Wojciech Adles, 209853 Wrocław, 12.04.17r

ŚR-N-15

Laboratorium Organizacji i Architektury Komputerów

Laboratorium 2:

Utrwalenie umiejętności tworzenia prostych konstrukcji programowych

1. Treść ćwiczenia

- Wczytać dwie liczby z plików w reprezentacji ósemkowej (0,5 pkt)
- Zamienić je na poprawnie zapisane w pamięci (2 pkt)
- Dodać je do siebie z użyciem flagi CF i rejestrów 8 bitowych (1 pkt)
- Zapisać wynik do pliku w reprezentacji heksadecymalnej jako znaki ASCII (1,5 pkt)

2. Przebieg ćwiczenia

2.1 Sekcja .data

Pierwszym krokiem było utworzenie szkieletu programu, z podziałem na sekcję z danymi, sekcję buforów oraz właściwy kod programu. Poniżej znajduje się pierwsza z nich.

```
.section .data

SYSREAD = 0
SYSWRITE = 1
SYSOPEN = 2
SYSCLOSE = 3
SYSEXIT = 60

filename:
        .asciz "number.txt"
filename2:
        .asciz "number2.txt"
filename3:
        .asciz "sumhex.txt"
```

Zdefiniowane tutaj zostały stałe potrzebne przy wywołaniach systemowych oraz trzy zmienne tekstowe zakończone znakiem zero, które zawierają nazwy potrzebnych do wykonania ćwiczenia plików.

2.2 Sekcja .bss

Poniżej, w sekcji .bss zostały zdefiniowane bufory których będziemy używać przy przetwarzaniu danych.

```
.section .bss

.comm filehandle, 8
.comm asciibuff, 1024
.comm octal, 400
.comm octal2, 400
.comm octalsum, 400
.comm asciibuffout, 1024
```

Są to kolejno od góry:

- filehandle: bufor w którym przechowywany będzie uchwyt do pliku
- asciibuff: bufor do którego skopiujemy liczby z pliku
- octal/octal2: bufory przechowujące przekonwertowane liczby
- octalsum: bufor w którym zapiszemy sumę dwóch wczytanych liczb
- asciibuffout: bufor z sumą zapisaną w formacie 0x znakami ASCII

2.3 Wczytywanie i konwersja liczb

Przechodzimy teraz do sekcji zawierającej właściwy kod programu. Na początku za pomocą wywołań systemowych otwieramy plik z pierwszą liczbą, czytamy z niego i zamykamy, w wyniku czego jest ona zapisana do bufora *asciibuff*. Ponadto po wczytaniu liczby z pliku kopiujemy zawartość rejestru *%rax* do rejestru *%rbx*, tak aby zachować liczbę wczytanych bajtów która będzie nam później potrzebna podczas konwersji z formatu ASCII do reprezentacji ósemkowej. Poniżej znajduje się dany fragment kodu.

```
read_file:
        movq $SYSREAD, %rax
        movq filehandle, %rdi
        movq $asciibuff, %rsi          #Destination of read bytes
        movq $1024, %rdx               #Number of bytes to read
        syscall
        movq %rax, %rbx                #Save actual number of bytes read
```

Następnie przechodzimy do jednej z głównych trudności tego laboratorium tj. zapisu wczytanych liczb do pamięci w poprawnej formie.

W architekturze x86 liczby są zapisywane w konwencji *Little endian* w której zapisuje się kolejne bajty danej liczby od najmniej znaczącego do najbardziej znaczącego. Główna trudność przy takim sposobie zapisu w tym zadaniu polega na tym, że w systemie ósemkowym na każdą cyfrę przypadają tylko 3 bity. Wynika z tego, że w bajcie danych mieści się dokładnie 2 i 2/3 cyfry, przez co jedna z nich jest podzielona pomiędzy dwa bajty. Wymusza to konwersję „po trzy” bajty, co wynika z własności, że na 3 bajtach możemy zapisać 8 liczb w reprezentacji ósemkowej (8 liczb trzybitowych = 24 bity czyli 3 bajty). Poniżej znajduje się fragment programu odpowiedzialny za konwersję.

```
ascii_to_octal:
        movb (%rsi, %rbx, 1), %al       #Moving first ASCII sign to %al
        sub $'0', %al                   #Conversion from ASCII to octal
        movb %al, (%r15)
        cmp $0, %rbx
        je after_ascii_to_octal         #If %rbx == 0 exit loop
        dec %rbx
        movb (%rsi, %rbx, 1), %al       #First byte of 'octal' -> 00000xxx
        sub $'0', %al
        salb $3, %al                    #Shifting value by 3 bits to the left
        orb %al, (%r15)                 #First byte of 'octal' -> 00yyyxxx
        cmp $0, %rbx
        je after_ascii_to_octal
        dec %rbx
```

Na początku kopiujemy pierwszą liczbę do rejestru *%al*, po czym odejmujemy od niej wartość znaku ASCII '0', dzięki czemu otrzymujemy liczbę w reprezentacji ósemkowej. Jest ona jednak wciąż zapisana na 8 bitach (trzy bity cyfry, pozostałe 5 bitów to zera). Kopiujemy ją do pierwszego bajtu w buforze *octal*, którego adres znajduje się w rejestrze *%r15*. Sprawdzamy czy była to ostatnio cyfra do konwersji, jeśli tak to wychodzimy z pętli. Następnie wczytujemy kolejną cyfrę do rejestru *%al* i konwertujemy ją do systemu ósemkowego. Po tym, przesuwamy ją o trzy bity w lewo aby nie nadpisała liczby znajdującej się już w buforze *octal*. Używając bitowej funkcji OR 'sklejamy' obie liczby tak, że pierwszy bajt pamięci wygląda następująco: 00yyyxxx, gdzie 'x' to bity pierwszej cyfry, a 'y' bity drugiej. Na podobnej zasadzie konwertujemy i wpisujemy pozostałe cyfry aż do zapełnienia 3 bajtów pamięci *octal*, po czym algorytm zapętlamy, sprawdzając po zapisie kolejnych cyfr czy przekonwertowaliśmy już całą liczbę. Następnie proces wczytania liczby z pliku i jej konwersji powtarzamy dla drugiej liczby.

2.4 Dodawanie liczb z wykorzystaniem flagi CF i rejestrów 8 bitowych

Kolejnym krokiem jest dodanie obu liczb do siebie i zapisanie jej w buforze wynikowym *octalsum*. Poniżej znajduje się fragment kod programu, który objaśnię krok po kroku.

```
add_numbers_start:
        clc                             #Clean carry flag
        pushf                           #Push flag register to stack
        movq $400, %rcx                 #Counter of 'octal' buffer bytes
        movq $0, %r15                   #Counter for 'octal' offset

add_numbers:
        popf                            #Pop flag register from stack
        movb octal(, %r15, 1), %al      #Load 1 byte to register
        movb octal2(, %r15, 1), %bl
        adcb %bl, %al
        pushf                           #Push flag register after ADC command
        movb %al, octalsum(, %r15, 1)   #Move summed number to its buffer
        dec %rcx
        inc %r15
        cmp $0, %rcx                    #If %rcx == 0 exit loop
        jg add_numbers
```

Zaczynamy od wyzerowania flagi przeniesienia i odłożenie tak zmodyfikowanego rejestru flag na stos i inicjalizujemy rejestry które posłużą nam jako liczniki. W głównej pętli dodawania pobieramy ze stosu zachowany rejestr flag i kopiujemy po jednym bajcie każdej liczby do 8 bitowych rejestrów *%al* i *%bl*. Dodajemy je do siebie z wykorzystaniem flagi przeniesienia i odkładamy na stos rejestr flag z 'podniesioną' (bądź nie) flagą CF. Kopiujemy dodany bajt do bufora wynikowego *octalsum*, aktualizujemy rejestry licznikowe i sprawdzamy czy nie był to ostatni bajt do dodania. Jeśli nie to zapętlamy algorytm. Po wykonaniu się tej pętli do końca otrzymamy w buforze *octalsum* sumę tych dwóch liczb w reprezentacji ósemkowej.

2.5 Konwersja wyniku do ASCII i zapis jej do pliku wynikowego

W dalszej części programu konwertujemy liczbę w reprezentacji ósemkowej do reprezentacji szesnastkowej, zapisanej w ASCII.

```
octal_to_ascii_hex:
        dec %rcx
        movb octalsum(, %rcx, 1), %al  #Load byte with 2 hex numbers
        movb octalsum(, %rcx, 1), %bl
        sarb $4, %al                   #Shift one of them
        and $0b1111, %al               #AND to erase higher bits
        and $0b1111, %bl
        add $'0', %al                  #Convert from hex to ASCII
        cmp $'9', %al                  #If higher than '9'(A-F number)
        jle num1_less
        add $7, %al
```

Kopiujemy bajt zawierający dwie liczby w notacji 0x do dwóch rejestrów, po czym przesuwamy w jednym z rejestrów ten bajt o 4 bity w prawo, aby otrzymać 4 starsze bity na pozycji młodszych. Następnie zerujemy 4 najstarsze bity w każdym z rejestrów, aby otrzymać w ten sposób na młodszych bitach liczbę heksadecymalną. Konwertujemy jedną z nich do ACSII poprzez dodanie wartości znaku ‘0’. Sprawdzamy czy po konwersji wartość otrzymanego znaku ASCII jest mniejsza bądź równa wartości ‘9’, jeśli tak powtarzamy proces konwersji dla drugiej liczby, jeśli nie dodajemy jeszcze wartość decymalną $7, ponieważ jest to cyfra z zakresu A-F.

Po konwersji zapisujemy bajty do bufora *asciibuffout* w kolejności od bajtu najbardziej znaczącego, dzięki czemu otrzymujemy liczbę ‘zrozumiałą dla ludzi’. Następnie zapisujemy ją do pliku wynikowego, uprzednio do otwierając za pomocą wywołań systemowych. Po zamknięciu pliku program kończy swoje działanie.

3. Wnioski

Podczas laboratorium zaznajomiliśmy się z obsługą plików, konwersji liczb pomiędzy różnymi systemami oraz zasadami zapisu ich w pamięci dla architektury x86. Napisany przeze mnie program działa poprawnie, nie zaimplementowałem pomijania zera dla najbardziej znaczącego bajtu przy konwersji sumy z systemu ósemkowego na szesnastkowy w formacie ASCII, przez co w przypadku gdy pierwsza liczba w najbardziej znaczącym bajcie wynosi zero, w pliku wynikowym otrzymujemy liczbę postaci 0A123BDEF.