

## Laboratorium 6. i 7. Biblioteki statyczne i dynamiczne współdzielone, stos

### Zadanie 1. Zbudować bibliotekę statyczną, której funkcje będą wywoływane z poziomu języka C.

1. Plik źródłowy *lib\_asm.s* zawiera dwie funkcje (bezargumentowe): *print\_call* i *print\_ret* wyświetlające:

- zawartość licznika rozkazów (*%rip*) w miejscu załadowania danej funkcji (adres pierwszej instrukcji),
- aktualne położenie wskaźnika stosu (*%rsp*),

oraz numer kolejnego (odpowiednio: odłożonego i ściąganego) adresu powrotnego.

Trzecią funkcję, obliczającą największy wspólny dzielnik (*Greatest Common Divisor, GCD*), trzeba dopisać w pliku *lib\_gcd.s*, w asemblerze, przekazując argumenty i chroniąc odpowiednie rejestry przed nadpisaniem w sposób kompatybilny z językiem C (zgodnie z ABI). Główną etykietę i typ funkcji zadeklarować należy analogicznie do tych w pliku *lib\_asm.s*.

Np. prosty algorytm Euklidesa ma postać:

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

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

lub w wersji rekurencyjnej:

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

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.

**Główny program** (*main*), pobierający argumenty i wyświetlający GCD należy napisać w języku C.

Sposób wyświetlania wyniku, przekazywania argumentów i sprawdzania ich poprawności jest dowolny: linia komend, funkcja *scanf* itp.

## Tworzenie biblioteki statycznej.

W przypadku jednego pliku źródłowego, zawierającego wszystkie funkcje biblioteczne postępowanie nie różni się od kompilacji zwykłego programu np.:

*libr.s* – biblioteka funkcji            *prog.c* – program główny (*main*)

- kompilujemy wszystkie pliki źródłowe: tworzymy obiekty – bloki kodu maszynowego (*.o*)

```
gcc -c prog.c
```

```
as libr.s -o libr.o
```

- następnie standardowo linkujemy:

```
gcc prog.o libr.o -o prog_stat
```

Gdy plików źródłowych jest wiele (np. każdy zawiera jedną funkcję biblioteczną) można postąpić podobnie, lub użyć archiwizatora *ar* i połączyć wszystkie pliki *.o* w jedno archiwum (*.a*):

*fun1.s* – biblioteka z funkcją 1   *fun2.s* – biblioteka z funkcją 2...   *prog.c* – program główny (*main*)

```
as fun1.s -o fun1.o
```

```
as fun2.s -o fun2.o
```

```
ar -r libfun.a fun1.o fun2.o
```

```
gcc -c prog.c
```

```
gcc -s prog.o libfun.a -o prog_stat
```

lub:

```
gcc -s prog.o -L. -lfun -o prog_stat
```

gdzie: opcjonalny przełącznik *-s* powoduje usunięcie z pliku wykonywalnego części niepotrzebnych przy linkowaniu statycznym np. tablic relokacji i symboli (w rezultacie plik wykonywalny jest mniejszy). Kropka po „*-L*” jest w ogólnym przypadku ścieżką dostępu do biblioteki. Należy zwrócić uwagę na nazwę archiwum i linkowanej biblioteki (z parametrem *-l*).

Gdyby podczas linkowania utworzonej statycznej biblioteki pojawił się błąd: *relocation x against `data' can not be used when making a PIE object...* należy dodać parametr *-no-pie* (zależne od wersji i ustawień pakietu *gcc*).

## Kolejne etapy zadania:

**2.** Uruchomić **kilka razy** program. Sprawdzić poprawność działania: wynik obliczeń, zaobserwować zachowanie licznika rozkazów oraz wierzchołka stosu.

**3.** Dopisać w funkcji *GCD* wyświetlanie położenia wierzchołka stosu po rozpoczęciu, jak i przed powrotem, np.:

```
gcd:
#zachowaj na stosie co trzeba
call print_call_rsp
#ściągnij ze stosu

...
#zachowaj na stosie co trzeba
call print_ret_rsp
#ściągnij ze stosu
ret
```

oraz, opcjonalnie, dodać wyświetlanie dwóch argumentów do niej przekazanych (zaraz po jej rozpoczęciu). Pilnować nadpisania wartości w rejestrach przez kolejne wywoływane funkcje i w razie konieczności zabezpieczać je na stosie!

4. Powtórzyć p. 2. i 3. dla różnych argumentów funkcji GCD. Sprawdzić liczbę iteracji, zachowanie licznika wywołań/powrotów (*cntr*).

5. Skompilować program z **wszystkimi** bibliotekami / funkcjami dołączonymi statycznie (czyli również *printf/scanf/atoi* itp.): dodać przełącznik *--static* podczas linkowania *gcc*. Sprawdzić działania i porównać rozmiar pliku wykonywalnego z poprzednimi wersjami.

## Zadanie 2. Biblioteka współdzielona - ładowana dynamicznie.

1. Zmienić tryb adresowania w funkcjach *print\_call/ret\_rsp* z na względny (*%rip* – opis poniżej), a następnie zbudować bibliotekę dynamiczną współdzieloną.

2. Tworzenie biblioteki współdzielonej, ładowanej dynamicznie

Mając już skompilowane pliki z programem głównym i funkcjami bibliotecznymi (jak w poprzednim przykładzie: jeden *libr.o*, bądź oddzielne: *fun1.o*, *fun2.o*... itp.) należy zbudować z nich bibliotekę - obiekt współdzielony:

```
gcc -shared libr.o -o libfun.so
```

```
gcc -shared fun1.o fun2.o -o libfun.so
```

Jeżeli wystąpi błąd: *relocation x against '.data' can not be used when making a shared object* trzeba sprawdzić, czy w dostępie do sekcji *.data* nie są używane adresy absolutne. Jeśli wystąpi „*against undefined symbol 'funkcja@GLIBC'*” – sprawdzić, czy funkcje języka C są wywoływane pośrednio – przez tablicę PLT (*@plt*).

Następnie, utworzyć plik wykonywalny korzystający z tej biblioteki:

```
gcc prog.o libfun.so -o prog_dyn
```

lub:

```
gcc prog.o -L. -lfun -o prog_dyn
```

3. Uruchomić **kilka razy** program (również bez podawania ścieżki dostępu do biblioteki). Sprawdzić jego zachowanie w każdym z przypadków.

Jeżeli próba uruchomienia (*./prog\_dyn*) zakończy się błędem ładowania biblioteki, można program uruchomić wskazując linkerowi ścieżkę dostępu do katalogu roboczego:

```
LD_LIBRARY_PATH=. ./prog_dyn      (ew. wykorzystując polecenia export i unset).
```

4. Zmienić argumenty funkcji *GCD* i powtórzyć od p. 3.

## Uwagi:

1. Każda funkcja/etykieta w bibliotece asemblerowej, która ma być dostępna „z zewnątrz” – dla innych programów, musi być zdefiniowana jako globalna (`.globl` lub `.global`). Należy również linkerowi określić, który symbol-etykieta faktycznie oznacza początek funkcji:

```
.type etykieta, @function
```

2. Standard (ABI, tabela 3.3) wymusza wyrównanie wierzchołka stosu po „wejściu” do funkcji do granicy 16 bajtów (8 na adres powrotny + 8 na ramkę stosu, adres ma być podzielny przez 16). Ponieważ ramki stosu (`%rbp`) w tym ćwiczeniu nie używamy – stos po wywołaniu funkcji jest wyrównany tylko do 8 bajtów. W przypadku, gdy w danym podprogramie są zagnieżdżone wywołania bibliotek języka C, w celu zapewnienia zgodności ze standardem, po wywołaniu funkcji należy dodać stosowny *padding*, a przed powrotem go usunąć.

W zależności od wersji `gcc` i wersji dostarczonych bibliotek – wymóg ten, w zależności od funkcji, przestrzegany jest mniej lub bardziej restrykcyjnie.

3. Kod funkcji dynamicznie ładowanych musi być napisany w sposób umożliwiający jego poprawne wykonanie w (prawie...) dowolnym miejscu w pamięci (tzw. *PIC* - *Position Independent Code*). W tym celu nie należy używać stałych adresów absolutnych (bezpośrednio numerów komórek pamięci) np. w dostępie do sekcji `.data`, a zamiast nich stosować dostępne w *x86-64* adresowanie względne licznikiem rozkazów `%rip` (u Intel'a nazywa się on *Instruction Pointer* - *IP*).

Przykład adresowania absolutnego:

symboliczny argument-etykieta w instrukcji:

```
mov $napis,%rdi (lea napis,%rdi)
```

zostanie zamieniony przez linker na:

```
.data
0x00601038 napis:      .string "tekst"

.text
0x00400660   mov    $0x00601038,%rdi    # %rdi=adres pierwszego elementu ciągu (absolutny)
0x00400667   xor    %eax,%eax
```

Takie samo ładowanie początkowego adresu ciągu tekstowego, ale z wykorzystaniem adresowania względnego i licznika rozkazów można wykonać instrukcją:

```
lea    napis(%rip),%rdi    #lea = load effective address
```

- przesunięcie między sekcjami `.data` i `.text` danej biblioteki **jest zawsze stałe** i nie zależy od fizycznego położenia biblioteki/programu w pamięci. Linker oblicza je automatycznie (tutaj `0x002009D1`). W programie źródłowym w argumencie instrukcji (w miejscu stałej – przesunięcia) podajemy etykietę!

- `%rip` wskazuje adres kolejnej instrukcji do wykonania (*xor*) – i to on **jest zależny** od miejsca załadowania programu w pamięci.

```
0x00400660   lea    0x002009D1(%rip),%rdi    #rdi=0x002009D1+0x00400667=0x00601038
0x00400667   xor    %eax,%eax
```

Adresy są przykładowe, typowe dla plików ELF i zostały skrócone do 32 młodszych bitów.

Funkcje ładowane dynamicznie należy wywoływać: `call funkcja@plt`. Mechanizm wywoływania tego typu funkcji został przedstawiony w prezentacji dot. stosu i bibliotek.