Tema de casă 2 - Alocator de memorie

În această temă veţi implementa un alocator simplu de memorie, similar sistemului malloc/free.

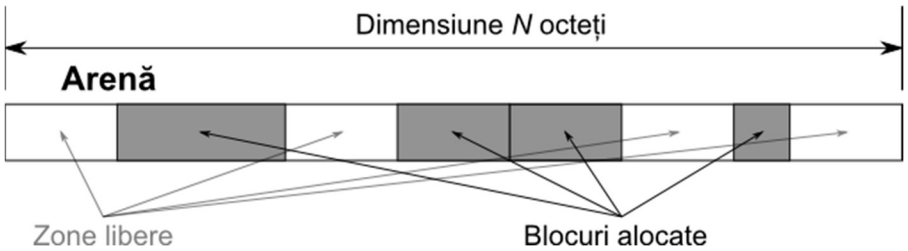
Cerința temei

Programul vostru va trebui să realizeze o simulare a unui sistem de alocare de memorie. Programul va primi la intrare comenzi de alocare, alterare, afişare şi eliberare de memorie, şi va trebui să furnizaţi la ieşire rezultatele fiecărei comenzi. Nu veţi înlocui sistemul standard `malloc()` şi `free()`, ci vă veţi baza pe el, alocând la început un bloc mare de memorie, şi apoi presupunând că acela reprezintă toată "memoria" voastră, pe care trebuie s-o gestionaţi.

Funcţiile unui alocator de memorie

Un alocator de memorie poate fi descris, în termenii cei mai simpli, în felul următor:

- Primeşte un bloc mare, compact (fără "găuri"), de memorie, pe care trebuie să-l administreze. Acest bloc, în terminologia de specialitate, se numeşte `arenă`. De exemplu, sistemul de alocare cu `malloc()` are în gestiune `heap`-ul programului vostru, care este un segment special de memorie special rezervat pentru alocările dinamice.
- Utilizatorii cer din acest bloc, porţiuni mai mici, de dimensiuni specificate. Alocatorul trebuie să găsească în arenă o porţiune continuă liberă (nealocată), de dimensiune mai mare sau egală cu cea cerută de utilizator, pe care apoi o marchează ca ocupată şi întoarce utilizatorului adresa de început a zonei proaspăt marcată drept alocată. Alocatorul trebuie să aibă grijă ca blocurile alocate să nu se suprapună (să fie `disjuncte`), pentru că altfel datele modificate într-un bloc vor altera şi datele din celălalt bloc.
- Utilizatorii pot apoi să ceară alocatorului să elibereze o porţiune de memorie alocată în prealabil, pentru ca noul spaţiu liber să fie disponibil altor alocări.
- La orice moment de timp, arena arată ca o succesiune de blocuri libere sau ocupate, ca în figura de mai jos.



O problemă pe care o are orice alocator de memorie este `cum este ţinută evidenţa blocurilor alocate, a porţiunilor libere şi a dimensiunilor acestora`. Pentru această problemă există în general două soluţii:

- Definirea unor zone de memorie separate de arenă care să conţină liste de blocuri şi descrierea acestora. Astfel, arena va conţine doar datele utilizatorilor, iar secţiunea separată va fi folosită de alocator pentru a găsi blocuri libere şi a ţine evidenţa blocurilor alocate.
- Cealaltă soluţie, `pe care voi o veţi implementa în această temă`, foloseşte arena pentru a stoca informaţii despre blocurile alocate. Preţul plătit este faptul

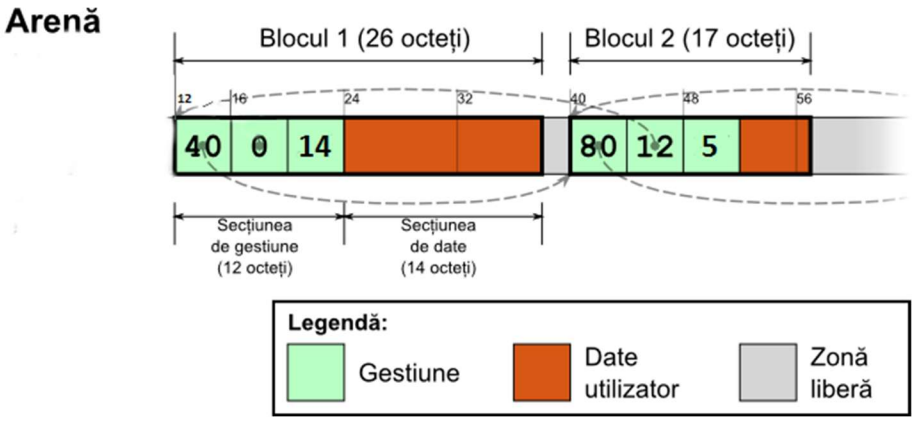
că arena nu va fi disponibilă în totalitate utilizatorilor, pentru că va conţine, pe lângă date, şi informaţiile de gestiune, însă avantajul este că nu are nevoie de memorie suplimentară şi este în general mai rapidă decât prima variantă.

Există mai multe metode prin care se poate ţine evidenţa blocurilor alocate în arenă, în funcţie de performanţele dorite. Voi va trebui să implementaţi un mecanim destul de simplu, care va fi prezentat în secţiunea următoare. Deşi nu este extrem de performant, se va descurca destul de bine pe dimensiuni moderate ale arenei (de ordinul MB).

Structura arenei

În continuare vom considera arena ca pe o succesiune (vector) de `N` octeţi (tipul de date `unsigned char`). Fiecare octet poate fi accesat prin indexul său (de la `0` la `N-1`). Vom considera că un index este un întreg fara semn pe 32 de biţi (tipul de date `uint32_t` din libraria `stdint.h`). De asemenea, va fi nevoie câteodată să considerăm 4 octeţi succesivi din arenă ca reprezentând valoarea unui index. În această situaţie, vom considera că acel index este reprezentat în format `'little-endian'` (revedeţi exercitiile de la laboratorul de pointeri pentru mai multe detalii şi citiţi acest articol mult mai descriptiv), şi astfel vom putea face cast de la un pointer de tip `unsigned char *` la unul de tip `uint32_t*`, pentru a accesa valoarea indexului, stocată în arenă.

Figura de mai jos ilustrează structura detaliată a arenei, în decursul execuţiei programului:



Structura unui bloc

Se poate observa că fiecare bloc alocat de memorie (marcat cu un chenar îngroşat) constă din două secţiuni:

- Prima secţiune, `de gestiune`, este reprezentată de 12 octeţi (`3 * sizeof(uint32_t)`) împărţiţi în 3 întregi (a câte 4 octeţi fiecare). Cei trei întregi reprezintă următoarele:
  - Primul întreg reprezintă indexul de start al blocului următor de memorie din arenă (aflat imediat "la dreapta" blocului curent, dacă privim arena ca pe o succesiune de octeţi de la stanga la dreapta). Se consideră că un bloc începe cu secţiunea de gestiune, şi toţi indicii la blocuri vor fi trataţi ca atare. Dacă blocul este ultimul în arenă (cel mai "din dreapta"), atunci valoarea primului întreg din secţiune va fi `0`.

    - Cel de-al doilea întreg din secţiune reprezintă indexul de start al blocului imediat anterior din arenă. Dacă blocul este primul în arenă, atunci valoarea acestui întreg va fi `0`.
    - Cel de-al treilea întreg din secţiune reprezintă lungimea sectiunii de date (a datelor alocate utilizatorului).
  - A doua secţiune conţine `datele efective` ale utilizatorului. Secţiunea are lungimea în octeţi egală cu dimensiunea datelor cerută de utilizator la apelul funcţiei de alocare. Indicele returnat de alocator la o nouă alocare reprezintă începutul acestei secţiuni din noul bloc, şi `'nu'` începutul primei secţiuni, întrucât partea de gestiune a memoriei trebuie să fie complet transparentă pentru utilizator.

Daca folositi uint32_t pentru memorarea indecsilor, aveti grija la UNDERFLOW! In acest sens, nu lucrati cu valori negative in acesti indecsi.
Daca totusi aveti nevoie sa folositi valori negative pentru indecsi in logica implementarii voastre, puteti folosi `int32_t`, sau `int` daca pe arhitectura voastra `sizeof(int)` este 4. Si aceste tipuri de date se incadreaza in restrictiile temei (testele vor fi de asa natura incat indecsii vostrii ar trebui sa se incadreze in `[0 .. 2 ^ 31 - 1]`).

Înlănţuirea blocurilor

`Indicele de start` reprezinta indicele primului bloc (cel mai "din stânga") din arenă. Acesta trebuie stocat de voi intr-o variabila separata. Dacă arena nu conţine niciun bloc (de exemplu, imediat după iniţializare), acest indice este `0`.
`Indicele de start` marchează începutul `lanţului de blocuri` din arenă: din acest indice putem ajunge la începutul primului bloc, apoi folosind secţiunea de gestiune a primului bloc putem găsi începutul celui de-al doilea bloc, şi asa mai departe, până când ajungem la blocul care are indexul blocului următor 0 (este ultimul bloc din arenă). În acest mod putem traversa toate blocurile din arenă, şi de asemenea să identificăm spaţiile libere din arenă, care reprezintă spaţiile dintre două blocuri succesive.
Primul bloc din arena trebuie sa aiba octetii indexului blocului anterior setati pe 0. Ultimul bloc din arena trebuie sa aiba octetii indexului blocului urmator setati pe 0.
Daca arena contine un singur bloc, atunci atat indexul blocului urmator cat si indexul blocului anterior vor fi 0.
Este de remarcat faptul că lanţul poate fi parcurs în ambele sensuri: dintr-un bloc putem ajunge atât la vecinul din dreapta, cât şi la cel din stânga.
De asemenea, atunci când este alocat un bloc nou sau este eliberat unul vechi, `'lanţul de blocuri trebuie modificat'`. Astfel, la alocarea unui nou bloc de memorie, trebuie să ţineţi cont de următoarele:

- Spaţiul liber în care este alocat noul bloc este mărginit de cel mult două blocuri vecine. Secţiunile de gestiune ale acestor vecini trebuie modificate astfel:
  - Indexul blocului următor din structura de gestiune a blocului din stânga trebuie să indice către noul bloc. Dacă blocul din stânga nu există, atunci este modificat indicele de start.
  - Indexul blocului precedent din structura de gestiune a blocului din dreapta trebuie să indice către noul bloc. Dacă blocul din dreapta nu există, atunci nu se întâmplă nimic.
- Secţiunea de gestiune a noului bloc va conţine indicii celor doi vecini, sau `0` în locul vecinului care lipseşte.

La eliberarea unui bloc, trebuie modificate secţiunile de gestiune a vecinilor într-o manieră similară ca la adăugare.

Funcţionarea programului

Programul vostru va trebui să implementeze o serie de operaţii de lucru cu arena, care vor fi lansate în urma comenzilor pe care le primeşte la intrare. Fiecare comandă va fi dată pe câte o linie, şi rezultatele vor trebui afişate pe loc. Secţiunea următoare prezintă sintaxa comenzilor posibile şi formatul de afişare al rezultatelor.

Întrucât pentru testare comenzile vor fi furnizate prin redirectare dintr-un fişier de intrare, iar rezultatele vor fi stocate prin redirectare într-un alt fişier, programul vostru nu va trebui să afişeze nimic altceva în afara formatului specificat (de exemplu, nu trebuie să afişati mesaje de tipul `"Introduceţi comanda: "`).

Folosiţi funcţiile de manipulare a şirurilor de caractere pentru a citi şi interpreta comenzile date la intrare. Este recomandată combinaţia getline() şi strtok() pentru o implementare elegantă.

Pentru o mai bună organizare a codului vostru, implementaţi execuţia fiecărei comenzi într-o funcţie separată. De asemenea, gândiţi-vă ce variabile trebuie păstrate globale, iar pe restul declaraţi-le local. Puteti folosi variabile globale in aceasta tema.

Formatul comenzilor

Fiecare comanda trebuie afisata pe cate o linie separata, exact cum este citita de la input. Programul vostru va trebui să accepte următoarele comenzi la intrare:

1. **`INITIALIZE <N>`**
   - Această comandă va trebui să realizeze iniţializarea unei arene de dimensiune `N` octeţi. Prin iniţializare se înţelege alocarea dinamică a memoriei necesare stocării arenei, setarea fiecărui octet pe 0, şi iniţializarea lanţului de blocuri (setarea indicelui de start pe `0`).
   - Comanda nu va afişa niciun rezultat.
2. **`FINALIZE`**
   - Această comandă va trebui să elibereze memoria alocată la iniţializare.
   - Comanda nu va afişa niciun rezultat.
3. **`DUMP`**
   - Această comandă va afişa întreaga hartă a memoriei, aşa cum se găseşte în acel moment, octet cu octet. Vor fi afişaţi câte 16 octeţi pe fiecare linie, în felul următor:
     - La începutul liniei va fi afişat indicele curent, în format hexazecimal, cu 8 cifre hexa majuscule.
     - Apoi este afişat un TAB (`\t`) , urmat de 16 octeţi, afişati separaţi printr-un spaţiu şi în format hexazecimal, cu 2 cifre hexa majuscule fiecare. Între cel de-al 8-lea şi cel de-al 9-lea octet se va afişa un spaţiu suplimentar.
     - Daca dimensiunea arenei nu este multiplu de 16, atunci pe ultima linie se vor afisa ultimii ARENA_SIZE % 16 octeti.
     - Nu este necesar sa realizati conversii de la zecimal la hexazecimal, puteti folosi `printf("%02X")` si `printf("%08X")` pentru afisare.
4. **`ALLOC <SIZE>`**
   - Comanda va aloca `SIZE` octeţi de memorie din arenă, unde `SIZE` e o valoare strict pozitiva. Ea va trebui să găsească o zonă liberă suficient de mare (care să

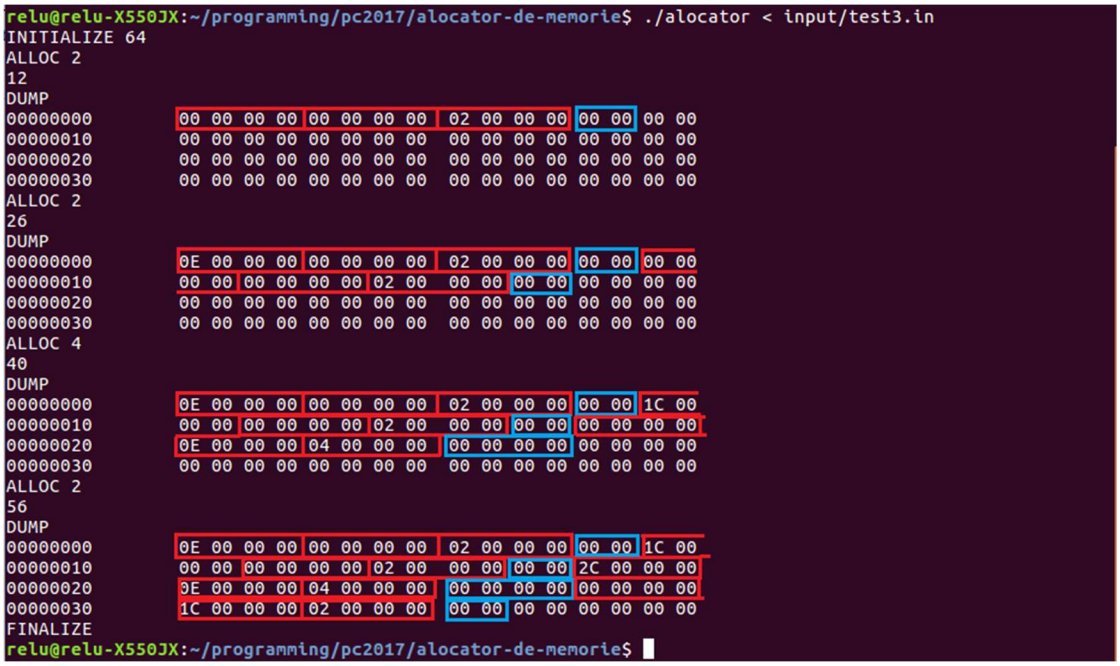
încapă `SIZE` octeţi + secţiunea de gestiune), şi să rezerve un bloc `'la început'` zonei (nu în mijloc, nu la sfârşit). Va trebui folosită prima zonă liberă validă, într-o căutare de la stânga la dreapta.
   - Comanda va afişa, în format zecimal, indexul de început al blocului alocat în arenă, sau `0` dacă nu a fost găsită nici o zonă liberă suficient de mare în arenă. Atenţie: Va trebui să afişaţi indexul secţiunii de date din noul bloc, şi nu al secţiunii de gestiune.
5. **`FREE <INDEX>`**
   - Comanda va elibera blocul de memorie al cărei secţiuni de date începe la poziţia `INDEX` în arenă. Practic, `INDEX`va fi o valoare care a fost întoarsă în prealabil de o comandă 'ALLOC'. În urma execuţiei acestei comenzi, octetii vechiului bloc (octeti gestiune + octeti date) vor fi setati pe 0, iar spaţiul de arenă ocupat va redeveni disponibil pentru alocări ulterioare.
   - Comanda nu va afişa niciun rezultat.
6. **`FILL <INDEX> <SIZE> <VALUE>`**
   - Comanda va seta `SIZE` octeţi din arenă la valoarea `VALUE`, cuprinsă între 0 şi 255 inclusiv, si va modifica octetii blocurilor începând cu blocul cu indexul `INDEX`. Atenție, această comandă NU modifica octeți de gestiune, ci doar octeți de date. Se vor seta octeti de date pana cand s-au setat `SIZE` octeti sau pana cand s-au parcurs toate blocurile de dupa blocul `INDEX` si s-au setat toti octetii de date ale acestora. Cu alte cuvinte,
   - `NR_OCTETI_SETATI = min(SIZE, SIZE(block(INDEX)) + SIZE(next(block(INDEX))) + SIZE(next(next(block(INDEX)))) + … )`.
   - Comanda nu va afişa niciun rezultat.

Va este garantat ca atat comanda FREE cat si comanda FILL vor primi cate un index VALID (adica un index returnat inainte de un apel ALLOC).

Nu este nevoie să vă preocupaţi de eventualele comenzi invalide. Veţi presupune că toate comenzile introduse vor fi corecte. Nu trebuie să verificaţi semantica operaţiilor cerute programului vostru. Executati comenzile EXACT cum le primiti la input.
Programul vostru va trebui sa functioneze corect daca dupa FINALIZE se apeleaza din nou INITIALIZE urmat de FINALIZE.
Primul bloc din arena nu incepe intotdeauna de la indexul 0! Pastrati indexul primului block din arena intr-o variabila si aveti grija sa verificati daca trebuie sa o modificati atunci cand faceti ALLOC sau FREE.

Exemple

Exemplul 1



Observaţii:

- Se initializeaza o arena cu 64 de octeti.
- Apelul ALLOC 2 intoarce indexul 12 deoarece primul bloc este alocat incepand cu indexul 0 din memorie dar primii 12 octeti sunt de gestiune.
- În primul output de `DUMP`, pe prima linie, octetii 0-3 sunt 0 (reprezinta indexul blocului urmator), octetii 4-7 sunt 0 (reprezinta indexul blocului precedent), iar octetii 8 - 11 sunt `'02 00 00 00'` (reprezinta valoarea intreaga 2 pe 4 octeti in format little endian).
- La al doilea apel ALLOC 2, indexul intors este 26. Pana acum arena contine:
  1. 0-11 octetii de gestiune bloc 1
  2. 12-13 octetii de date bloc 1
  3. 14-25 octetii de gestiune bloc 2
  4. 26-27 octetii de date bloc 2
- La apelul ALLOC 4, indexul intors este 40. Pana acum arena contine:
  1. 0-11 octetii de gestiune bloc 1
  2. 12-13 octetii de date bloc 1
  3. 14-25 octetii de gestiune bloc 2
  4. 26-27 octetii de date bloc 2
  5. 28-39 octetii de gestiune bloc 3
  6. 40-43 octetii de date bloc 3
- La ultimul apel ALLOC 2, indexul intors este 56. Pana acum arena contine:
  1. 0-11 octetii de gestiune bloc 1
  2. 12-13 octetii de date bloc 1
  3. 14-25 octetii de gestiune bloc 2
  4. 26-27 octetii de date bloc 2
  5. 28-39 octetii de gestiune bloc 3
  6. 40-43 octetii de date bloc 3
  7. 44-55 octetii de gestiune bloc 4
  8. 56-57 octetii de date bloc 4
- Observati si modificarea octetilor de gestiune pe parcursul adaugarii blocurilor. De exemplu, in ultimul `'DUMP'`, octetii 0-3 reprezinta indexul zonei de gestiune a

blocului 2, `0E 00 00 00` reprezinta 14. Octetii 44-47 sunt 0 pentru ca blocul 4 este ultimul in arena, iar octetii 48-51 reprezinta indexul zonei de gestiune pentru blocul 3: `1C 00 00 00` adica 28.

Exemplul 2

Observaţii:

- Pentru toate cele 4 alocari de la inceput, indecsii zonelor de gestiune ale celor 4 blocuri incep de la adresele '`0x0`', '`0x10`', '`0x20`' si respectiv '`0x40`'.
- Dupa apelul '`FREE 12`' toti cei 16 octeti rezervati pentru blocul 1 sunt setati pe 0. Din zona de gestiune pentru blocul 2 ar fi trebuit sa se modifice octetii pentru indexul anterior, insa in acest caz ei raman 0 pentru ca blocul 2 devine primul bloc din arena si trebuie sa aiba octetii indexului anterior setati pe 0.
- Dupa apelul '`ALLOC 4`' de dupa '`FREE 12`', se aloca un bloc de dimensiune 4 la indexul 12, iar memoria ajunge in aceeasi stare ca la primul '`DUMP`'.
- Comanda '`FILL 12 100 255`' scrie '`0xFF`' peste toti octetii de date din blocuri (incepand cu primul). Observati ca dimensiunea data ca parametru 100 este mai mare ca suma dimensiunilor tuturor blocurilor in prealabil alocate pana in acest moment (4 + 4 + 20 + 4 = 32). Prin urmare, doar 32 de octeti vor fi scrisi. Insa, in figura observam doar 28 de octeti '`0xFF`' si 4 octeti '`0x7F`'. Acest lucru este datorat celei de-a doua comenzi '`FILL 76 4 127`' care suprascrie octetii 76-79, initial setati cu valoarea '`0xFF`'.

Restrictii

`0 < ARENA_SIZE ⇐ 2 ^ 16`