piątek/tydzień nieparzysty/10:30 Wrocław, dn. 15. marca 2012

Artur Zochniak, nr albumu 184725

Implementacja funkcji rekurencyjnych w Assemblerze dla platformy Linux/x86

sprawozdanie z laboratorium kursu „Architektura komputerów 2”

rok akademicki 2011/2012, kierunek INF

Prowadzący:

prof. dr hab. inż. Janusz Biernat

Spis treści

Cele laboratorium

Celem labolatorium było:

- napisanie prostej funkcji rekurencyjnej w celu zapoznania się z tematem,
- napisanie funkcji rekurencyjnej wykorzystującej.

Podstawy wywoływania funkcji

Miejsce argumentów wywołania funkcji

Każde wywołanie funkcji poprzedzone jest przekazaniem jej parametrów. Parametry można przekazywać rejestrami (tak jak np. realizowane przy obsłudze przerwań - `int`) lub umieszczać na stosie (`push $4` jeśli argumentem jest liczba 4).
Funkcja rekurencyjna może działać poprawnie tylko jeśli ma indywidualny dla każdego wywołania kontekst. Kontekst ten można rozumieć jak zbiór zmiennych lokalnych, które parametryzują dane wywołanie funkcji, z tego powodu, aby wywołać, funkcję rekurencyjną należy przekazywać jej argumenty przez stos, w przeciwnym wypadku wszystkie wywołania funkcji będą pracować w tym samym kontekście.

Opis realizacji

Wywołując funkcję poleceniem `call fnc` procesor otrzymuje polecenie, aby z aktualnie przetwarzanego miejsca przeskoczył do miejsca w pamięci fnc. Jednak nie jest

ważne tylko gdzie skierujemy procesor, ale, równie bardzo ważną rzeczą jest to, czy procesor będzie wiedział gdzie ma wrócić.

Umożliwia to zachowanie tzw. *śladu skoku*, czyli miejsca, do którego funkcja, po zakończeniu wykonywania się powinna wrócić. Za każdym razem, przed wywoływaniem polecenia `call`, procesor umieszcza na stosie wartość rejestru `%EIP` (natomiast polecenie `ret` zdejmuje te wartość ze stosu) oraz otrzymuje informację o następnnej instrukcji programu, która zostałaby wykonana, gdyby nie wywołano funkcji, i zapisuje ją w rejestrze bazowym `%ebp`, `%bp` (ang. *(extended) base pointer*).

Przez czas, kiedy zwykład (nie rekurencyjna) funkcja jest wykonywana, rejestr `%ebp` nie jest modyfikowany, ale w przypadku funkcji rekurencyjnej należy liczyć się z tym, że jeszcze raz zostanie wywołana funkcja, a to wiązałoby się z utratą informacji zapisanej w `%ebp` (powędrowałby tam adres fragmentu ciała wykonywanej funkcji), aby uniknąć utraty informacji o miejscu wywołującym funkcję należy zapisać ślad skoku (rejestr `%ebp`) w miejscu, które umożliwiłoby przechowywanie indywidualnej informacji o każdorazowym wykonaniu funkcji rekurencyjnej. Takim miejscem jest stos.

Konwencją jest to, że pierwszym krokiem po wywołaniu funkcji jest zapisanie kopii rejestru wskaźnika bazy (`%ebp`) na stos i zaraz przed wyjściem z funkcji wczytanie go z tego samego miejsca, do którego został skopiowany (realizowane jest to poleceniami `push %ebp` oraz `pop %ebp`).

Odczytywanie argumentów funkcji w jej ciele

Pozostaje jeszcze jeden ważny aspekt. Łatwość odczytywania argumentów funkcji.

Stos budowany jest „w tył”, tj. im później informacja została do niego dodana, tym będzie się znajdowała w pamięci z adresem o mniejszej wartości.

Mechanizm działania funkcji

Zakładając, że program skompilowany jest dla architektury 32-bitowej, można wywnioskować, że długość pojedyńczego słowa to 4 bajty.

1. Na stos wędruje 4-bajtowa wartość wskaźnika %EIP,
◯ 2. następnie (zgodnie z konwencją) umieszczamy na stosie wartość rejestru bazowego `%ebp`,
☆ 3. aktualna wartość stosu znajduje się w rejestrze `%esp` (ang. *extended stack pointer*), jest to również informacja o kontekście wywołania funkcji (umożliwia odczytywanie argumentów wywołania). Konwencja mówi, że wartość tego rejestru możemy

skopiować do rejestru `%ebp` (którego wartość przed chwilą skopiowano na stos) i używać rejestru `%ebp` jako bazy dla kontekstu wywołania funkcji. np. aby dostać się do pierwszego argumentu w ciele funkcji nie trzeba dbać o to, czy modyfikowano stos, można po prostu skorzystać z notacji `8(%ebp)` dla pierwszego 4-bajtowego argumentu funkcji, `12(%ebp)` dla drugiego 4-bajtowego argumentu funkcji, itd.

▢ 4. należy pamiętać, aby przywrócić stan stosu do momentu wywoływania funkcji, realizuje się to wywołaniem `movl %ebp, %esp`. Po wywołaniu tej instrukcji możemy być pewni, że pierwszą wartością na stosie jest wartość rejestru bazowego `%ebp`, następnie wartość rejestru `%EIP` umieszczona na stosie przez polecenie `call`.

⬠ 5. Umieszczamy ponownie informację o śladzie skoku w rejestrze `%ebp` wywołując `pop %ebp`.

△ 6. Poleceniem `ret` wędrujemy do następnej instrukcji po wywołaniu funkcji i kontynuujemy pracę.

7. Wynik działania funkcji najczęściej wędruje do rejestru `%eax`.

Funkcja rekurencyjna: suma liczb od 1 do n

```
1.  .type sum @function
2.  sum:
3.    push %ebp      ⎫ ◯
4.    mov %esp, %ebp
5.  function_body:
6.    mov 8(%ebp), %ebx ⎫ ☆
7.    mov $1, %eax
8.    cmp $1, %ebx
9.    jbe finish_sum
10.   dec %ebx
11.   push %ebx
12.   call sum
13.   inc %ebx
14.   add %ebx, %eax
15. finish_sum:
16.   mov %ebp, %esp  ⎫ ▢
17.   pop %ebp    ⎫ ⬠
18.   ret ⎫ △
```

Dane wejściowe:

n sumę ile pierwszych liczb naturalnych obliczyć

Algorytm:

Funkcja zwraca sumę [`linia 14.`] *aktualne przekazanego argumentu* oraz *wartość funkcji wywołanej ze swoim argumentem wywołania pomniejszonym o jeden.*

Jeśli argumentem funkcji jest liczba mniejsza lub równa 1 [`linia 8.`], to funkcja zwraca 1 (`linia 7.`).

Przykład: program uruchomiony z argumentem działa następująco:

sum(10) = 10+sum(10-1)=10+9+sum(8) = … = 10+9+…+3+sum(1) = 55,

co jest zgodne z prawdą, gdyż: *10*(10+1)/2=55.*

Dane wyjściowe:

suma liczb od 1 do n.

Funkcja rekurencyjna generująca n-tą liczbę Fibbonaciego

```
1. generateNthFibbNr=6
2. EXIT=1
3. SYSC=0x80
4. STDIN = 0
5. STDOUT = 1
6. READ=3
7. WRITE=4
8. .section .data
9. .globl _start
10..globl exiting
11..type fibb @function
12.fibb:
13.   push %ebp
14.   mov %esp, %ebp
15.function_body:
16.   push %ebx # (A) function is using ebx as temporary register
17.   mov 8(%ebp), %ebx
18.   mov $1, %eax
19.   cmp $2, %ebx #first two fibb numbers are 1's
20.   jbe finish
21.started:
22.   dec %ebx
23.   push %ebx
24.   call fibb # now f(n-1) is in eax
25.   push %eax # (B) f(n-1) is on stack
26.   dec %ebx
27.   push %ebx
28.   call fibb #f(n-2) is in %eax
29.   pop %ebx # recovering f(n-1) from (B)
30.   add %ebx, %eax
31.   pop %ebx #neutralizing (A)
32.finish:
33.   mov %ebp, %esp
34.   pop %ebp
35.   ret
36._start:
37.   nop
38.   pushl $generateNthFibbNr
39.   call fibb
40.   result:
41.   mov %eax, %ebx
```

```
42.exiting:
43.   movl $EXIT, %eax
44.   int $SYSC
```

Dane wejściowe:

n – którą liczbę fibonacciego obliczyć.

Algorytm:

parametr wywołania funkcji jest zmniejszany o 1 [`linia 22.`] i otrzymana wartość staje się argumentem rekurencyjnego wywołania [`linia 23.`] – w wyniku otrzymujemy w `linii 25.` `n-1`-ty wyraz ciągu fibonacciego. Wynik znajduje się w rejestrze `%eax`, ale następne wywołanie funkcji zmieni ten rejestr, zatem wynik jest kopiowany na stos [`linia 25.`]. Kolejna dekrementacja [`linia 27.`] i wywołanie funkcji [`linia 28.`] – w wyniku czego obliczony zostaje `n-2`-gi wyraz ciągu fibonacciego. wynik znajduje się w eax. Wczytanie poprzedniego wyrazu ciagu ze stosu [`linia 29.`] oraz zsumowanie dwóch poprzednich wyrazów ciągu [`linia 30.`].

$$fibb(n)=fibb(n-1)+fibb(n-2)$$

Dane wyjściowe:

n-ta liczba fibonacciego

Podsumowanie

Przy implementowaniu funkcji rekurencyjnych należy pamiętać o tym, aby gromadzić kolejne informacje o *śladach skoków*.

Korzystanie z rejestru bazowego `%EBP` zawierającego informacje o kontekście wywołania funkcji pomaga w odnalezieniu (indeksowaniu) argumentów funkcji.

Wiedza zdobyta dzięki laboratorium pozwala zrozumieć jak to możliwe, że pomimo tego, że w jęzku C++ nie ma możliwości odczytania argumentu dowolnego numeru, to dzięki skorzystaniu z Assemblera można zaimplementować np. funkcję z biblioteki standardowej wejścia/wyjścia `<stdio.h> printf()`.