Wrocław 3 marca 2015

Kamil Cała  
209954  
Środa 7:15 TN

Sprawozdanie z laboratorium nr 0

25.02.2015r  
Rok akademicki 2014/2015, Informatyka  
Prowadzący: Mgr. Aleksandra Postawka

# Opis ćwiczenia

Celem zerowego ćwiczenia było zapoznanie się z:

* podstawami poruszania się w środowisku terminala Linuxowego,
* używania programu Make, który automatyzuje proces kompilacji i linkowania programów
* konstrukcją i składnią używanych przez powyższy program plików konfiguracyjnych Makefile
* tworzeniem poprawnych plików źródłowym w języku Assembly
* kompilacją plików źródłowych za pomocą asemblera GNU as
* wywyoływaniem funkcji systemowych (zarówno 32 i 64 bitowych) za pomocą przerwania programowego 0x80 i syscall (jedynie to drugie zostało wykonane w praktyce)
* używaniem debuggera GNU gdb

Założenia ćwiczenia zostały zrealizowane poprzez wykonanie czterech kolejnych zadań:

1. Stworzenie programu hello world w C i kompilacja przy użyciu programu Make oraz kompilatora gdb
2. Stworzenie programu hello world w ASM przy użyciu syscall
3. Stworzenie programu w ASM, który pobiera ciąg znaków ze standardowego wejścia i wypisuje je na standardowym wyjściu
4. Modyfikacja programu z punktu 3 - ciąg znaków jest teraz modyfikowany. Wielkie litery zamieniane są w małe, a małe w wielkie.

# 

# 

# Przebieg ćwiczenia

## Hello World w C

Na początku utworzyłem prosty program “Hello world” w C (z racji na niewielki rozmiar programów w tym ćwieczeniu, wklejam ich pełny kod do sprawozdania)

#include <stdio.h>

int main(){

printf("Hello, world!\n");

return 0;

}

Następnie utworzyłem plik Makefile. Zgodnie z nim, program został skompilowany (za pomocą komendy make) do pośredniego pliku hello.o a następnie poddany linkowaniu, czego efektem jest ostateczny plik wykonywalny.

hello: hello.o

gcc -o hello hello.o

hello.o: hello.c

gcc -c hello.c

Program zadziałał zgodnie z oczekiwaniami:

$ ./hello

Hello, world!

## Hello World w ASM

Następnym krokiem było stworzenie takiego samego programu, lecz tym razem już w języku Assembly dla platformy Linux/x64:

Plik źródłowy zaczyna się od zadeklarowania stałych (nazw symbolicznych) dla wybranych funkcji systemowych i ich argumentów. Należy zrobić to w sekcji .data. Potrzebne numery funkcji odnalazłem w pliku unistd\_64.h

.data

SYSREAD = 0

SYSWRITE = 1

SYSEXIT = 60

STDOUT = 1

STDIN = 0

EXIT\_SUCCESS = 0

Następnie zadeklarowana jest stała message typu .ascii

message: .ascii "Hello, world\n"

message\_len = .-message

Do wypisania wiadomości na standardowym wyjściu użyta zostanie funkcja systemowa syswrite, która jako jeden z argumentów, wymaga długości ciągu znaków. Aby uniknąć ręcznego liczenia długośći naszej wiadomości, używam prostego wyrażenia które zostanie policzone w trakcie kompilacji programu. W tym wyrażeniu ‘.’ oznacza bieżącą pozycję w sekcji .text. Obliczana jest więc różnica pomiędzy tą pozycją a warością etykiety message. Obliczenie to musi wystąpić natychmiast po inicjalizacji stałej, gdyż w przeciwnym wypadku może dać nieprawidłowe wyniki.

Kolejnym krokiem jest wywołanie funkcji systemowej SYSWRITE. Odpowiednie wartości muszą zostać wpisane do odpowiednich rejestrów procesora:

.text ; rozpoczęcie sekcji kodu programu

.globl \_start ; wskazanie punktu wejścia programu

\_start:

movq $SYSWRITE, %rax ; funkcja przeznaczona do wywołania

movq $STDOUT, %rdi ; 1 argument - systemowy deskryptor standarodwego wyjścia

movq $message, %rsi ; 2 argument - adres początkowy wiadomości

movq $message\_len, %rdx ; 3 argument - długość łańcuchu znaków

syscall ; wyowłanie funkcji systemowej

Jak można wywnioskować na podstawie powyższego koduy, wywołanie funkcji systemowej na platformie Linux/x64 polega na umieszczeniu numeru odpowiedniej funkcji w rejestrze **rax,** jej argumentów w rejestrach **rdi, rsi, rdx** itd, oraz wykonanie polecenia syscall.

Ostatnią rzeczą jaką należy wykonać jest zakończenie programu poprzez wywołanie funkcji systyemowej SYSEXIT z argumentem EXIT\_SUCCESS (w innych przypadkach można zwrócić odpowiedni kod błędu). Operację tą wykonuje się analogicznie:

movq $SYSEXIT, %rax ; funkcja przeznaczona do wywołania

movq $EXIT\_SUCCESS, %rdi ; argument funkcji

syscall ; wywołanie funkcji systemowej

## Program pobierający ciąg znaków i wypisujący go na wyjście standardowe

W tym programie ponad to, co zostało zrobione w “Hello World”, trzeba dodać kilka nowych elementów. Pierwszym z nich jest zdeklarowanie bufora tekstowego w sekcji .bss programu. W tym wypadku bufor będzie miał rozmiar 512 bajtów.

.bss

.comm user\_input, 512

Kolejnym krokiem jest wczytanie ciągu znaków z wejścia standardowego za pomocą funkcji systemowej SYSREAD:

.text ; rozpoczęcie sekcji kodu programu

.globl \_start ; wskazanie punktu wejścia do programu

\_start:

movq $SYSREAD, %rax ; funkcja przeznaczona do wywołania

movq $STDIN, %rdi ; 1 argument - systemowy deskryptor wejścia standardowego

movq $user\_input, %rsi ; 2 argument - adres początkowy bufora

movq $512, %rdx ; 3 argument - długość czytanego ciągu znaków

syscall ; wywołanie funkcji systemowej

Następnie należy wypisać zapisany w buforze ciąg znaków na wyjście standardowe:

movq $SYSWRITE, %rax

movq $STDOUT, %rdi

movq $user\_input, %rsi ; argument funkcji będący adresem początkowym bufora

movq $512, %rdx ; maksymalna długość bufora

syscall

Wyjście z programu następuje w ten sam sposób co wcześniej.

## Program modyfikujący ciąg znaków

Ostatnim etapem ćwiczenia była modyfikacja poprzedniego programu tak, aby pomiędzy pobraniem ciągu znaków i jego wypisaniem, modyfikował go w ten sposób, że małe litery zostały zamienione w wielkie i na odwrót. W tym celu dodałem w środku programu następującą pętlę:

movq %rax, %rdx #save user's input lenght to rdx

dec %rdx #we have to decrease lenght by 1, so we skip '\n'

movq $0, %rbx #rbx will be our counter - let's initialize it wth 0

\_startLoop: #The loop

mov user\_input(,%rbx,1), %rax #move current letter to rax

xor $0x20, %rax #change letter case with xor binary operation

mov %rax, user\_input(,%rbx,1) #take the letter back to it's place in buffer

inc %rbx #increase counter

cmp %rdx, %rbx #check ending condition

jl \_startLoop #if not met - repeat loop

Pętla pobiera po kolei każdy kolejny znak z ciągu do rejestru, przeprowadza na nim operację XOR, po czym zwraca go do buforu. “Trik” z operacją XOR działa, gdyż w kodowaniu ASCII litery wielkie znajdują się w odległości równej 32 od liter małych (czyli ‘a’-’A’ == 32). Poprzez operację XOR z liczbą 0x20\_16 możemy “przerzucić” bit odpowiadający wartośći 32, zmieniając tym samym wielkość litery. Ten sam efekt można by uzyskać poprzez użycie instrukcji warunkowych i dodanie lub odjęcie 32 od wartości znaku w zależności od jego wielkości. Uczyniłoby to program bardziej przejrzystym ale też dłuższym. Minusem użycia operacji XOR jest to, iż spacja która ma w ANSII wartość 32 zostaje zmieniona w 0, przez co wszelkie spacje zostają usunięte z ciągu znaków. Można to oczywiśćie naprawić odpowiednią instrukcją skoku warunkowego.

## GNU Debugger

W celu przeprowadzenia analizy działania programu można uruchomić go przy użyciu debuggera gdb. Pozwala to na efektywniejsze wykrywanie błędów w programie i lepsze zrozumienie jego działania.

Aby to zrobić należy:

1. W kodzie programu po etykiecie .globl \_start dodać nic nie robiącą instrukcję nop. Jest to sposób na obejście bugu debuggera gdb występującego w przypadku programów 64-bitowych. Bez jego użycia, niemożliwe jest późniejesze ustawienie breakpointów.  
     
   .text  
   .globl \_start  
   \_start:  
   nop
2. Skompilować program z flagą -gstabs  
   as -gstabs -o zad3.o zad3.s

Teraz można uruchomić program przez gdb

$gdb zad3

Podstawowe operacje które można wykonywać to:

b - przejście do zadanego breakpointu

n - ruch o jeden krok

print - wypisanie zawartośći zmiennej

info - z odpowiednim atrybutem (np. info registers - wypisanie na ekranie zawartosći wszystkich rejestrów)

# Podsumowanie

Wykonianie powyższych ćwiczeń pozwoliło na opanowanie umiejętności tworzenia i uruchamiania programów asemblerowych, oraz podstaw obsługi debuggera gdb. Na tej podstawie można przejść do wykonywania kolejnych, bardziej rozbudowanych i skomplikowanych ćwiczeń leboratoryjnych.