalniške vire, kar lahko poslabša delovanje programa.

## 5.2 RAZLIČNE VRSTE SMETARJEV

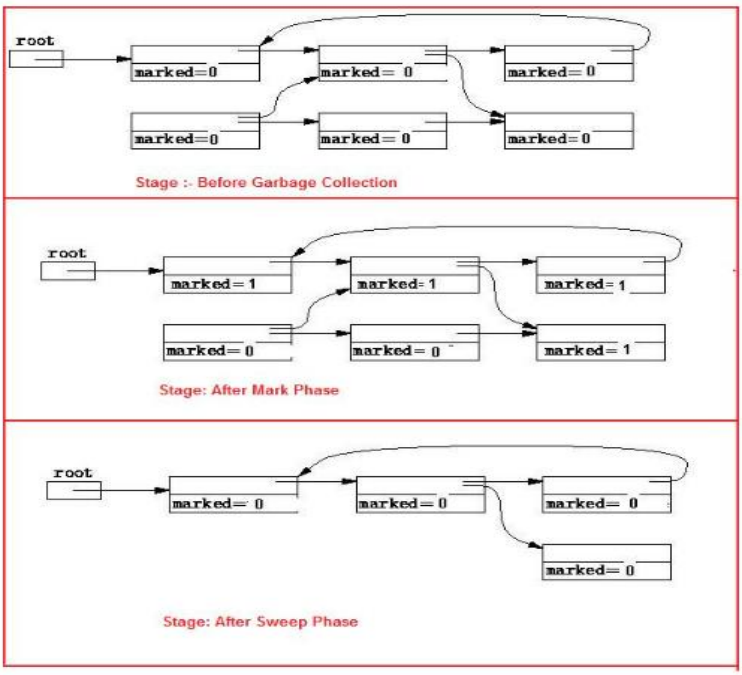
### 5.2.1 Označi in počisti (Mark-and-sweep)

Algoritem označi in počisti je preprost in učinkovit algoritem, ki se uporablja za ponovno pridobitev oz. sproščanje pomnilnika, ki ga aplikacija ne uporablja več. Algoritem deluje tako, da najprej označi vse objekte, ki so še v uporabi. To je po navadi implementirano z enim samim bitom. Ko so vsi živi predmeti označeni, se smetar sprehodi po kopici in povrne vse neoznačene objekte. Algoritem mark-and-sweep je eden od " tracing garbage collectors", ker sistematično sledi in identificira vse objekte, do katerih je mogoče priti neposredno ali posredno iz korenskih referenc programa.

Ko je objekt ustvarjen, je bit označevanja nastavljen na 0 (false). Faza označevanja je odgovorna za identifikacijo vseh živih objektov na kopici. Izvedemo ga tako, da prečkamo kopico in označimo vse objekte, ki so dosegljivi iz korenin kopice. Korenine kopice so objekti, ki so neposredno dostopni aplikaciji, kot so lokalne spremenljivke in reference objektov.

Faza pometanja je odgovorna za sproščanje pomnilnika vseh neoznačenih predmetov v kopici. Izvede se tako, da se sprosti pomnilnik neoznačenih objektov. Celoten postopek lahko vidimo na sliki 2.

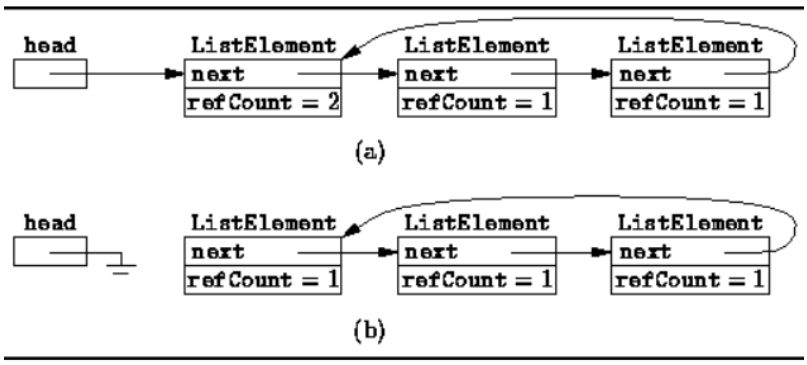
Algoritem učinkovito obravnava ciklične reference in pravilno identificira dosegljive objekte. Glavni pomanjkljivosti sta prekinitve med izvajanjem programa in fragmentacija pomnilnika.




Slika 2: Prikaz algoritma mark-and-sweep

### 5.2.2 Štetje referenc

Ta algoritem je sorazmerno preprost in se ukvarja, s štetjem števila referenčnih kazalcev na vsak dodeljen objekt. Vsak objekt vsebuje število referenc, ki se poveča za vsako novo referenco in zmanjša, če je referenca prepisana ali se referenčni objekt sprosti. Ko števec referenc doseže nulo, se pomnilnik sprosti. Štetje referenc zagotavlja, da so objekti uničeni takoj, ko je uničena njihova zadnja referenca. Ena glavnih težav tega algoritma je, da težko zaznava ciklične reference, kar pomeni, da pride do uhajanja pomnilnika.




Slika 3: Prikaz težave s cikličnimi referencami

### 5.2.3 Analiza pobega (Escape analysis)

Analiza pobega je tehnika, ki med prevajanjem določi, ali je objekt, dodeljen znotraj funkcije, dostopen zunaj nje. Če objekt "uide" in je dostopen drugim funkcijam ali nitim, mora biti dodeljen na kopici. V nasprotnem primeru se lahko dodeli na sklad, kar omogoča njegovo sprostitev ob zaključku funkcije, s čimer se izogne stroškom upravljanja pomnilnika na kopici.

## 5.3 SMETARJI V RAZLIČNIH PROGRAMSKIH JEZIKIH

Različni programski jeziki uporabljajo različne GC algoritme. C# uporablja generacijski GC, ki razdeli pomnilnik v generacije glede na življenjsko dobo objektov. Z ločitvijo objektov v različne generacije lahko GC deluje bolj učinkovito, saj se osredotoči predvsem na pomnilniške bloke, kjer se pričakuje največ smeti. Java ponuja več algoritmov (npr. G1, ZGC…) za različne zahteve. Izberemo ga tako, da pri zagonu aplikacije podamo JVM argumente. Najpogosteje uporabljan GC je G1 (Garbage first), ki razdeli pomnilnik na manjše regije in pri zbiranju smeti prednostno cilja tiste z največ smeti, kar omogoča učinkovito čiščenje z minimalnimi pavzami.. Python uporablja kombinacijo referenčnega štetja in algoritma za detekcijo ciklov. JavaScript uporablje GC, ki je vgrajen v brskalnike in temelji na algoritmu mark-and-sweep.

# 6 IMPLEMENTACIJA ALOKATORJA

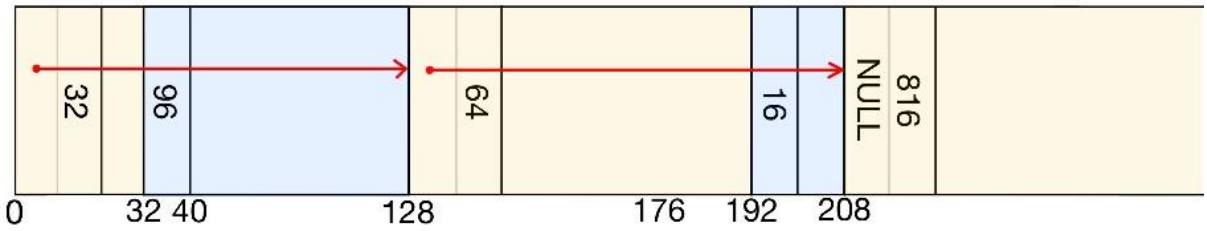
Celotna izvorna koda alokatorja je na voljo na naslednji povezavi: https://github.com/t1k3i/talloc.

## 6.1 OSNOVNA FUNCKIONALNOST IN UPORABA

V tem delu seminarske naloge sem sam implementiral svoj alokator pomnilnika v C-ju, ki ga lahko programerji uporabljajo za svoje potrebe. Poskušal sem narediti nekaj podobnega malloc-u in free-ju iz C-ja. To sem implementiral, kot nekakšno zunanjo knjižnico, ki jo programer samo vključi v svoj program. Izpostavljene sta samo dve funkciji in sicer talloc in tfree. Talloc sprejme samo en argument (velikost zahtevanega prostora) in vrača kazalec na začetek dodeljenega prostora. Tfree ne sprejema argumentov in nič ne vrača, ampak samo sprosti že dodeljeni prostor. Za shranjevanje podatkov sem se odločil uporabiti statično tabelo (uporabljam jo kot kopico), kar pomeni, da podatki niso shranjeni v dinamičnem pomnilniku (kopici), temveč v delu pomnilnika, kjer so globalne spremenljivke. Čeprav bi bilo mogoče pomnilnik dodeliti tudi z uporabo sistemskih klicev, kot sta mmap ali sbrk, sem se odločil za to metodo zaradi preprostosti in kontrole nad pomnilniškimi virom. Pri shranjevanju podatkov sem moral poskrbeti tudi za metapodatke, pri čemer sem moral paziti, da ne uporabljam alokatorjev iz standardne knjižnice C, saj sem razvijal svojo lastno implementacijo alokatorja. Kopice in metapodatkov ni treba posebej inicializirati, saj se to zgodi ob prvem klicu talloc funkcije. Odločil sem se za metodo prvega prileganja (First-Fit), kar pomeni da blok alociram takoj, ko najdem blok, ki ima dovolj prostora.

## 6.2 IMPLEMENTACIJSKE PODROBNOSTI

Pri alokatorju nisem uporabljal zunanjih alokatorjev, zato sem na kopici poleg dejanskih podatkov shranjeval tudi vse pripadajoče metapodatke. Uporabil sem povezan seznam, ki je namenjen sledenju prostim blokom na kopici (tabela bajtov). Na začetku ima seznam samo eno vozlišče, ki predstavlja celotno kopico, kasneje pa se dodajajo nova vozlišča, ko se kopica fragmentira. Vsako vozlišče se nahaja na naslovu v tabeli, kjer se prosti blok začne. Prvih nekaj bajtov je namenjenih shranjevanju velikosti prostega bloka in kazalca na naslednji prosti blok. Notranja struktura kopice je prikazana na sliki 4. Vsi bloki so poravnani na velikost, ki ustreza strukturi s kazalcem in spremenljivko tipa size_t (običajno 16 bajtov), njihova minimalna velikost pa je prav tako 16 bajtov. Tako zagotovimo pravilno poravnavo in preprečimo morebitno prepisovanje podatkov. Metapodatki se seveda prepišejo, ko se blok dodeli v uporabo. Vsak dodeljen oz. uporabljan blok pa ima v prvih nekaj bajtih zapisano tudi svojo velikost, ki je potrebna pri funkciji tfree, da ta ve, kaj dodati v povezan seznam prostih blokov. Pri funkciji tfree sem naletel na težavo, kjer so se po sprostitvi pomnilnika zaporedni bloki obravnavali kot ločeni prosti bloki. Težavo sem rešil tako, da po vsakem sproščanju preverim, ali so sosednji bloki v seznamu tudi zaporedni bloki v »kopici«, in jih po potrebi združim.




Slika 4: Primer tabele (kopice), s 1024 bajti. Modra barva prikazuje zasedene dele, rumena pa proste.

# 7 ZAKLJUČEK

Pri pisanju seminarske naloge sem spoznal, kako kompleksno je upravljanje pomnilnika in kaj vse sodi k temu področju. Na programskem nivoju, še posebej pri jeziku C, je upravljanje pomnilnika zahtevno, saj vsa odgovornost pade na programerja, da pravilno dodeli in sprosti pomnilnik, kar lahko vodi v različne napake, kot so puščanje pomnilnika in neveljavni kazalci. Bolje sem spoznal tudi razlike med kopico in skladom.

Raziskava GC-jev mi je pokazala, zakaj je v jezikih kot npr. Java, delo s pomnilnikom veliko lažje. Spoznal sem različne algoritme kot so »Mark and sweep«, štetje referenc in »Escape analysis«, ter ugotovil, razlike v GC-ju pri različnih programskih jezikih.

Z implementacijo svojega alokatorja v C-ju pa sem še praktično preizkusil, kako deluje dinamična dodelitev pomnilnika. Ta izkušnja mi je dala globlji vpogled v izzive in optimizacije, ki jih uporabljajo sodobni alokatorji in GC-ji.

Mislim, da je za vsakega programerja pomembno, da razume vsaj osnove iz tega področja, saj bo tako bolj učinkovito programiral in si bo s tem privarčeval kar nekaj časa, predvsem pri »razhroščevanju« svojih programov.

## VIRI

Wikipedia. Tracing garbage collection. https://en.wikipedia.org/wiki/Tracing_garbage_collection (dostopno januar 2025)

Wikipedia. Garbage collection. https://en.wikipedia.org/wiki/Garbage_collection_(computer_science) (dostopno januar 2025)

Medium. Stack, Heap Memory. https://medium.com/@dhleee0123/stack-heap-memory-86daeb1b48f7 (dostopno januar 2025)

TechTarget. What is memory managment in a computer Enviroment? https://www.techtarget.com/whatis/definition/memory-management (dostopno januar 2025)

Medium. The Basics of Application Memory Management. https://medium.com/dvt-engineering/the-basics-of-application-memory-management-19f060c2d0f (dostopno januar 2025)

Wikipedia. C dynamic memory allocation. https://en.wikipedia.org/wiki/C_dynamic_memory_allocation (dostopno januar 2025)

Java IQ. Mark-And-Sweep (Garbage Collection Algorithm). https://sandeepin.wordpress.com/2011/12/11/mark-and-sweep-garbage-collection-algorithm/ (dostopno januar 2025)

Why reference counting does not work. https://book.huihoo.com/data-structures-and-algorithms-with-object-oriented-design-patterns-in-java/html/page423.html (dostopno januar 2025)