Arhitectura sistemelor de calcul

Programare multi-modul

Programarea multi-modul încurajează descompunerea problemelor complexe în sub-probleme și reutilizarea soluțiilor existente pentru a îmbunătăți eficiența.

Subprograme în limbajul de asamblare x86

O variantă de împărțire a codului o reprezintă modularizarea codului. Limbajul de asamblare nu recunoaște noțiunea de subprogram. Putem crea o secvență de instrucțiuni care să fie apelată din alte zone, iar după terminarea ei execuția programului se întoarce la punctul exact de unde procedura a fost apelată.

O astfel de secvență se numește procedură. Apelul unei proceduri se poate face și cu o instrucțiune jmp dar problema este că procesorul nu ține minte de unde a fost trimis la procedură și prin urmare nu știe unde să revină cu execuția după terminarea procedurii. Este necesar ca la apelul unei proceduri să salvăm undeva adresa de revenire,

iar revenirea din procedură este de fapt o instrucțiune de salt la acea adresă.

Locul unde se salvează adresa de revenire este **stiva de execuție**
Este nevoie de o stivă deoarece o procedură poate apela o altă procedură, acea procedură poate apela alta s.a.m.d.

Există două instrucțiuni ce permit apelul și revenirea din proceduri: **call** și **ret**.

Sintaxa:
call *eticheta*
Instrucțiunea **call** este de fapt o instrucțiune jmp care în plus introduce în vârful stivei adresa instrucțiunii care urmează - valoarea din EIP (instrucțiunea care vine imediat după call),
Instrucțiunea **ret** extrage din vârful stivei o adresă și execută un salt la acea adresă (se modifică EIP, valoarea extrasă de pe stivă este stocată în EIP). Instrucțiunea nu are argumente deoarece adresa la care sare programul este extrasă din vârful stivei.

Observație

Pentru evitarea ambiguităților atunci când apelăm o etichetă cu caracterul '.' la început înseamnă că ne referim la eticheta unei proceduri definite în programul local, altfel la cel principal.

Exemplu:

```
.eticheta ; etichetă utilizată în procedură
eticheta  ; etichetă globală
```

Asamblorul NASM oferă un mecanism simplu prin care un program poate fi împărțit în mai multe fișiere. Directiva `%include` (`#include` din C++) permite separarea declarațiilor programului în unul sau mai multe fișiere.

Se permite separarea declarațiilor în unul sau mai multe fișiere ce vor fi incluse acolo unde respectivele declarații sunt necesare. Întocmai cum și în C se obișnuiește separarea / gruparea declarațiilor în fișiere header (cu extensia .h) ce sunt ulterior incluse de codul scris în fișiere C care necesită declarațiile în cauză.

Exemplu:

lab11_procedura.asm

```
; programul calculeaza factorialul unui numar si afiseaza in consola rezultatul
; procedura factorial este definita in segmentul de cod si primeste pe stiva ca  si parametru un numar
bits  32
global  start
extern  printf, exit
import  printf msvcrt.dll
import  exit msvcrt.dll
segment  data use32 class=data
        format_string db "factorial=%d",  10, 13, 0
segment  code use32 class=code
;  urmeaza definirea procedurii
factorial:
        mov eax, 1
        mov ecx, [esp + 4]
        ; mov ecx, [esp + 4] scoate de pe stiva  parametrul procedurii
        ; ATENTIE!!! in capul stivei este adresa de  retur
        ; parametrul procedurii este imediat dupa  adresa de retur
        ; a se vedea desenul de mai jos
        ;
        ; stiva
        ;
        ;|------------------|
        ;| adresa retur  |  <-  esp
        ;|------------------|
        ;|   00000006h  |  <- parametrul pasat procedurii, esp+4
        ;|------------------|
        ; ....
        .repet:
                mul ecx
        loop .repet ; atentie, cazul ecx = 0 nu e  tratat!
        ret
;  programul "principal"
start:
        push dword 6        ; se salveaza pe stiva numarul  (parametrul procedurii)
        call factorial      ; apel procedura
        ; afisare rezultat
        push eax
        push format_string
        call [printf]
        add esp, 4*2
        push 0
        call [exit]
```

lab11_proc_main.asm - Diferența față de lab11_procedura.asm este ca procedura factorial este definită în alt fisier (factorial.asm) fiind necesara includerea acestuia folosind directiva %include.

```
;  programul calculeaza factorialul unui numar si afiseaza in consola rezultatul
;  procedura factorial este definita in fisierul factorial.asm
bits  32
global  start
import  printf msvcrt.dll
import  exit msvcrt.dll
extern  printf, exit
;  codul definit in factorial.asm va fi copiat aici
%include  "factorial.asm"
segment  data use32 class=data
        format_string db  "factorial=%d", 10, 13, 0
segment  code use32 class=code
start:
        push dword 6
        call factorial
        push eax
        push format_string
        call [printf]
        add esp, 2*4
        push 0
        call [exit]
```

factorial.asm

```
%ifndef  _FACTORIAL_ASM_ ; continuam daca _FACTORIAL_ASM_ este nedefinit
%define  _FACTORIAL_ASM_ ; si ne asiguram ca devine definit
;  astfel %include permite doar o singura includere
;definire  procedura
factorial:   ; int _stdcall factorial(int n)
        mov eax, 1
        mov ecx, [esp + 4]
        ;  mov ecx, [esp + 4] scoate de pe stiva parametrul procedurii
        ; pentru explicatii a se vedea  lab11_procedura.asm
        repet:
                mul ecx
        loop repet ; atentie, cazul ecx  = 0 nu e tratat!
        ret 4
%endif
```

Programe din mai multe module

Un program scris în limbaj de asamblare poate fi împărțit în mai multe fișiere sursă. Trebuie să se respecte următoarele reguli:

1) Toate segmentele vor fi declarate cu modificatorul **public**, pentru că la final se vor concatena segmentele de cod din fiecare modul, la fel și segmentul de date
2) Etichetele și variabilele dintr-un modul care trebuie "exportate" în alte module trebuie să facă obiectul unor declarații **global**.
3) Etichetele și variabilele care sunt declarate într-un modul și sunt folosite în alt modul trebuie să fie "importate" prin directiva **extern**.

De asemenea, trecerea execuției dintr-un modul în altul se poate face doar prin instrucțiuni de salt (jmp, call sau ret)
Fiecare modul se va asambla diferit folosind comanda:

nasm -fobj nume_modul.asm

apoi modulele se vor lega împreună cu comanda

alink modul_1.obj modul_2.obj ... modul_n.obj -oPE -subsys console -entry start

Etape asamblare / link-editare / executie

Asamblare:

masm -f obj modul.asm

Optiunea -f indică tipul de fișier care să fie generat, în cazul acesta fișier obj.

Link-editare

alink modul_1.obj ... modul_n.obj -oPE -subsys console -entry start

În folderul masm din asm_tools există fișierul "ALINK.TXT" care descrie opțiunile pentru alink.

Alink options:

-o x

x:
COM = output com file
EXE = output exe file
PE = output WIN32 PE file (.EXE)

-subsys x

x:
windows, win or gui => windows subsystem

console, con or char => console subsystem
native => native subsystem
posix => POSIX subsystem

-entry name

Optiunea specifică punctul de intrare în program (prima instrucțiune care se execută)

Debug:

OLYDBG.EXE modul.exe

Execuție:

modul.exe