Laborator 06: Structuri, vectori, explorarea memoriei

În acest laborator vom introduce noțiunea de structură din limbajul assembly, vom lucra cu operații specializate pe șiruri și vom vedea câteva avantaje – legate de securitate și debugging – a explorării memoriei.

Structuri

Structurile sunt folosite pentru a grupa date care au tipuri diferite, dar care pot fi folosite împreună pentru a crea un tip compus.

În continuare vom trece prin pașii necesari pentru a folosi o structură: declararea, instanțierea și accesarea câmpurilor unei structuri.

Declararea unei structuri

În NASM, o structură se declară folosind construcția `struc <nume structura>`, urmată de o listă de câmpuri și încheiată cu `endstruc`.

Fiecare câmp al structurii este definit prin următoarele: o etichetă (folosită pentru a putea accesa membrii), specificatorul de tip și numărul de elemente.

Exemplu:

```
struc mystruct
    a:    resw 1    ; a va referi un singur element de dimensiune un cuvânt
    b:    resd 1    ; b va referi un singur element de dimensiune un dublu
cuvânt
    c:    resb 1    ; c va referi un singur element de dimensiune un octet
    d:    resd 1    ; d va referi un singur element de dimensiune un dublu
cuvânt
    e:    resb 6    ; e va referi 6 elemente de dimensiune un octet
endstruc
```

Aici sunt folosite pseudo-instrucțiunile NASM din familia `res` pentru a defini tipul de date și numărul de elemente pentru fiecare dintre câmpurile structurii. Pentru mai multe detalii despre sintaxa `res` urmați acest link: http://www.nasm.us/doc/nasmdoc3.html#section-3.2.2

Fiecare etichetă ce definește un câmp reprezintă offset-ul câmpului în cadrul structurii. De exemplu, b va avea valoarea 2, deoarece sunt 2 octeți de la începutul structurii până la câmpul b (primii 2 octeți sunt ocupați de cuvântul a).

Dacă doriți să folosiți același nume de câmp în două structuri diferite, trebuie să prefixați numele etichetei cu `.` (dot) astfel:

```
struc mystruct1
    .a:    resw  1
    .b:    resd  1
endstruc

struc mystruct2
    .a:    resd  16
    .b:    resw  1
endstruc
```

Folosiți contrucția `mystruct2.b` pentru aflarea valorii offset-ului lui 'b' din cadrul structurii mystruct2.

Instanțierea unei structuri

O primă variantă pentru a avea o structură în memorie este de a declara-o static în secțiunea `.data`. Sintaxa folosește macro-urile NASM `istruc` și `iend` și keyword-ul `at`.

În exemplul următor este prezentată instanțierea statică a structurii declarate mai sus, unde `struct_var` este adresa din memorie de unde încep datele.

```
struct_var:
    istruc mystruct
        at a, dw        -1
        at b, dd        0x12345678
        at c, db        ' '
        at d, dd        23
        at e, db        'Gary',
    iend
```

Pentru a nu inițializa valorile membrilor greșit, va trebui să aveți grijă ca pentru fiecare câmp, tipul de date din instanțiere să corespundă tipului din declarare.

Alocarea dinamică a unei structuri

Pentru a aloca dinamic o structură vom folosi un apel al funcției `malloc`. Va trebui să cunoaștem dinainte dimensiunea structurii. În cazul exemplului nostru, o instanțiere a structurii are 17 octeți.

În primul rând vom avea în secțiunea `.data` dimensiunea structurii și pointer-ul care va fi setat la valoarea pe care o va întoarce `malloc`.

```
section .data
    struct_size:   dd 17
    struct_ptr:    dd
```

La un moment dat, în secțiunea `.text` vom avea apelul `malloc(17)` care va aloca memorie de pe heap și va întoarce adresa de început a zonei alocate.

```
    mov eax, [struct_size]        ; pregătim apelul funcției malloc
    push eax
    call malloc
    mov dword [struct_ptr], eax  ; salvăm adresa întoarsă de malloc
    add esp, 4
```

Accesarea valorilor dintr-o structură

Pentru a accesa și/sau modifica un anumit membru al structurii instanțiate trebuie să îi cunoaștem adresa. Această adresă se poate obține calculând suma dintre adresa de început a structurii și offset-ul din cadrul structurii al membrului dorit .

Următoarea secvență de cod prezintă punerea unei valori în câmpul b al structurii și, ulterior, afișarea valorii acestui câmp.

```
    mov eax, 12345
    mov dword [struct + b], eax ; adresa câmpului b este adresa de bază a
structurii instanțiate static + offset-ul câmpului (dat de eticheta 'b')

    mov ebx, dword [struct + b] ; punerea valorii din câmpul b în registrul
ebx pentru afișare
    PRINT_DEC 4, ebx
    NEWLINE
```

Vectori

Putem considera un vector ca o înșiruire de elemente de același tip, plasate contiguu în memorie. Ați observat ceva similar în laboratoarele trecute când declaram static șiruri de caractere în secțiunea `.data`.

Declararea unui vector

În general, datele statice declarate pot fi inițializate sau neinițializate. Diferențierea se face atât prin faptul că la datele inițializate oferim o valoare inițială, dar și prin sintaxa NASM folosită.

De exemplu, pentru a declara un vector de 100 de cuvinte inițializate cu valoarea 42, vom folosi construcția:

```
section .data
    myVect:    times 100    dw 42
```

Pe de altă parte, dacă dorim declararea unui vector de 20 de elemente dublu cuvinte neinițializate, folosim instrucțiuni din familia `res` astfel:

```
section .bss
    myVect:    resd 20
```

Instrucțiuni de operare pe șiruri

Deoarece operațiile pe vectori sunt des întâlnite în programe, au fost implementate instrucțiuni speciale care facilitează: transferul de date între doi vectori, compararea a doi vectori, găsirea unui element într-un vector, parcurgerea unui vector etc.

O instrucțiune pe vectori poate avea un operand sursă, unul destinație, sau pe amândoi. Convențional, șirul sursă se află poziționat în segmentul DS, iar șirul destinație în ES. Mai mult, registrul SI este utilizat ca offset pentru adresa elementului curent din șirul sursă, iar DI este offset pentru șirul destinație.

Deși fiecare dintre aceste instrucțiuni care vor fi prezentate în continuare pot fi folosite independent, există o construcție specială pentru a crea bucle, prin prefixarea instrucțiunii de operare pe șiruri cu una dintre următoarele mnemonici:

- `rep` - repetă cât timp ECX != 0
- `repe`/`repz` - repeat while »equal« (ex: repetă până găsim un element diferit în vector)
- `repne`/`repnz` - repeat while »not equal« (ex: repetă până găsim un element comun în vector)

Utilizarea unuia dintre aceste prefixe are ca efect repetarea instrucțiunii prin hardware, fapt care duce la o îmbunătățire a performanței (și chiar a memoriei, datorită eliminării surplusului de instrucțiuni ca `jmp` și `cmp`). Aceste bucle se numesc și **bucle hardware**.

Pe lângă registrele ESI și EDI, instrucțiunile din familia `rep` mai folosesc următoarele resurse:

- registrul ECX
- flag-ul Zero (ZF) - prin care se verifică condiția de egalitate/inegalitate, în cazul instrucțiunilor `repz` și `repnz`
- flag-ul Direction (DF) - prin care se specifică dacă registrele ESI și EDI se incrementează (DF = 0) sau de decrementează (DF = 1) după fiecare instrucțiune de operare pe șiruri.

În continuare vor fi prezentate în detaliu instrucțiunile folosite pentru lucrul cu vectori.

MOVS (Move data from string to string)

Se transferă un element (octet/cuvânt/dublu cuvânt) de la sursă (DS:SI) la destinație (ES:DI) și se actualizează ESI și EDI pentru a face referire la următorul element din șir.

Utilizată împreună cu prefixul `rep`, realizează un transfer de bloc memorie-memorie.

Dacă instructiunea conține numele operanzilor, asamblorul poate infera tipul șirului; dacă nu, trebuie specificat în mod explicit tipul operației: pe byte, word sau double word. Astfel, prototipurile posibile pentru această instrucțiune sunt:

```
    movs <sir_dest>, <sir_src>
    movsb, movsw, movsd
```

CMPS (Compare strings)

Instrucțiunea realizează comparația dintre valoarea aflată la EDI și cea aflată la ESI (în această ordine, deci invers față de un CMP normal), și actualizează în mod corespunzător registrul de indicatori.

Împreună cu prefixul `repe/repz`, instrucțiunea determină prima pereche de elemente diferite din cele doua șiruri.

Formele in care poate apărea această instrucțiune sunt similare cu instrucțiunea `movs`:

```
    cmps <sir_dest>, <sir_src>
    cmpsb, cmpsw, cmpsd
```

SCAS (Scan string)

Această instrucțiune realizează o comparație între elementul curent al sirului destinație (ES:DI) și acumulator (AL/AX/EAX), și actualizează flag-urile. Ce de obicei, sunt actualizate registrele SI și DI.

Mnemonici posibile:

```
    scas <sir_dest>
    scasb, scasw, scasd
```

LODS (Load from string)

Instrucțiunea transferă elementul situat la adresa DS:SI în acumulator (AL/AX/EAX), și actualizează SI.

Nu are sens ca această instructiune să fie însoțită de prefixul de repetare, deoarece în acumulator ar rămâne numai ultimul element transferat. Din acest motiv, instrucțiunea se folosește numai în bucle soft.

```
    lods <sir_src>
    lodsb, lodsw, lodsd
```

STOS (Store to string)

Instrucțiunea transferă un element din acumulator (AL/AX/EAX) în șirul destinație (adresat de ES:DI), actualizând DI pentru a indica la urmatorul element. Dacă e folosit împreună cu prefixul de repetare, putem inițializa un șir cu o constantă.

```
    stos <sir_dest>
    stosb, stosw, stosd
```

Vectori de structuri

Adesea vom avea nevoie de vectori care să conțină elemente de dimensiuni mai mari decât cea a unui cuvânt dublu. Pentru a obține acest lucru vom combina cele două concepte prezentate anterior și vom folosi vectori de structuri. Bineînțeles, instrucțiunile de operare pe șiruri nu vor funcționa, deci vom fi nevoiți să ne întoarcem la metoda clasică de accesare a elementelor: cea prin adresarea explicită a memoriei.

Pentru exemplul din această secțiune, creăm o structură ce reprezintă un punct într-un spațiu 2D.

```
struc point
    .x:    resd 1
    .y:    resd 1
```

Declararea unui vector de structuri

Deoarece NASM nu suportă niciun mecanism pentru a declara explicit un vector de structuri, va trebui să declarăm efectiv o zonă de date în care să încapă vectorul nostru.

Considerând că ne dorim un vector zeroizat de 100 de elemente de tipul structurii `point` (care este de dimensiune 8 octeți), trebuie să alocăm 100 * 8 (= 800) octeți.

Obținem:

```
section .data
    pointArray:    times 800    db
```

În plus, NASM oferă o alternativă la calculul “de mână” al dimensiunii unei structuri, generând automat macro-ul `<nume structura>_size`. Astfel, exemplul anterior poate deveni:

```
section .data
    pointArray:    times point_size * 100    db
```

Parcurgerea unui vector de structuri

Cum am mai spus, pentru accesarea câmpului unui element dintr-un vector trebuie să folosim adresarea normală (în particular adresarea “based-indexed with scale”). Formula pentru aflarea adresei elementului este `baza_vector + i * dimensiune_struct`.

Presupunând că avem în registrul ebx adresa de început a vectorului și în eax indicele elementului pe care dorim să îl accesăm, exemplul următor prezintă afișarea valorii câmpului y a acestui element.

```
    mov ebx, pointArray                          ; mutăm în ebx adresa de
început a șirului
    mov eax, 13                                  ; să zicem că vrem al 13-lea
element
    mov edx, [ebx + point_size * eax + point.y] ; calcularea adresei
```

```
câmpului dorit

    PRINT_UDEC 4, edx
    NEWLINE
```

Pentru parcurgerea vectorului putem folosi instrucțiunea `loop`, având la fiecare iterație, în registrul eax, indicele curent. Putem să afișăm valorile din ambele câmpuri ale fiecărui element din vector cu următorul program:

```
%include "io.inc"

struc   point
    .x: resd 1
    .y: resd 1
endstruc

section .data
    pointArray: times point_size * 100 db
    print_str:  db "(%d, %d)", 13,

section .text
    global CMAIN
    extern printf

CMAIN:

    push ebp
    mov ebp, esp

    xor edx, edx
    xor eax, eax
    mov ecx, 100 ; dimensiunea vectorului; ecx este folosit pentru loop
label:
    push eax ; salvăm registrele
    push ecx ; salvăm registrele

    mov edx, [pointArray + point_size * eax + point.x] ; accesăm membrul y
    push edx ; ultimul parametru pentru printf

    mov edx, [pointArray + point_size * eax + point.x] ; accesăm membrul x
    push edx ; al doilea parametru pentru printf

    push print_str ; șirul format pentru printf
    call printf
    add esp, 12

    pop ecx
    pop eax
    inc eax ; incrementarea indicelui de iterare
    loop label
```

```
    leave
    ret
```

Explorarea memoriei

Explorarea memoriei poate fi necesară în mai multe cazuri, printre care:

- inspectarea memoriei unui program după un crash al acestuia pentru a determina cauzele crash-ului
- inspectarea memoriei pentru a găsi vulnerabilități în program
- descoperirea de *memory leak*-uri
- căutarea după o anumită valoare

Majoritatea utilitarelor de debugging și analiză dinamică (gdb, IDA, radare2) suportă, într-un fel sau altul, explorarea memoriei.

Unul dintre scopurile explorării memoriei este să afișăm toate valorile din memorie care încep de la o anumită adresă (și, opțional, se termină la o altă adresă). Un caz practic ar fi să afișăm *stack frame*-ul curent, adică ce se află între esp și ebp (după alocarea variabilelor locale).

```
%include "io.inc"

section .data
    print_str db "%x", 13,  ; afișăm memoria în hexadecimal
section .text
    global CMAIN
    extern printf
CMAIN:
    push ebp
    mov ebp, esp

    sub esp, 128    ; (simulare de) alocare de memorie

    xor eax, eax
    mov ebx, esp    ; ebx este interatorul prin memorie

label:
    push ebx

    mov eax, [ebx]

    ; afișarea a 4 octeți din memorie
    push eax
    push print_str
    call printf
    add esp, 8


    pop ebx
```

```
    add ebx, 4      ; incrementarea adresei pentru a referi la următorii 4
octeți
    cmp ebx, ebp    ; condiția de terminare
    jl label

    leave
    ret
```

Un alt exemplu relevant este căutarea unei anumite valori într-o zonă de memorie. Pentru a implementa asta putem folosi una dintre instrucțiunile de operare pe șiruri `scas`, într-o buclă hardware. În următorul exemplu căutarea se face tot în *stack-frame*-ul curent (între esp și ebp).

```
%include "io.inc"

section .text
global CMAIN
CMAIN:
    push ebp
    mov ebp, esp

    sub esp, 16         ; alocare de variabile locale

    mov ecx, 4          ; numărul de dublu cuvinte dintre esp și ebp (16 / 4
== 4)
    mov eax, 0x12345678 ; valoarea căutată
    mov edi, esp        ; esp este adresa de început a "șirului"
    cld                 ; resetarea bit-ului de direcție pentru a avea
incrementarea lui DI (DF = 0)
    repne scasd
    jne finish          ; dacă nu a găsit în ecx pași

    PRINT_STRING "Found"
finish:
    leave
    ret
```

Pe de altă parte, dacă dorim să ignorăm o anumită valoare (spre exemplu 0) putem folosi

```
    mov al,
    repe scasb       ; ignoră zero-uri; edi va referi la următoarea adresă
după primul element diferit de 0
    dec edi
    PRINT_HEX 4, edi ; afișează primul număr diferit de zero
```

Tutoriale și exerciții

În cadrul exercițiilor vom folosi arhiva de laborator.

Descărcați arhiva, decomprimați-o și accesați directorul aferent.

[1p] 1. Tutorial: Afișare a conținutului unei structuri

În programul `print_structure.asm` sunt afișate câmpurile unei structuri.

Urmăriți codul, observați construcțiile și modurile de adresare a memoriei. Rulați codul.

Treceți la următorul pas doar după ce ați înțeles foarte bine ce face codul. Vă va fi greu să faceți următorul exercițiu dacă aveți dificultăți în înțelegerea exercițiului curent.

[1.5p] 2. Modificare a unei structuri

Scrieți cod în cadrul funcției `main` astfel încât să modificați câmpurile structurii `sample_student` pentru ca

- anul nașterii să fie 1993
- vârsta să fie 22
- grupa să fie 323CA

Trebuie să modificați conținutul structurii din cod, adică trebuie să scrieți în zona de memorie aferentă câmpului din structură. Nu modificați structura din secțiunea `.data`, este vorba să folosiți cod pentru a modifca structura.

Pentru modificarea grupei, va trebui să schimbați al treilea octet/caracter al câmpului `group` (adică octetul/caracterul cu indexul 2).

[1p] 3. Tutorial: Alocare a unei structuri pe stivă

În programul `on_stack_structure.asm` se alocă o structură ca o variabilă locală funcției `main`: se face loc pe stivă folosind `sub esp, 80` și apoi se inițializează câmpuri din structură și se afișează. Urmăriți codul, observați construcțiile și modul de adresare a memoriei.

[1.5] 4. Alocare a câmpurilor unei structuri pe stivă

Actualizați programul de mai sus pentru a completa și celelalte câmpuri așa cum este indicat în comentariul marcat cu `TODO`.

Folosiți construcția `rep movsb` pentru a completa câmpurile de tip string (array de bytes).

[0.5p] 5. Tutorial: Prelucrare a unei structuri

În programul `process_structure.asm` câmpul `id` al variabilei `sample_student` (de tip structură `stud_struct`) este populat cu inițialele de la prenumele și numele studentului. Urmăriți codul,

observați construcțiile și modul de adresare a memoriei.

[2p] 6. Prelucrare a unei structuri

Actualizați programul de anterior astfel încât câmpul `id` să fie inițializat la primele 3 litere din prenume, urmate de primele trei litere din nume, urmate de semnul `-` (*minus*) și urmate de numele grupei. Adică pentru intrarea definită în fișier, afișarea va însemna mesajul *AndVoi-323CA*.

Folosiți construcția `rep movsb` pentru a completa câmpul `id` cu secvențele de subșiruri din prenume, nume și grupă.

[1p] 7. Tutorial: Populare a unui vector de structuri

În programul `structure_array.asm` este definită o variabilă `students` similară unui vector de structuri. În cadrul programului se face inițializarea și afișarea acestui vectori de structuri, folosindu-se șirul vid (primul caracter este) în cadrul câmpurilor de tip șir. Urmăriți codul, observați construcțiile și modul de adresare a memoriei Rulați codul.

[1.5p] 8. Alocarea și populare unui vector de structuri

Actualizați programul de mai sus astfel încât vectorul de structuri să nu mai fie o variabilă globală neinițilizată (în `.bss`) ci să fie alocat pe stivă.

[1.5p] 9. Bonus: Alocare dinamică

Realizați o alocare dinamică (cu `malloc`) a unui șir/array de bytes de o dimensiune dată (să spunem 128 de octeți) și apoi inițializați acel șir cu octetul/caracterul `'a'`.

Pentru inițializare folosiți `rep stosb`.

[1.5p] 10. Bonus: Dump la zonă de memorie

Faceți *dump* la zona de memorie alocată dinamic înainte și după inițializarea ei. Prin *memory dump* înțelegem să fie afișat fiecare octet în format hexazecimal.

[1p] 11. Bonus: Dump pe zona de cod a procesului curent

Faceți *dump* la zona de memorie de cod (text) a procesului curent. Începeți de la adresa funcției main și afișați un număr dat de octeți/caractere (de exemplu `100`).