

## Przekazywanie argumentów i wywoływanie funkcji bibliotecznych języka C.

Wykorzystanie funkcji *System Calls* do wczytywania lub wyświetlania wielu danych różnych typów (innych niż łańcuchy tekstowe) jest generalnie kłopotliwe i wymaga dodatkowych procedur konwersji.

Bardziej uniwersalnymi narzędziami, pozwalającymi konwertować i scalać w ciągi tekstowe wiele różnych zmiennych, są np. funkcje *printf* i *scanf*, dołączone do standardowej biblioteki języka C.

Wykorzystanie funkcji bibliotecznych języka C/C++ **wymaga przestrzegania kilku reguł**, m.in.:

- kolejności przekazywania argumentów w odpowiednich rejestrach,
- konieczności zabezpieczania pierwotnych wartości w rejestrach (przed zniszczeniem – nadpisaniem) przez wywoływany podprogram.

Te, jak i wiele innych wytycznych zawartych jest w *Application Binary Interface (ABI)* – tablica 3.4 strona 21:

- pierwsze trzy argumenty całkowitoliczbowe przekazuje się (podobnie jak *System Calls*) w rejestrach *%rdi*, *%rsi* i *%rdx* (o odpowiednim rozmiarze), kolejne – wg w/w tablicy,
- funkcja (napisana np. w języku C) zwraca wartość (całkowitoliczbową) w *%rax* (lub *%eax* – stosownie do typu danych)
- argumenty typów zmiennoprzecinkowych przekazuje się w długich, 128 bitowych rejestrach wektorowych (*%xmm*) – na razie nie będą potrzebne, niemniej jednak:
- **przed wywołaniem funkcji należy w *%eax* przekazać informację o liczbie wykorzystywanych rejestrów wektorowych** (w tym zadaniu – zero). Uwaga: nie jest to konieczne, jeśli wywoływana funkcja nie wykorzystuje tych rejestrów – niemniej bez analizy kodu danej funkcji tego nie wiemy. Dodatkowo sytuacja może zmieniać się w raz kolejnymi wersjami pakietu GCC i dołączonych bibliotek.

**Proszę zwrócić uwagę** na trzecią kolumnę w tablicy 3.4 (*preserved across function calls*).

Ponieważ procesor (pojedynczy rdzeń, pomijamy Hyper Threading itp.) posiada jeden bank z ograniczoną liczbą rejestrów ogólnego przeznaczenia – może się zdarzyć (a w normalnych, złożonych programach jest to reguła), że zarówno funkcja nadrzędna (wywołująca - *caller*) jak i podrzędna (wywoływana - *callee*) będą musiały używać tych samych rejestrów do przechowywania swoich danych.

Aby uchronić wartości rejestrów funkcji nadrzędnej przed nadpisaniem przez f. podrzędną, *ABI* definiuje która z funkcji ma obowiązek zabezpieczyć wartości w poszczególnych rejestrach.

Np. pisząc funkcję *main* nie trzeba się martwić czy podczas wywołania *printf* wartość *%rbx* zostanie zniszczona (*printf* ma obowiązek ją zabezpieczyć np. odkładając na stos, w przeciwieństwie do np. *%rcx* – który zostanie nadpisany).

## Zadanie 1.

W pliku *nwd.s* uzupełnić funkcję obliczającą Największy Wspólny Dzielnik (*NWD*), do której zarówno argumenty, jak i zwracana wartość będą przekazywane w rejestrach zgodnych z *ABI*. Np. prosty algorytm Euklidesa ma postać:

```
unsigned int NWD(unsigned int a , unsigned int b)
(czyli a w %edi, b w %esi, obliczony NWD w %eax)

while (a!=b)
{
    if (a>b) a=a-b;
    else b=b-a;
}
return a
```

lub w wersji rekurencyjnej:

```
unsigned int NWD(unsigned int a, unsigned int b)
{
    if (b==0) return a;
    else NWD(b, a % b);
}
```

Wybrany algorytm należy napisać w asemblerze. Pierwszą wersję można implementować na wiele sposobów, np. wykorzystując instrukcje wykonywane warunkowo (*cmov*), w przypadku wariantu rekurencyjnego – obliczanie modulo zostało wytłumaczone w instrukcji do ćwiczenia 3. i 4.

## Zadanie 2.

Przekazać brakujące argumenty do funkcji *printf* i wyświetlić wynik w postaci:

```
printf("NWD(%u,%u)=%u", a, b, c);
```

Proszę zwrócić uwagę na kolejność argumentów przekazanych do *printf* (podobna zasada obowiązuje dla wszystkich funkcji w języku C):

pierwszy argument to adres ciągu tekstowego - w *%rdi* (*%edi*),  
kolejne argumenty – **od lewej do prawej**: *a* w *%esi*, *b* i *c* – w kolejnych rejestrach zgodnie z *ABI*.

**Pamiętać o przekazaniu liczby użytych rejestrów wektorowych w *%eax*** (tu ich nie wykorzystujemy, więc 0).

**Ciągi tekstowe w C muszą kończyć się bajtem zerowym (NULL)** - podajemy jako argument tylko wskaźnik na początek ciągu, nie trzeba liczyć i przekazywać jego długości – funkcje w C czytają ciąg od jego adresu początkowego, do napotkania znaku zerowego.

Allokacja ciągów zgodnych z C (różne sposoby):

```
napis: .ascii      "tekst\n\0"
napis: .asciz      "tekst\n"
napis: .string     "tekst\n"
```

zdefiniowanie ciągu jako *.string* (*.asciz*) powoduje, że kompilator sam „dołoży” znak zerowy na końcu.

Proszę zwrócić uwagę, że główna funkcja w programie teraz ma etykietę *main*, tak jak w C (nie *\_start*).

**Do linkowania należy użyć *gcc*** (składnia taka sama jak w przypadku *ld*, którego *gcc* wywołuje pośrednio).

*Gcc* automatycznie dołącza funkcje inicjujące i kończące program (np. *\_start*, *libc\_start\_main* – „normalnie” niewidoczne dla programisty), nie ma więc potrzeby wywoływania funkcji systemowej *EXIT* – wystarczy zwrot numeru błędu w *%eax* i powrót z *main* instrukcją *ret*.

### Uwaga:

Wykonywanie zadań 1 i 2 najlepiej zacząć od skompilowania, zlinkowania i uruchomienia „pustego” pliku *nwd.s*. Później, mając działające „wyświetlanie” dopisać funkcję *NWD* i pozostałe elementy programu.

Jeśli podczas linkowania nowszymi wersjami gcc (np. 7.3 w górę) pojawią się błędy związane z tworzeniem kodu typu *Position Independent Code/Executable (PIC/PIE)* – czyli takiego, który nie wykorzystuje adresów absolutnych komórek pamięci i może być ładowany, np. dynamicznie, w (prawie) dowolne miejsce w pamięci:

*relocation R\_X86\_64\_32S against `.data' can not be used when making a PIE object; recompile with -fPIC*

należy dodać przy linkowaniu parametr: **-no-pie** (różne wersje gcc mają różne ustawienia domyślne).

Jeśli plik wykonywalny się utworzy, a podczas jego uruchomienia wystąpi *segmentation fault* (naruszenie ochrony pamięci) – odkomentować pierwszą i przedostatnią instrukcję w funkcji *main* (przesunięcie wierzchołka stosu *%rsp* o 8 bajtów odpowiednio: sub w dół, add - w górę).

Informacja dodatkowa:

Wg standardu (*ABI*, tabela 3.3) po „wejściu” do podprogramu/funkcji wierzchołek stosu musi być wyrównany do granicy 16 bajtów (8 na adres powrotny + 8 na ramkę stosu). Ponieważ ramki stosu (w *%rbp*) nie używamy – stos po wywołaniu *main* jest wyrównany tylko do 8 bajtów. W zależności od wersji gcc i wersji dostarczonych bibliotek – wymóg ten, w zależności od funkcji, przestrzegany jest mniej lub bardziej restrykcyjnie.

Na razie ramką stosu się nie martwić...

### Zadanie 3.

Argumenty funkcji *NWD* przekazać jako parametry podczas wywoływania programu (z linii komend terminala) np.:

```
./program_nwd 40 50
```

Parametry z linii komend do funkcji *main* przekazywane są tak samo jak do każdej innej funkcji w C:

```
int main(int argc, char **argv)
```

*argc* (w *%edi*) – liczba parametrów ( $\geq 1$ ) Pierwszym parametrem, zawsze przekazywanym, jest nazwa programu!

*argv* (w *%rsi*) – adres do tablicy, której **ośmiobajtowe** elementy zawierają adresy kolejnych parametrów (przekazywanych z linii komend jako ciągi tekstowe!)

np. dla powyższego wywołania programu, wartość w *%rsi* wskazuje na początek trójelementowej tablicy:

*%rsi+0* -> element 0 - adres parametru 1 (np. „program\_nwd”)

*%rsi+8* -> element 1 - adres parametru 2 (40 – jako string!)

*%rsi+16* -> element 2 - adres parametru 3 (50 – jako string!)

dostęp do poszczególnych elementów tablicy wyjaśniony został w instrukcji do laboratoriów 3 i 4.

Np. zapis trzeciego parametru do *%rax* może mieć postać: `mov 16(%rsi), %rax`

**Uwaga: każdy z pobranych z pamięci parametrów (jako ciąg tekstowy) musi zostać przekształcony na liczbę,** np. (prostą...) funkcją *atoi*:

```
int atoi(&string);
```

Podczas dwukrotnego wywołania funkcji *atoi* należy pamiętać, że może ona nadpisać wartości części rejestrów (zgodnie z *ABI*).

W programie powinna zostać uwzględniona (przynajmniej minimalna) obsługa błędów. Np. po wywołaniu programu bez parametrów, bądź z podaniem nieprawidłowej ich liczby – powinno nastąpić wyjście poprzedzone stosownym komunikatem.