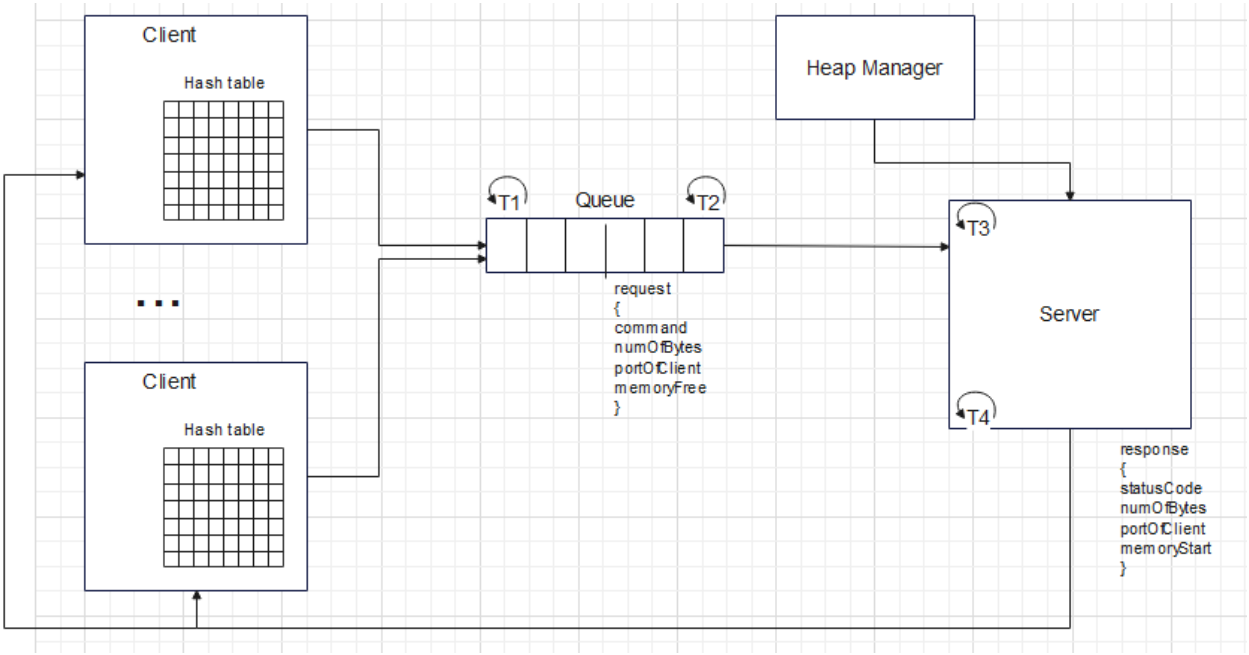
Memory Management (MM3)

Uvod:

Cilj ovog projekta je simulacija rada Heap Manager-a, tj. zauzimanje i oslobađanje memorije na zahtev korisnika aplikacije. Komunikacija između klijenta i servera se odvija tako što klijent zatraži da zauzme/obriše određeni deo memorije, a server mu pomoću svojih internih mehanizama to i omogućuje.

Dizajn:



Aplikacija se sastoji od 4 komponente: *Client*, *Server*, *Queue* I *HeapManager.*

Client: Konzolna aplikacija koja šalje podatke kroz strukturu *request* ka komponenti Queue. Ta aplikacija korisniku pruža mogućnost da izabere šta želi da izvrši kroz ponuđeni meni. Izborom neke od opcija iz menija šalje se zahtev ka serveru kroz komponentu queue. Klijent kao odgovor od servera prima podatke upakovane u strukturu *response* i tu posmatramo podatke:

- relativnu adresu koju smesta u strukturu HashTable (u slučaju da je zatražio zauzimanje memorije)
- status o uspešnosti oslobađanja memorije
- obaveštenje da je uspešno iniciran ispis statistike memorije.

Server: Konzolna aplikacija koja prima podatke iz komponente queue I obrađuje ih pomoću funkcija iz komponente HeapManager. Ova komponenta sadrži 2 niti, tj. Razdvojeno je slanje odgovora ka klijentskoj utičnici I primanje zahteva sa queue komponente. Ovime postižemo da se 2 operacije vrše konkurentno.

T3: Obrađuje zahteve iz queue i manipuliše memorijom.

T4: Šalje odgovor ka serveru koji zavisi od uspešnosti manipulacije memorijom.

Queue: Konzolna aplikacija koja služi kao veza klijenta i servera. Unutar nje se nalazi implementirana struktura queue unutar koje smeštamo sve zahteve koji dođu od klijenata. Ti zahtevi se vade jedan po jedan iz strukture i šalju ka serveru. Kao i na serveru, i ovde imamo 2 niti kojima razdvajamo slanje ka serveru i primanje zahteva od klijenata.

T1: Prima zahteve sa klijentske utičnice I smešta pristigle podatke u queue strukturu

T2: Vadi podatke iz queue strukture I šalje ih ka serverskoj utičnici.

HeapManager: Biblioteka čije funkcije se pozivaju unutar serverske komponente (biblioteka implementirana u serverskom projektu). Unutar ove biblioteke se nalazi sva logika za manipulaciju sa memorijom. Pošto radimo sa deljenom memorijom koja može da se menja iz više niti, obezbeđena je mehanizmima *mutex* I *semaphore*.

Strukture podataka:

U projektu se korišćene strukture podataka: *Queue* i *HashTable*

Queue: FIFO struktura koja skladišti pristigle zahteve, tipa request, sa klijentske strane, čuva ih i kada se stvori prilika, šalje ih ka serveru na obradu. Queue se koristi jer želimo da šaljemo zahteve redom i u ovoj strukturu ih čuvamo samo kako bismo izbegli zagušenje mreže u slučaju velikog broja zahteva.

HashTable: Struktura koja čuva parove “brojač-relativna adresa” (broj pozicija od početka zauzete memorije) koje je korisnik, putem zahteva, zauzeo. Izborom neke od opcija iz strukture, ta opcija se briše jer smatramo da smo oslobodili taj deo memorije. HashTable se koristi zbog brzine pretrage izabranog elementa, tj. Kako ne bismo morali da prođemo kroz celu strukturu da bismo našli traženi element.

Opis najbitnijih funkcija:

Funkcije na klijentu:

request* userMenu(HashTable* ht) - Prikazuje na konzoli opcije koje korisnik može da izabere I u zavisnosti od izabrane opcije popunjava strukturu odgovarajućim podacima. Povratna vrednost funkcije je pokazivač na popunjenu strukturu. Kao parametre funkcija prima pokazivač na strukuturu HashTable i to nam služi kako bismo ispisali sadržaj hash tabele.

Funkcije u heap manager-u:

void init_memory(char* memory, int start) - Kreirana memorija se inicijalizuje sa "_" karakterom I to nam služi zbog vizuelnog prikaza na samoj konzoli kako bismo pratili zauzetost memorije (sa "_" će biti označena slobodna memorija a sa "x" ili "0" zauzeta memorija).

char* create_memory() - Alocira se 100 bajta memorije koju će korisnici moći da koriste i ta memorije se po potrebi može proširiti ako korisnici zauzmu sve (expand_memory funkcija)

int alocate_memory(char* memory, int bytes) - Na osnovu prosleđenog broja bajtova se u memoriji traži mesto za simulaciju alokacije. To mesto zavisi od količine bajtova I zauzetosti trenutnog bloka. Ako je početak bloka već zauzet, a količina bajtova koje je korisnik tražio veća od preostalog prostora u bloku, tada se prelazi u novi blok I nastaje "rupa u memoriji" (fragment).

void free_memory(char* memory, int start) - Prosleđena memorija kroz parametre se bajt po bajt iterativno *override-uje, tj.* menja se iz zauzete u slobodnu.

char* expand_memory(char* memory) - U slučaju da je sva do sada alocirana memorija za korisnike zauzeta, pristupa se ovoj funkciji koja vrši realokaciju memorije(stara memorija se kopira na novo mesto I uvećava tj. proširuje) ali ovaj put uvećana za 20 bajtova.

Zaključak:

Rešenje je realizovano jednostavnom klijent-server komunikacijom pomoću UDP transportnog protokola. Prilagođeno je istovremenom radu sa većim brojem klijenata koji zauzimaju/oslobađaju jednu zajedničku memoriju. Svaki klijent vidi isključivo svoje zauzete memorije, a statistika o izmenama u memoriji se ispisuje na osnovu zahteva svih korisnika.

Potencijalna unapređenja:

Hashtable bi se mogla unaprediti mehanizmom za rešavanje kolizije, tj. da svaki put kada se heširana vrednost poklopi da te elemente ulančavamo pomoću jednostruko spregnute liste. Time ne gubimo ni jedan element iz strukture, a vreme pretrage se blago usložnjava jer ako nam treba element iz tabele koji je spregnut, moramo proći kroz svaki element iz spregnute liste kako bismo našli onaj traženi.