Laborator 10: Interactiunea C-assembly

Având în vedere că limbajul de asamblare prezintă dificultăți atât în citirea cât și în dezvoltarea codului, tendința generală este aceea de a se migra către limbaje de nivel înalt (care sunt mult mai ușor de citit și oferă un API mult mai ușor de utilizat). Cu toate acestea, tot există situații în care, din rațiuni de optimizare, se folosesc mici rutine assembly care sunt integrate în modulul limbajului de nivel inalt.

În acest laborator vom vedea cum se pot integra module de assembly în programe C și viceversa.

Utilizarea procedurilor assembly în funcții C

Pentru ca un program C să ajungă să fie executat, este necesar ca acesta să fie tradus în codul mașina al procesorului; aceasta este sarcina unui compilator. Având în vedere că acest cod rezultat în urma compilării nu este întotdeauna optim, în anumite cazuri se preferă înlocuirea unor porțiuni de cod scris în C cu porțiuni de cod assembly care să facă același lucru, însă cu o performanță mai bună.

Declararea procedurii

Pentru a ne asigura că procedura assembly și modulul C se vor combina cum trebuie și vor fi compatibile, următorii pași trebuie urmați:

- declararea labelului procedurii ca fiind global, folosing directiva GLOBAL. Pe lângă asta, orice date care vor fi folosite de către procedură trebuie declarate ca fiind globale.
- folosirea directivei EXTERN pentru a declara procedurile și datele globale ca fiind externe.

Setarea stivei

Atunci când se intră intr-o procedură, este necesar să se seteze un stack frame către care să se trimită parametrii. Desigur, dacă procedura nu primește parametri, acest pas nu este necesar. Așadar, pentru a seta stiva, trebuie inclus următorul cod:

```
push ebp
mov ebp, esp
```

EBP-ul ne oferă posibilitatea să îl folosim ca un index în cadrul stivei și nu ar trebui alterat pe parcursul procedurii.

Conservarea registrelor

Este necesar ca procedura să conserve valoarea registrelor ESI, EDI, EBP și a registrelor segment. În cazul în care aceste registre sunt corupte, este posibil ca programul să producă erori la întoarcerea din procedura assembly.

Transmiterea parametrilor din C către procedura assembly

Programele C trimit parametrii către procedurile assembly folosind stiva. Să considerăm următoarea secvență de program C:

```
extern int Sum();
   ...
int a1, a2, x;
   ...
x = Sum(a1, a2);
```

Când C-ul execută apelul către Sum, mai întâi face push la argumente pe stivă, în ordine inversă, apoi face efectiv call către procedură. Astfel, la intrarea în corpul procedurii, stiva va fi intactă.

Cum variabilele `a1` și `a2` sunt declarate ca fiind valori `int`, vor folosi fiecare câte un cuvânt pe stivă. Metoda aceasta de pasare a parametrilor se numește pasare prin valoare. Codul procedurii Sum ar putea arăta în felul următor:

```
Sum:
        push    ebp             ; creează stack frame pointer
        mov     ebp, esp
        mov     eax, [ebp+8]    ; ia primul argument
        mov     ecx, [ebp+12]   ; ia al doilea argument
        add     eax, ecx        ; suma celor 2
        pop     ebp             ; reface base pointerul
        ret
```

Este interesant de remarcat o serie de lucruri. În primul rând, codul assembly pune în mod implicit valoarea de retur a procedurii în registrul `eax`. În al doilea rând, comanda `ret` este suficientă pentru a ieși din procedură, datorită faptului că compilatorul de C se ocupă de restul lucrurilor, cum ar fi îndepărtarea parametrilor de pe stivă.

Apelarea de funcții C din proceduri assembly

În majoritatea cazurilor, apelarea de rutine sau funcții din biblioteca standard C dintr-un program în limbaj de asamblare este o operație mult mai complexă decât viceversa. Să luăm exemplul apelării funcției `printf` dintr-un program în limbaj de asamblare:

```
global  main

extern  printf

section .data

text    db      "291 is the best!", 10, 0
strformat db    "%s", 0

section .code

main
        push    dword text
        push    dword strformat
        call    printf
        add     esp, 8
        ret
```

Remarcați faptul că procedura este declarată ca fiind globală și se numește `main` - punctul de pornire al oricărui program C. Din moment ce în C parametrii sunt puși pe stivă în ordine inversă, offsetul stringului este pus prima oară, urmat de offsetul șirului de formatare. Funcția C poate fi apelată după aceea, însa stiva trebuie restaurată la ieșirea din funcție.

Când se face linkarea codului assembly trebuie inclusă și biblioteca standard C (sau biblioteca care conține funcțiile pe care le folosiți).

Exerciții

În cadrul laboratoarelor vom folosi repository-ul de git al materiei IOCLA - https://github.com/systems-cs-pub-ro/iocla [https://github.com/systems-cs-pub-ro/iocla]. Repository-ul este clonat pe desktop-ul mașinii virtuale. Pentru a îl actualiza, folosiți comanda `git pull origin master` din interiorul directorului în care se află repository-ul (`~/Desktop/iocla`). Recomandarea este să îl actualizați cât mai frecvent, înainte să începeți lucrul, pentru a vă asigura că aveți versiunea cea mai recentă. Dacă doriți să descărcați repository-ul în altă locație, folosiți comanda `git clone https://github.com/systems-cs-pub-`

ro/iocla ${target}.Pentru mai multe informații despre folosirea utilitarului `git`, urmați ghidul de la Git Immersion [https://gitimmersion.com/].

În acest semestru am avut parte de o participare redusă la cursurile PCLP2. Ne dorim să înțelegem care sunt motivele pentru această situație, așa că vă rugăm să completați acest formular de feedback [https://docs.google.com/forms/d/1kesynY60tv8pduYFdCzB3tfLhj1uYDbVIRHCCgsa9jE]. Formularul este anonim și necesită aproximativ 30 de secunde pentru completare. Rezultatele acestuia ne vor ajuta să vă înțelegem mai bine perspectiva și să ne adaptăm nevoilor voastre.

1. Tutorial: Calcul maxim în assembly cu apel din C

În subdirectorul `1-2-max-c-calls/` din arhiva de sarcini a laboratorului găsiți o implementare de calcul a maximului unui număr în care funcția `main()` este definită în C de unde se apelează funcția `get_max()` definită în limbaj de asamblare.

Urmăriți codul din cele două fișiere și modul în care se transmit argumentele funcției și valoarea de retur.

Compilați și rulați programul. Pentru a-l compila rulați comanda:

```
make
```

În urma rulării comenzii rezultă executabilul mainmax pe care îl putem executa folosind comanda:

```
./mainmax
```

Acordați atenție înțelegerii codului înainte de a trece la exercițiul următor.

Valoarea de retur a unei funcții este plasată în registrul `eax`.

2. Extindere calcul maxim în assembly cu apel din C

Extindeți programul de la exercițiul anterior (în limbaj de asamblare și C) astfel încât funcția `get_max()` să aibă acum semnătura `unsigned int get_max(unsigned int *arr, unsigned int len, unsigned int *pos)`. Al treilea argument al funcției este adresa în care se va reține poziția din vector pe care se găsește maximul.

La afișare se va afișa și poziția din vector pe care se găsește maximul.

Pentru reținerea poziției, cel mai bine este definiți o variabilă locală `pos` în funcția `main` din fișierul C (`main.c`) în forma

```
unsigned int pos;
```

iar apelul funcției `get_max` îl veți face în forma:

```
max = get_max(arr, 10, &pos);
```

3. Depanare stack frame corupt

În subdirectorul `3-stack-frame/` din arhiva de sarcini a laboratorului găsiți un program C care implementează afișarea stringului `Hello world!` printr-un apel al funcției `print_hello()` definită în assembly pentru prima parte a mesajului, urmat de două apeluri ale funcției `printf()` direct din codul C.

Compilați și rulați programul. Ce observați? Mesajul printat nu este cel așteptat deoarece din codul assembly lipsește o instrucțiune.

Folosiți GDB pentru a inspecta adresa din vârful stivei înainte de execuția instrucțiunii `ret` din funcția `print_hello()`. Către ce pointează? Urmăriți valorile registrelor EBP si ESP pe parcursul execuției acestei funcții. Ce ar trebui să se afle în vârful stivei după execuția instrucțiunii `leave`?

Găsiți instrucțiunea lipsă și rerulați executabilul.

Pentru a putea restaura stiva la starea sa de la începutul funcției curente, instrucțiunea `leave` se bazează pe faptul că frame pointerul funcției a fost setat.

4. Tutorial: Calcul maxim în C cu apel din assembly

În subdirectorul `4-5-max-assembly-calls/` din arhiva de sarcini a laboratorului găsiți o implementare de calcul a maximului unui număr în care funcția `main()` este definită în limbaj de asamblare de unde se apelează funcția `get_max()` definită în C.

Urmăriți codul din cele două fișiere și modul în care se transmit argumentele funcției și valoarea de retur.

Compilați și rulați programul.

Acordați atenție înțelegerii codului înainte de a trece la exercițiul următor.

5. Extindere calcul maxim în C cu apel din assembly

Extindeți programul de la exercițiul anterior (în limbaj de asamblare și C) astfel încât funcția `get_max()` să aibă acum semnătura `unsigned int get_max(unsigned int *arr, unsigned int len, unsigned int *pos)`. Al treilea argument al funcției este adresa în care se va reține poziția din vector pe care se găsește maximul.

La afișare se va afișa și poziția din vector pe care se găsește maximul.

Pentru a reține poziția, cel mai bine este să definiți o variabilă globală în fișierul assembly (`main.asm`) în secțiunea `.data`, în forma

```
pos: dd 0
```

Această variabilă o veți transmite (prin adresă) către apelul `get_max` și prin valoare pentru apelul `printf` pentru afișare.

Pentru afișare modificați șirul `print_format` și apelul `printf` în fișierul assembly (`main.asm`) ca să permită afișare a două valori: maximul și poziția.

6. Tutorial: Conservare registre

În subdirectorul `6-7-regs-preserve/` din arhiva de sarcini a laboratorului găsiți funcția `print_reverse_array()` implementată printr-un simplu loop ce face apeluri repetate ale funcției `printf()`.

Urmăriți codul din fișierul `main.asm`, compilați și rulați programul. Ce s-a întâmplat? Programul rulează la infinit. Acest lucru se întâmplă deoarece funcția `printf()` nu conservă valoarea din registrul ECX, folosit aici ca și contor.

Decomentați liniile marcate cu `TODO1` și rerulați programul.

7. Depanare SEGFAULT

Decomentați liniile marcate cu TODO2 în fișierul assembly de la exercițiul anterior. Secvența de cod realizează un apel al funcției `double_array()`, implementată în C, chiar înainte de afișarea vectorului folosind funcția văzută anterior.

Compilați și rulați programul. Pentru depanarea segfault-ului puteți folosi utilitarul `objdump` pentru a urmări codul în limbaj de asamblare corespunzător funcției `double_array()`. Observați care din registrele folosite înainte și după apel sunt modificate de această funcție.

Adăugați în fișierul assembly instrucțiunile pentru conservarea și restaurarea registrelor necesare.

8. Bonus: Calcul maxim în assembly cu apel din C pe 64 de biți

Intrați în subdirectorul `8-max-c-calls-x64/` și faceți implementarea calculului maximului în limbaj de asamblare pe un sistem pe 64 de biți. Porniți de la programul de la exercițiile 4 și 5 în așa fel încât să îl rulați folosind un sistem pe 64 de biți.

https://en.wikipedia.org/wiki/X86_calling_conventions [https://en.wikipedia.org/wiki/X86_calling_conventions].

Primul lucru pe care trebuie să-l aveți în vedere este că pe arhitectura x64 registrele au o dimensiune de 8 octeți și au nume diferite decât cele pe 32 de biți (pe lângă extinderea celor tradiționale: `eax` devine `rax`, `ebx` devine `rbx`, etc., mai există altele noi: R10-R15: pentru mai multe informații vedeți aici [https://stackoverflow.com/questions/20637569/assembly-registers-in-64-bit-architecture]).

De asemenea, pe arhitectura x64 parametrii nu se mai trimit pe stivă, ci se pun în registre. Primii 3 parametri se pun în: `RDI`, `RSI` și `RDX`. Aceasta nu este o convenţie adoptată uniform. Această convenţie este valabilă doar pe Linux, pe Windows având alte registre care sunt folosite pentru a transmite parametrii unei funcţii.

Convenția de apel necesită ca, pentru funcțiile cu număr variabil de argumente, `RAX` să fie setat la numărul de registre vector folosiți pentru a pasa argumentele. `printf` este o funcție cu număr variabil de argumente, și dacă nu folosiți alte registre decât cele menționate în paragraful anterior pentru trimiterea argumentelor, trebuie să setați `RAX = 0` înainte de apel. Citiți mai multe aici [https://stackoverflow.com/questions/38335212/calling-printf-in-x86-64-using-gnu-assembler].

9. Bonus: Calcul maxim în C cu apel din assembly pe 64 de biți

Intrați în subdirectorul `9-max-assembly-calls` și faceți implementarea calculului maximului în C cu apel din limbaj de asamblare pe un sistem pe 64 de biți. Porniți de la programul de la exercițiile 6 și 7 în așa fel încât să îl rulați folosind un sistem pe 64 de biți. Urmați indicațiile de la exercițiul anterior și aveți grijă la ordinea parametrilor.

Soluții

Soluțiile pentru exerciții sunt disponibile aici [https://elf.cs.pub.ro/asm/res/laboratoare/lab-10-sol.zip].