Tema de casă 2 - Alocator de memorie

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

Responsabil: Mădălina Hristache

Autor inițial: Ștefan Bucur [mailto:stefan.bucur@gmail.com]

Deadline: 11.12.2014

Menționăm că pentru testare (pe vmchecker) se folosește o mașină virtuală pe 32 de biți. Arhiva de test folosește un astfel de binar. În caz că sistemul vostru de operare de pe mașina fizică este pe 64 de biți, sugerăm să faceți testarea finală și pe o mașină (virtuală sau nu) de 32 de biți.

Obiective

În urma realizării acestei teme, studentul va fi capabil:

- să lucreze cu pointerii pentru a manipula date stocate într-o memorie;
- să lucreze cu mecanismul de alocare de memorie din biblioteca standard C;
- să înţeleagă şi să implementeze conceptele din spatele unui alocator de memorie;
- să lucreze cu funcţii de manipulare a şirurilor de caractere;
- să înţeleagă şi să genereze hărţi de memorie.

Enunţul temei

În spiritul Crăciunului, Mihai s-a oferit să o ajute pe sora sa, Ilinca, să înțeleagă cum funcționează alocarea de memorie dinamică. Primul impuls a fost să îi arate codul sursă [http://sources.redhat.com/cgi-bin/cvsweb.cgi/libc/malloc/?cvsroot=glibc#dirlist] al funcțiilor de alocare din `glibc`. Când a văzut însă cum arată codul pentru malloc() [http://sources.redhat.com/cgi-bin/cvsweb.cgi/~checkout~/libc/malloc/malloc.c?rev=1.127.2.44&content-type=text/plain&cvsroot=glibc], s-a lăsat păgubaş considerându-l prea greu de urmărit. Astfel, Mihai s-a decis să proiecteze singur un sistem de alocare de memorie, simplu şi pe înţelesul tuturor.

Cerința temei

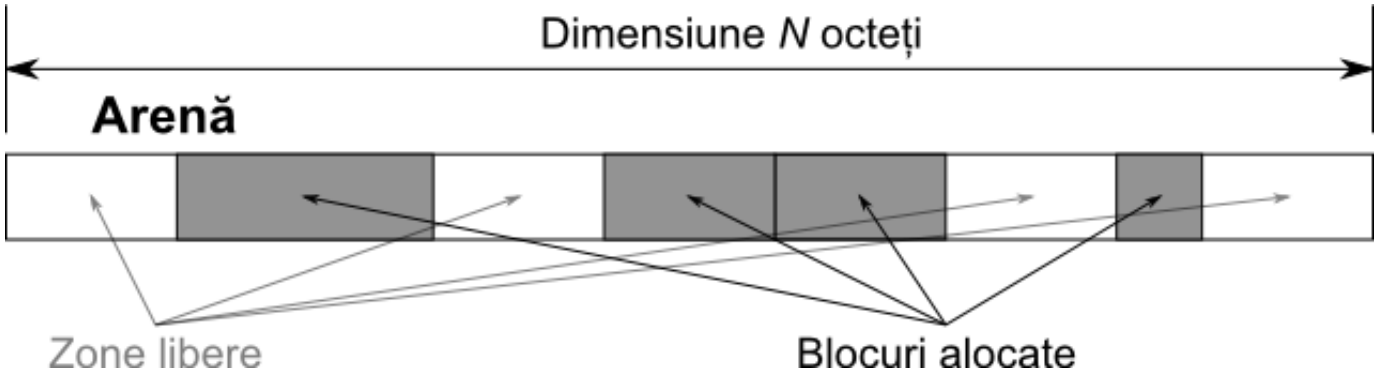
Programul vostru va trebui să realizeze o simulare a unui sistem de alocare de memorie. Programul va primi la intrare comenzi de alocare, alterare, afişare şi eliberare de memorie, şi va trebui să furnizaţi la ieşire rezultatele fiecărei comenzi. Nu veţi înlocui sistemul standard `malloc()` şi `free()`, 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:

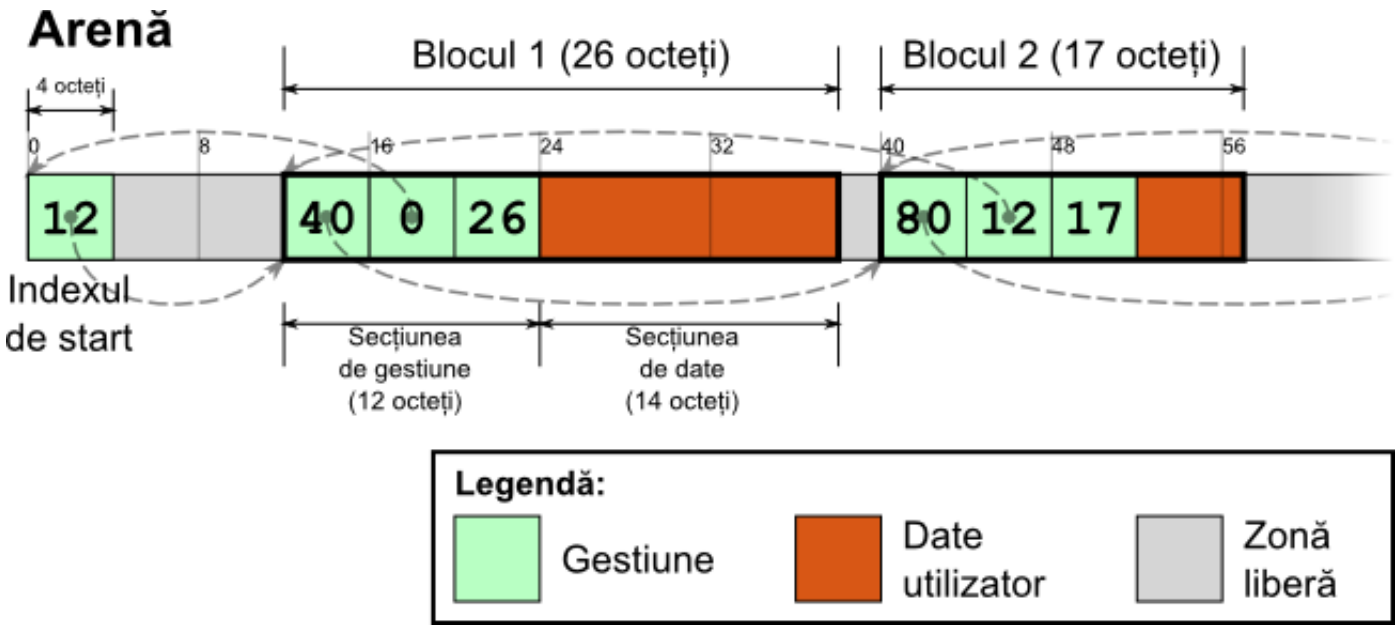
- Definirea unor zone de memorie separate de arenă care să conţină liste de blocuri şi descrierea acestora. Astfel, arena va conţine doar datele utilizatorilor, iar secţiunea separată va fi folosită de alocator pentru a găsi blocuri libere şi a ţine evidenţa blocurilor alocate.
- Cealaltă soluţie, `pe 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 cu semn pe 32 de biţi (tipul de date `int` pe un sistem de operare pe 32 de biţi). 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 exerciţiile de la laboratorul de pointeri pentru mai multe detalii şi citiţi acest articol [http://en.wikipedia.org/wiki/Endianess] mult mai descriptiv), şi astfel vom putea face cast de la un pointer de tip `unsigned char *` la unul de tip `int*`, 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(int)`) 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 totală a blocului, adică lungimea celor două secţiuni la un loc (nu doar 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.

Înlănţuirea blocurilor

După cum se poate observa din figura de mai sus, la începutul arenei sunt rezervaţi 4 octeţi care reprezintă `indicele de start` - indicele primului bloc (cel mai "din stânga") din arenă. Dacă arena nu conţine nici un 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.

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 unui 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 fgets() 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.

Formatul comenzilor

Programul vostru va trebui să accepte următoarele comenzi la intrare:

1. **`INITIALIZE <N>`**
   - Această comandă va fi apelată prima, şi 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 nici un rezultat.
2. **`FINALIZE`**
   - Această comandă este apelată ultima, şi va trebui să elibereze memoria alocată la iniţializare şi să încheie programul.
   - Comanda nu va afişa nici un 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.
- Pe ultima linie, indiferent de numărul de octeţi din arenă, se va afişa indexul ultimului octet din arenă + 1 (practic, dimensiunea arenei), în format hexazecimal cu 8 cifre hexa majuscule.

4. **ALLOC <SIZE>**
   - Comanda va aloca `SIZE` octeţi de memorie din arenă. 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 începutul'` 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, spaţiul de arenă ocupat de vechiul bloc va redeveni disponibil pentru alocări ulterioare.
   - Comanda nu va afişa nici un rezultat.
6. **FILL <INDEX> <SIZE> <VALUE>**
   - Comanda va seta `SIZE` octeţi consecutivi din arenă, începând cu indexul `INDEX`, la valoarea `VALUE`, cuprinsă între 0 şi 255 inclusiv. Atenție, această comandă poate modifica și octeți de gestiune, nu numai octeți de date. În acest caz, se garantează ca arena nu va deveni coruptă după o serie de comenzi `FILL` consecutive.
   - Comanda nu va afişa nici un rezultat.
7. **SHOW <INFO>**
   - Comanda va afişa informaţii statistice despre starea arenei. `INFO` poate fi una din următoarele:
     a. **FREE**
        - Vor fi afişaţi (în format zecimal) numărul de octeţi liberi din arenă, împreună cu numărul de regiuni (zone continue) libere din arenă sub forma următoare:

```
<nblocks> blocks (<nbytes> bytes) free
```

     b. **USAGE**
        - Vor fi afişaţi, pe câte o linie:
          - Numărul de octeţi folosiţi din arenă (numai secţiunile de date)
          - Eficienţa utilizării (în procente), egală cu numărul de octeţi folosiţi raportat la numărul de octeţi rezervaţi (care nu sunt liberi)
          - Fragmentarea (în procente), egală cu numărul de zone libere - 1, raportat la numărul de blocuri alocate. Pentru o arenă fără nici un bloc alocat, fragmentarea va fi considerată 0.
        - Formatul afişării este:

```
<nblocks> blocks (<nbytes> bytes) used
<eff>% efficiency
<fragm>% fragmentation
```

c. **ALLOCATIONS**

- Vor fi afişate pe câte o linie, zonele libere şi alocate, în ordinea în care sunt aşezate în arenă. Fiecare linie va fi de forma:

```
{FREE|OCCUPIED} <N>
```

- unde `{ .. | .. }` reprezintă faptul că doar una dintre valori va fi afişată. `N` reprezintă dimensiunea (nenulă), în octeţi, a zonei respective.

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. De exemplu va trebui să executaţi întocmai comenzi care cer scrierea în zone de memorie nealocate sau rezervate gestiunii, exact aşa cum în C puteţi realiza operaţii invalide cu pointeri şi programul să vă dea Segmentation Fault.

Exemple

În această secţiune sunt ilustrate câteva exemple de rulare a programului, pentru a înţelege mai bine modul în care programul vostru trebuie să se comporte. Fiecare exemplu este urmat apoi de câteva explicaţii.

Exemplul 1

```
INITIALIZE 100
ALLOC 13
16
FILL 16 13 255
DUMP
00000000 04 00 00 00 00 00 00 00 00 00 00 19 00 00 00
00000010 FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00
00000064
FREE 16
ALLOC 50
16
ALLOC 40
0
ALLOC 30
0
ALLOC 20
78
FILL 78 20 127
DUMP
00000000 04 00 00 00 42 00 00 00 00 00 00 00 3E 00 00 00
00000010 FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 04 00 00 00 20 00 00 00 7F 7F
00000050 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F
00000060 7F 7F 00 00
00000064
FREE 16
FREE 78
FINALIZE
```

Observaţii:

- Comenzile (intrarea) este marcată cu albastru, iar ieşirea cu negru. Programul vostru nu trebuie să afişeze nimic altceva în afara a ce este ilustrat în acest exemplu.
- În output-ul de `DUMP`, au fost marcate cu verde zonele de gestiune. În primul dump, în primul chenar se poate recunoaşte indexul de start, în al doilea chenar secţiunea de gestiune a blocului alocat. Cu roşu au fost figurate datele utilizatorilor.

- Se poate observa că cererile de alocare prea mari au fost respinse, afişându-se 0 (care este un index de date invalid, pentru că pe poziţia 0 stă indexul de start, şi nu se poate aloca memorie în acea zonă).

Exemplul 2

```
INITIALIZE 100
ALLOC 10
16
ALLOC 10
38
ALLOC 10
60
ALLOC 10
82
ALLOC 10
0
FREE 16
FREE 60
FILL 38 10 255
FILL 82 10 255
DUMP
00000000 1A 00 00 00 1A 00 00 00 00 00 00 00 16 00 00 00
00000010 00 00 00 00 00 00 00 00 00 00 46 00 00 00 00 00
00000020 00 00 16 00 00 00 FF FF FF FF FF FF FF FF FF FF
00000030 46 00 00 00 1A 00 00 00 16 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 1A 00 00 00 16 00
00000050 00 00 FF FF FF FF FF FF FF FF FF FF 00 00 00 00
00000060 00 00 00 00
00000064
FINALIZE
```

În acest exemplu se poate observa modul în care sunt actualizate secţiunile de gestiune ale vecinilor blocurilor eliberate, pentru a le înlătura din lanţul de blocuri. De asemenea, se poate observa în dump faptul că un bloc eliberat nu este modificat (resetat pe 0, de exemplu).

Exemplul 3

```
INITIALIZE 100
ALLOC 20
16
ALLOC 20
48
SHOW FREE
1 blocks (32 bytes) free
SHOW USAGE
2 blocks (40 bytes) used
58% efficiency
0% fragmentation
SHOW ALLOCATIONS
OCCUPIED 4 bytes
OCCUPIED 32 bytes
OCCUPIED 32 bytes
FREE 32 bytes
FREE 16
SHOW FREE
2 blocks (64 bytes) free
SHOW USAGE
1 blocks (20 bytes) used
55% efficiency
100% fragmentation
SHOW ALLOCATIONS
OCCUPIED 4 bytes
FREE 32 bytes
OCCUPIED 32 bytes
FREE 32 bytes
FINALIZE
```

În acest exemplu se poate observa cum se modifică statisticile arenei pe măsură ce blocurile sunt alocate şi eliberate.

BONUS

Se va acorda un bonus de **20 de puncte** (pe lângă faimă şi respect :)), dacă se vor implementa, pe lângă comenzile standard prezentate mai devreme, şi următoarele comenzi:

1. **ALLOCALIGNED <SIZE> <ALIGN>**
   - Această comandă va funcţiona ca ALLOC, cu excepţia faptului că indexul returnat va trebui să fie aliniat la ALIGN octeţi, unde ALIGN este o putere a lui 2 (poate fi 1, 2, 4, 8, etc.). Un index INDEX este aliniat la ALIGN octeţi dacă INDEX % ALIGN == 0.
2. **REALLOC <INDEX> <SIZE>**
   - Această comandă va realoca o zonă de memorie întoarsă în prealabil la adresa INDEX într-un nou spaţiu de memorie de dimensiune SIZE şi va afişa indexul secţiunii de date a noului bloc alocat. De asemenea, va copia datele aflate în vechiul bloc în noul bloc. Daca SIZE este mai mic decât dimensiunea originală, vor fi copiaţi numai SIZE octeţi (va avea loc o trunchiere).
   - **Anteţie:** Pentru găsirea unei zone de memorie libere va trebui să reluaţi procedura de căutare de la stânga la dreapta. Nu este valid să verificaţi că în locul curent există deja spaţiu pentru expansiune / micşorare.
3. **SHOW MAP <LENGTH>**
   - Comanda va afişa pe mai multe linii un şir de LENGTH caractere, fiecare caracter fiind fie "*" sau ".", care va descrie zonele libere sau ocupate din arenă. Un caracter este "*" dacă în zona descrisă de el se află cel puţin un octet rezervat, altfel el va fi ".". Fiecare linie va afişa maxim 80 de astfel de caractere. Dacă dimensiunea arenei este N, atunci un caracter va reprezena x = N / LENGTH octeți. Dacă cel puțin unul din cei x octeți este ocupat, se va afișa "*", altfel ".". Atenție, x poate fi și subunitar.

Exemplu

```
INITIALIZE 100
ALLOCALIGNED 10 32
32
SHOW ALLOCATIONS
OCCUPIED 4 bytes
FREE 16 bytes
OCCUPIED 22 bytes
FREE 58 bytes
FILL 32 10 255
SHOW MAP 50
**........**********..........
SHOW MAP 31
**....********.........
SHOW MAP 2
*.
SHOW MAP 200
********.....**********************************
****.....
.....
DUMP
00000000 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000010 00 00 00 00 00 00 00 00 00 00 00 00 16 00 00 00
00000020 FF FF FF FF FF FF FF FF FF FF 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00
00000064
REALLOC 32 50
16
DUMP
00000000 04 00 00 00 00 00 00 00 00 00 00 00 3E 00 00 00
00000010 FF FF FF FF FF FF FF FF FF FF 00 00 16 00 00 00
00000020 FF FF FF FF FF FF FF FF FF FF 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00
00000064
SHOW MAP 100
**********.....
.....
```

Testare

Testarea temei se va face folosind un script de evaluare automata, ce poate fi descărcat la această adresă [http://ocw.cs.pub.ro/courses/_media/programare/tema4_eval.zip].

Instrucţiuni de utilizare

- Arhiva se dezarhivează în directorul vostru de lucru (acolo unde este compilat executabilul `./allocator`).
- Arhiva conţine un fişier `Makefile.test`, din care se porneşte operaţia de testare. Pentru a rula varianta standard de testare (fără bonus), rulaţi:
  - `make -f Makefile.test test`
- Pentru a rula varianta de testare cu bonus, rulaţi:
  - `make -f Makefile.test test-bonus`
- Pentru a curăţa toate fişierele de ieşire generate pe parcursul testării, rulaţi:
  - `make -f Makefile.test clean`
- Dacă apar erori şi testele eşuează, puteţi să vă uitaţi în directorul `_tests` şi să comparaţi rezultatele voastre (fişierele `.out`), cu cele ale implementării de referinţă (fişierele `.out.ref`).
- Arhiva include şi un binar al implementării de referinţă (`./reference`), care este folosit intern de tester. Puteţi de asemenea să-l rulaţi separat şi să experimentaţi cu el şi alte situaţii.

Alte precizări

- Tema va fi trimisă atât pe vmchecker [https://elf.cs.pub.ro/vmchecker/] cât și pe moodle [http://cs.curs.pub.ro], sub forma unei arhive ZIP.
- Arhiva va trebui să conţină `în directorul rădăcină` doar următoarele:
  - Codul sursă al programului vostru (fişierele `.c` şi eventual `.h`).
  - Un fişier `Makefile` care să conţină regulile `build` şi `clean`. Regula `build` va compila programul într-un executabil cu numele **`allocator`**. Regula `clean` va şterge executabilul şi eventual toate binarele intermediare (fişiere obiect) generate de voi.
  - Un fişier `README` care să conţină prezentarea implementării alese de voi. Dacă aţi implementat şi bonusul, menţionaţi acest lucru în README.
  - Un fișier gol `bonus` dacă ați implementat și bonus-ul (folosit intern de vmchecker pentru a determina dacă trebuie să ruleze sau nu și aceste teste)
- O temă care nu compilează nu va primi nici un punct.
- Criteriile de notare sunt următoarele:
  - **`70 puncte`** - testele automate din arhiva de testare.
  - **`20 puncte`** - calitatea şi eficienţa implementării, utilizarea corespunzătoare a pointerilor/memoriei.
  - **`10 puncte`** - explicaţiile din README şi aspectul codului sursă.
  - **`20 puncte`** - implementarea bonusului.

Barem corectare

- Teste **basic**: 30 puncte

- Teste **advanced**: 25 puncte
- Teste **random**: 15 puncte
- Implementare **bonus**: 20 puncte

Mențiuni suplimentare:

- dacă pentru o anumită categorie de teste nu au trecut toate testele, am punctat astfel:
    - teste **basic**: 10 puncte fiecare test trecut
    - teste **advanced**: 5 puncte fiecare test trecut
    - teste **random**: 5 puncte fiecare test trecut
    - teste **bonus**: 5 puncte fiecare test trecut
- cele 10 puncte pentru README și aspectul codului sursă s-au împărțit astfel:
    - existența unui README relevant: 2 puncte
    - claritatea codului sursă: 8 puncte
- cele 20 puncte pentru calitatea şi eficienţa implementării, utilizarea corespunzătoare a pointerilor/memoriei s-au împărțit astfel:
    - calitatea și eficiența implementării: 10 puncte
        - am scăzut 5 puncte dacă codul nu este modularizat
            - am scăzut 2 puncte dacă codul este modularizat, dar folosește funcții foarte lungi
        - am scăzut 3 puncte pentru warning-uri de compilare
    - utilizarea corespunzătoare a pointerilor/memoriei: 10 puncte
        - am scăzut 5 puncte dacă memoria nu a fost alocată dinamic
        - am scăzut 2 puncte dacă memoria a fost alocată dinamic, dar nu s-a eliberat la final memoria
        - am scăzut 5 puncte dacă programul are accese nevalide la memorie (testat cu valgrind)