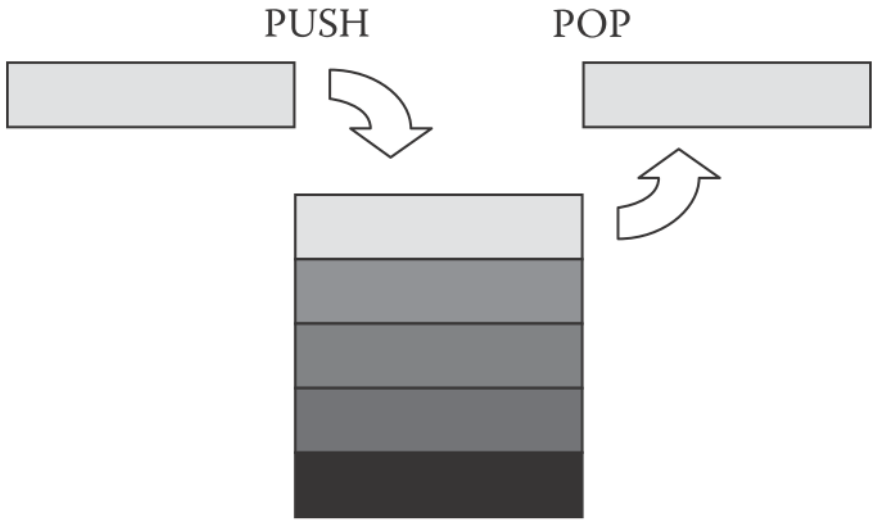
Stos – struktura danych:

- zorganizowana jako kolejka *Last In First Out* (LIFO)
  (ostatni wprowadzony element opuści ją jako pierwszy)
- umieszczona w pamięci RAM komputera
- zapewnia bezpośredni, łatwy dostęp do ostatniego odłożonego elementu, wierzchołka stosu
- adresowana **wskaźnikiem stosu (stack pointer)**

W ogólnym przypadku obsługiwana z reguły dwoma instrukcjami:

push: umieszcza element na stosie

pop/pull: ściąga ostatni element ze stosu



Stos – architektura x86-64

- **stos jest malejący/opadający (descending stack)**
  tzn. „rośnie w dół” – w kierunku malejących adresów (wierzchołek znajduje się najniżej)

- **wskaźnik stosku** (64bit – %rsp) **wskazuje na adres ostatniego odłożonego elementu (tzw. stos „pełny”)**

- w trybie 64 bitowym na stos są odkładane **jedynie** 64 bitowe argumenty (reg/mem/imm)!

Instrukcje push i pop wykonują operacje:

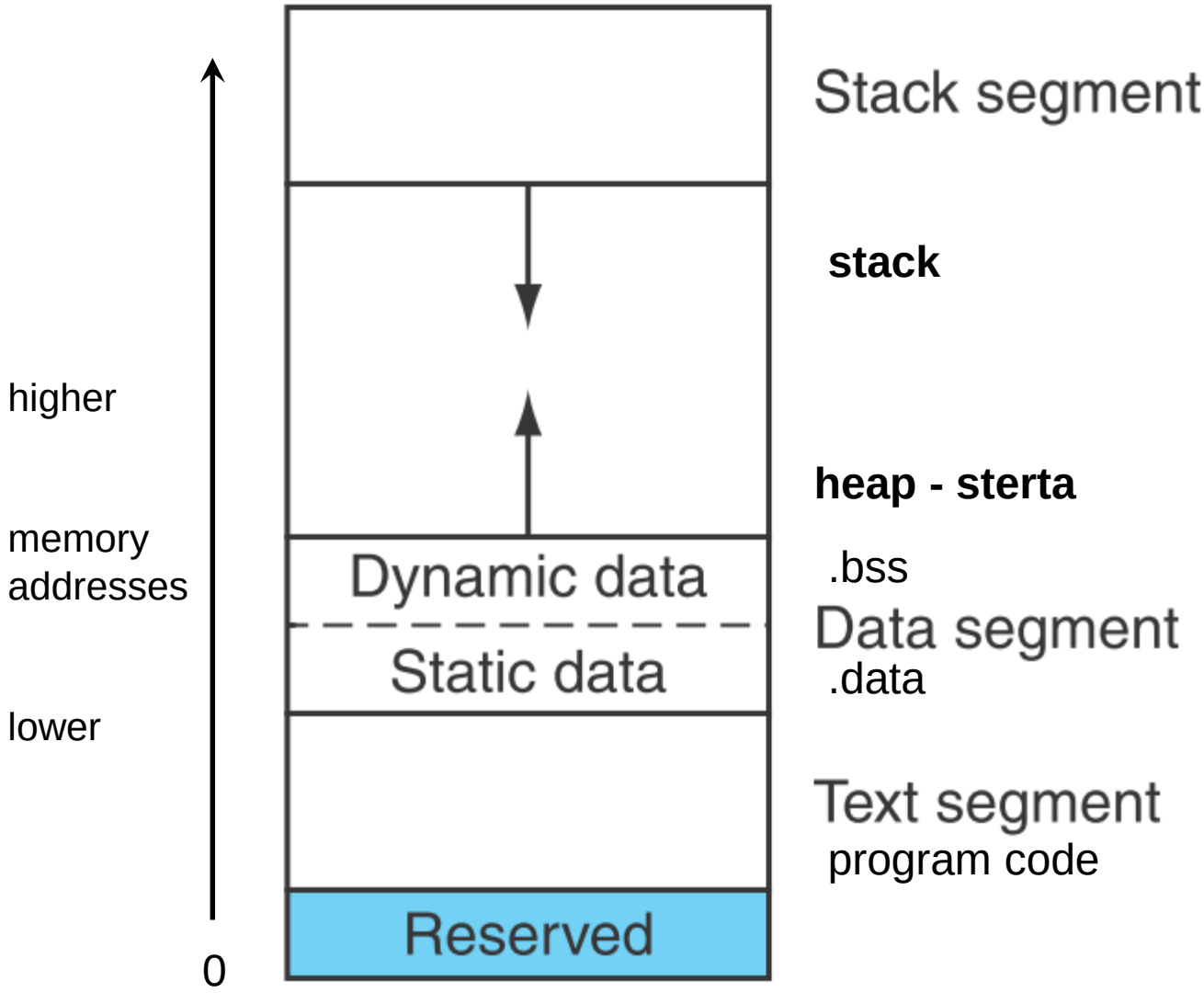
- **push** arg

```
1. sub  $8 , %rsp
2. mov  64bit_arg,(%rsp)
```

- **pop** arg

```
1. mov  (%rsp),%64_bit_register
2. add  $8 , %rsp
```

Mapa pamięci (memory map) programu. Dlaczego stos jest malejący?



Kod (.text) i dane statyczne (.data) – nie zmieniają położenia ani rozmiaru podczas całego „życia” programu. Ich rozmiar jest oszacowany i znany podczas kompilacji i linkowania.
Stos i sterta (pamięć/obiekty alokowane dynamicznie) mogą zmieniać swój rozmiar, są więc umieszczone na przeciwległych końcach przestrzeni adresowej.

Przeznaczenie stosu (niskopoziomowe)

- **register spilling** – zabezpieczanie wartości rejestrów przed ich nadpisaniem w przypadku konieczności ponownego użycia np.

  - braku wolnych rejestrów w danym fragmencie programu

  - podczas wywołania podprogramów

- **zachowywanie adresu powrotnego podczas wywoływania podprogramów***

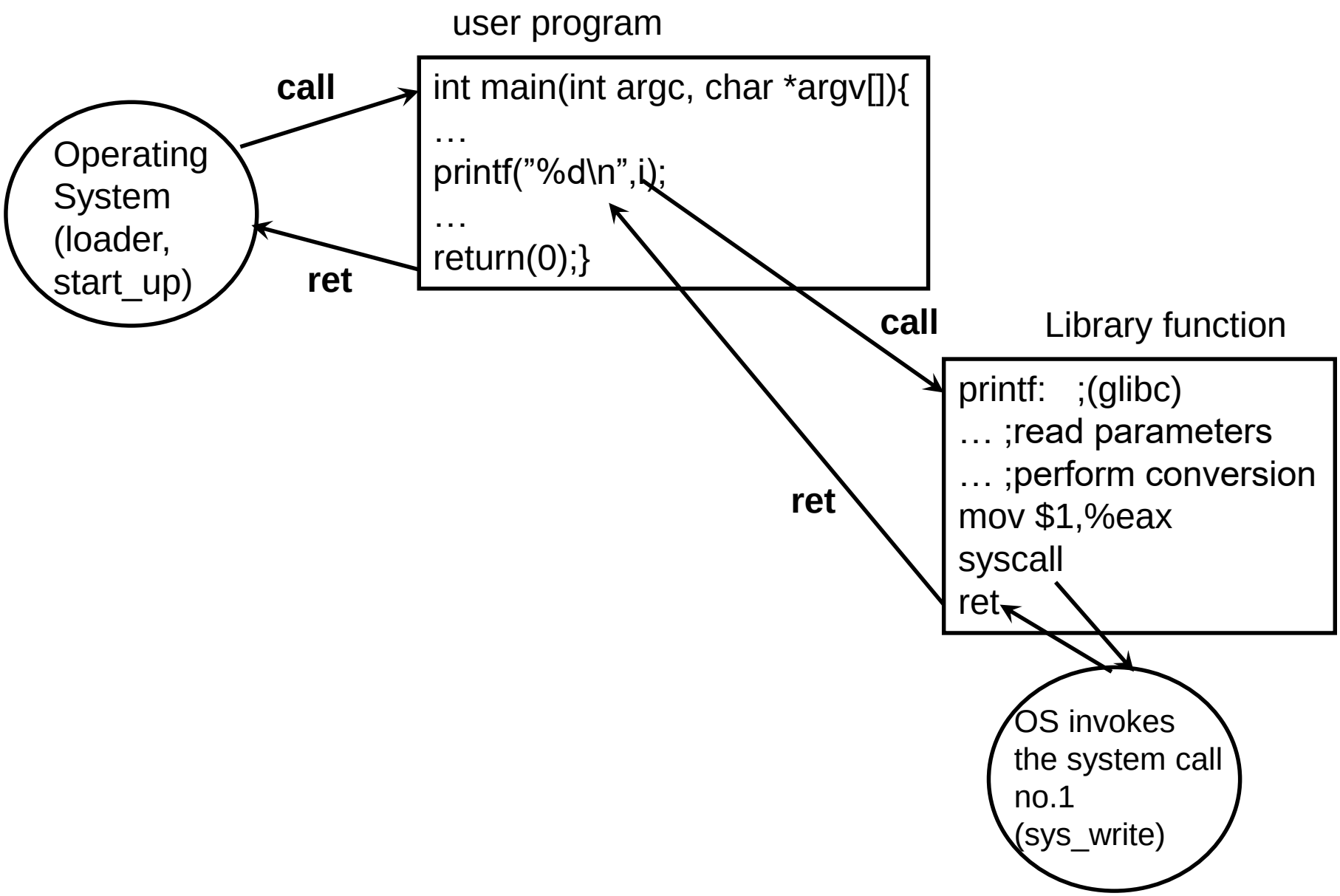
- **przekazywanie parametrów do funkcji**
  (często używane dawniej, np. w systemach 32 bitowych)

- **alokowanie pamięci na zmienne lokalne/automatyczne podprogramów****

* procesory MIPS i ARM zachowują adres powrotu w rejestrze „Link Register”

** nowoczesne kompilatory – jeśli to możliwe – starają się trzymać zmienne lokalne w rejestrach procesora (szybszy dostęp)

Wywoływanie podprogramów (to już było...)



Wywoływanie podprogramów:

- umieszczenie argumentów w miejscu dostępnym dla podprogramu (obecnie rejestry, ale może być stos lub przekazany wskaźnik do miejsca w pamięci)
- skok do podprogramu i zachowanie **adresu powrotu***
- alokacja zasobów (np. pamięci na zmienne lokalne) niezbędnych dla podprogramu
- wykonanie podprogramu
- umieszczenie zwracanej wartości w miejscu do którego ma dostęp funkcja nadrzędna
- **zwolnienie miejsca** po zmiennych lokalnych
- powrót do funkcji nadrzędnej** (ew. zwolnienie miejsca na stosie po argumentach) i jej kontynuacja

Wywoływanie podprogramów

* **adres powrotu** jest to adres **kolejnej** instrukcji do wykonania w przerywanej funkcji (brany z licznika rozkazów np. %rip):

- podczas wywołania podprogramu jest to adres następnej instrukcji po call.

- w przypadku konieczności obsługi wyjątku lub przerwania zewnętrznego o wyższym priorytecie procesor kończy wykonywanie bieżącej operacji i aktualne zadanie jest przerywane. Adres powrotu wskazuje na kolejną instrukcję po ostatnio ukończonej (miejsce kontynuacji przerwanego programu).

** instrukcja ret mechanicznie ściąga ze stosu ostatni element (jak pop) i traktuje go jako adres docelowy skoku. Programista musi zadbać by przed wykonaniem rozkazu ret wskaźnik stosu wskazywał na właściwy element, będący prawidłowym adresem powrotnym.

Przekazywanie parametrów przez stos, alokacja zmiennych lokalnych (przykład 32 bitowy!)

```
int main(void)
{
   …
   subroutine(arg1, arg2, arg3);
   …
   return(0);
}
```

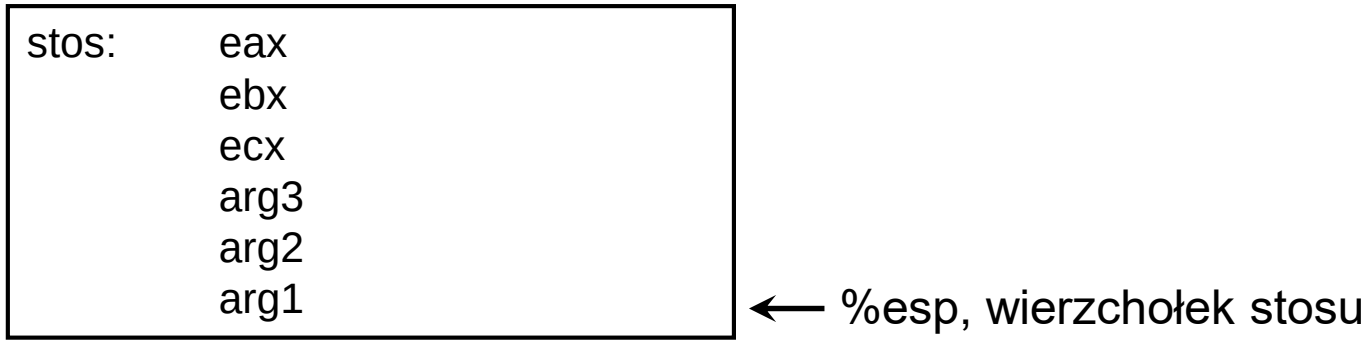
```
void subrouitne(int arg1, int arg2, int arg3)
{ int local_1, local_2;
…
}
```

Założenia:

- Zarówno funkcja główna (main) jak i podprogram (subroutine) wykorzystują rejestry: eax, ebx i ecx do swoich celów.
- Rejestry eax, ebx, i ecx nie są zachowywane przez podprogram (więc musi to zrobić main)
- Podprogram przechowuje dwie zmienne lokalne typu int **na stosie**

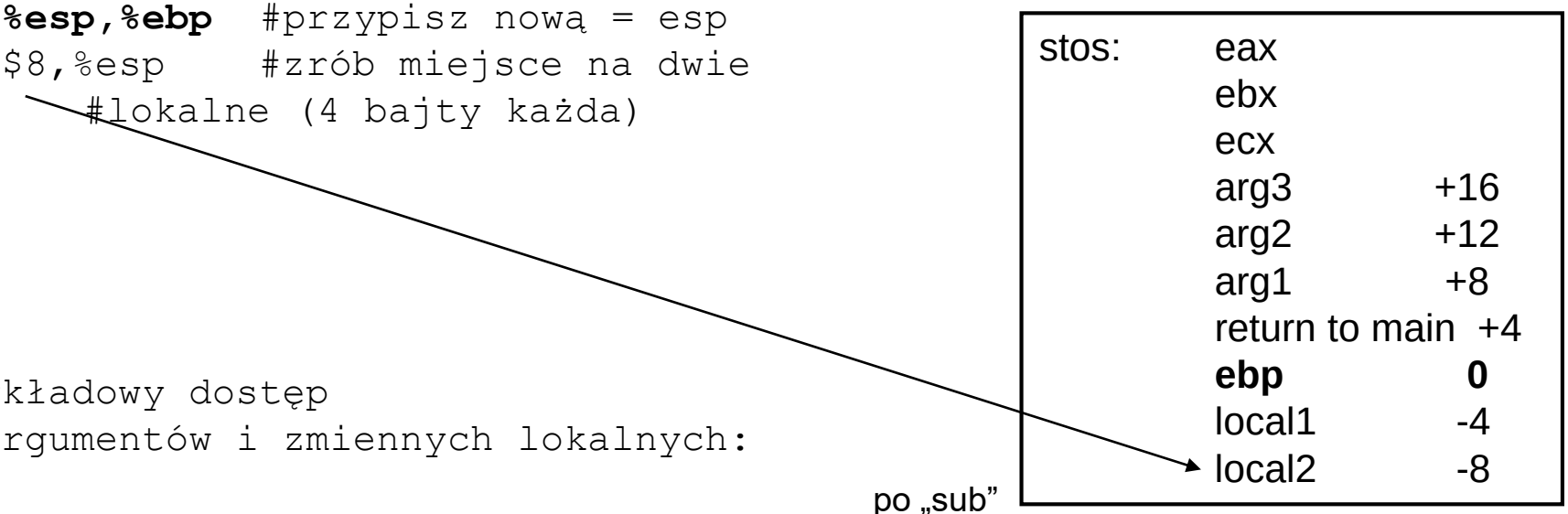
(1) przed wywołaniem funkcji

```
main:
…
push %eax #zachowaj wartośći w rej.
push %ebx #eax, ebx and ecx
push %ecx #używane przez main

push arg3 #umieść na stosie
push arg2 #argumenty
push arg1 #zaczynając od ostatniego

call subroutine
```



(2) po wejściu w podprogram

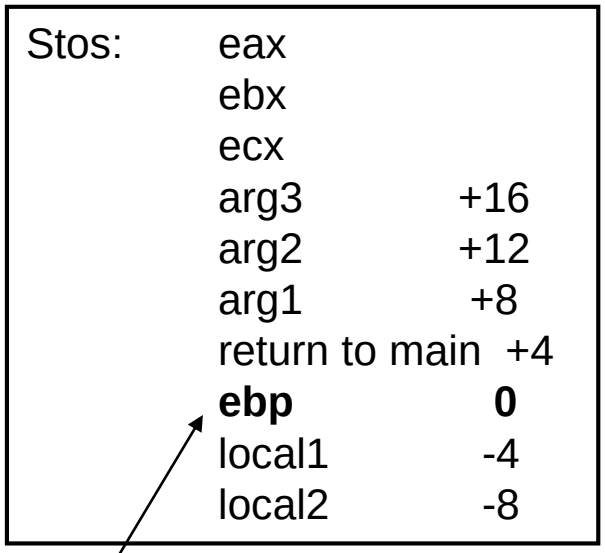
```
subroutine:
push %ebp          #zachowaj starą ramkę stosu (z main)
mov  %esp,%ebp     #przypisz nową = esp
sub  $8,%esp       #zrób miejsce na dwie
…          #lokalne (4 bajty każda)
```




```
#przykładowy dostęp
#do argumentów i zmiennych lokalnych:

mov  8(%ebp),%eax #pobierz arg1
mov 12(%ebp),%ebx #arg2
mov 16(%ebp),%ecx #i arg3 ze stosu
…
mov -4(%ebp),%eax #pobierz local1
mov -8(%ebp),%ebx #pobierz local2

…
mov %ebp,%esp  #zwolnij pamięć po zmiennych
               #lokalnych
pop %ebp       #przywróć mainowi jego ramkę stosu
ret
```

(3) po wejściu w podprogram

```
subroutine:
push %ebp          #zachowaj starą ramkę stosu (z main)
mov  %esp,%ebp     #przypisz nową = esp
sub  $8,%esp       #zrób miejsce na dwie
…          #lokalne (4 bajty każda)
```



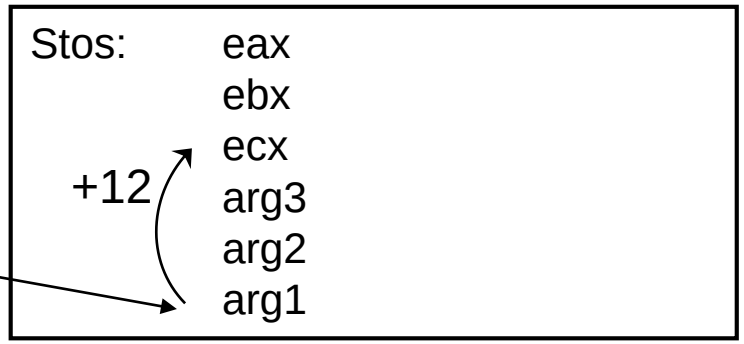

```
#przykładowy
#dostęp do argumentów i zmiennych lokalnych:

mov  8(%ebp),%eax #pobierz arg1
mov 12(%ebp),%ebx #arg2
mov 16(%ebp),%ecx #i arg3 ze stosu
…
mov -4(%ebp),%eax #pobierz local1
mov -8(%ebp),%ebx #pobierz local2

…
mov %ebp,%esp   #zwolnij pamięć po zmiennych
                #lokalnych
pop %ebp        #przywróć mainowi jego ramkę stosu
ret             #teraz %esp wskazuje na adres powrotny
```

czyli: przesuń %esp
(gdziekolwiek by nie był)
na miejsce gdzie jest zachowana
poprzednia wartość %ebp

(4) powrót do main:

```
main:

call subroutine

add $12,%esp    #zwolnij pamięć na
                #stosie po
                #trzech argumentach

pop %ecx        #przywróć oryginalne
pop %ebx        #wartości rejestrom
pop %eax        #eax, ebx i ecx

...
```

po powrocie do main
%esp wskazuje na arg1



• instukcje typu pop, ret, iret nie usuwają fizycznie danych ze stosu, przesuwają jedynie wskaźnik. Dane pozostają w pamięci, aż zostaną nadpisane podczas odkładania kolejnych elementów.

• instrukcje operujące na stosie wykonują zwykły dostęp do pamięci – tym samym spowalniają wykonywanie programów.

Ramka stosu

- **wskaźnik stosu** z reguły **zmienia swoje położenie** w trakcie wykonywania programu, co za tym idzie przesunięcie (liczba bajtów) między jego aktualnym położeniem a danymi na stosie również się zmienia

- utworzona dla każdego* podprogramu, dedykowana **ramka stosu** (zwyczajowo w rejestrze ebp/rbp) zapewnia **stały punkt odniesienia,** pozwalający na łatwiejszy dostęp do umieszczonych na stosie danych (np. zmiennych lokalnych i argumentów): przesunięcie między ebp/rbp a konkretnymi danymi na stosie **jest stałe**.

* w przypadkach prostych programów oraz przechowywania zmiennych lokalnych w rejestrach ramki stosu nie tworzy się (podobnie postępuje również kompilator).

Należy pamiętać, że zgodnie z ABI 64 bit przed wywołaniem funkcji bibliotecznych C wierzchołek stosu musi być wyrównany do granicy 16 bajtów (8 adres powrotu + 8 ramka stosu). Jeżeli ramki stosu nie tworzymy, wskaźnik stosu najlepiej przesunąć szybką operacją arytmetyczną niż odkładać i ściągać jakiś argument np.

sub $8,%rsp na początku podprogramu i add $8,%rsp przed powrotem.

Dzięki temu i standard jest zachowany, i unikamy dwóch dostępów do pamięci.

Typy stosów – cztery warianty

- **malejący/opadający/scending**

  „rośnie w dół” przestrzeni adresowej,
  kolejne elementy odkładane są w komórkach o coraz niższych adresach

- **rosnący/wznoszący/ascending**

  kolejne elementy zapisywane są pod coraz wyższymi adresami
  (wierzchołek stosu ma najwyższy adres)

W każdym z nich wskaźnik stosu może wskazywać na:

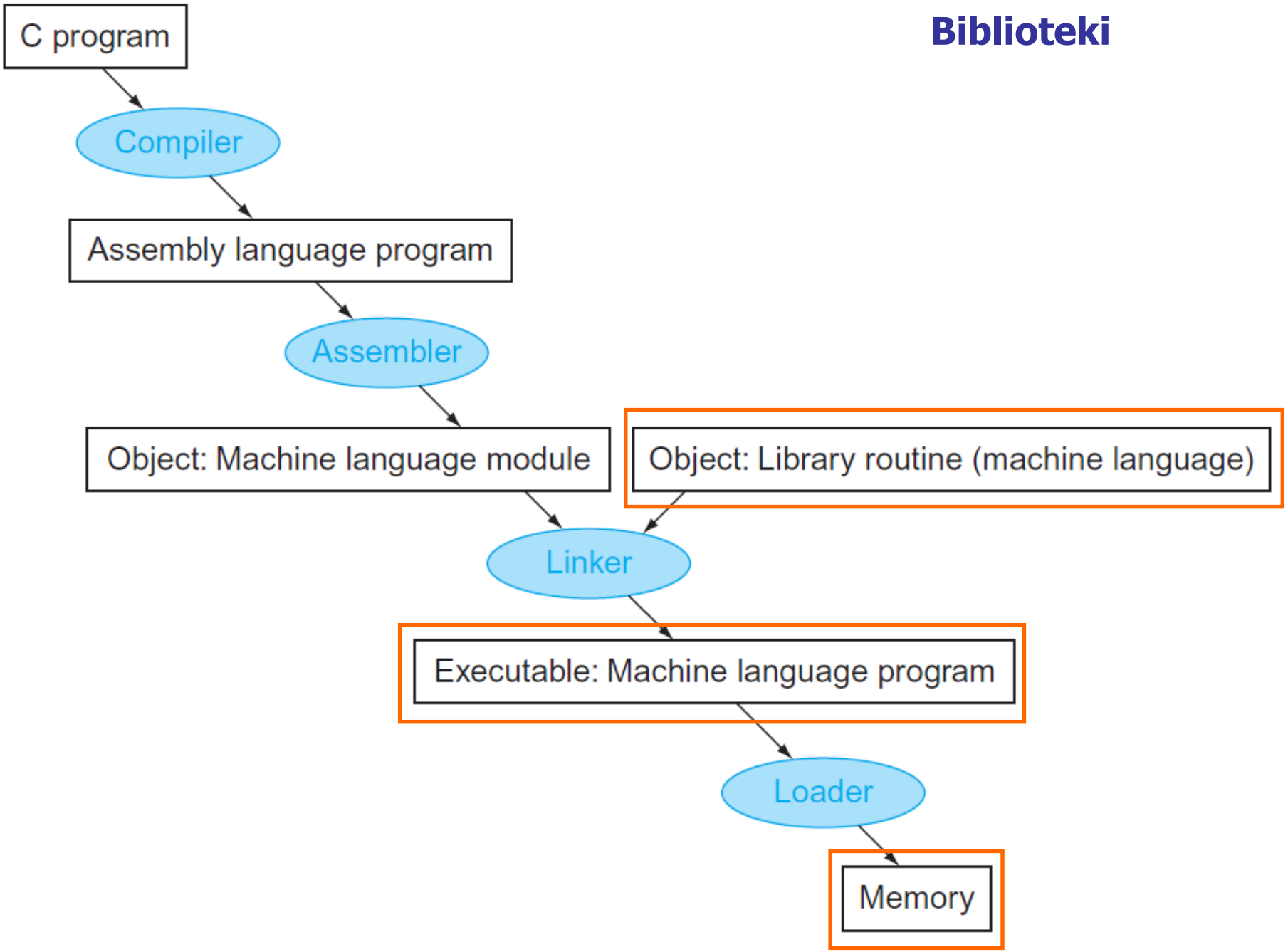
- Ostatni odłożony element – jest to **stos pełny (full)**

- wolne miejsce w pamięci, w które zostanie zapisany kolejny element
  (**stos pusty – empty**)

- większość mikroprocesorów ma implementowany sprzętowo jeden rodzaj stosu –
  najczęściej pełny malejący (*vide*: mapa pamięci procesu w typowym systemie operacyjnym)

- stos rosnący spotykany jest w mikrokontrolerach (np. 8051), a np. niektóre procesory ARM
  posiadają instrukcje mogące pracować z dowolnym stosem.

Biblioteki


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Uruchamianie programu pod kontrolą systemu operacyjnego (uproszczone)

Odpowiednia funkcja (loader) system operacyjnego:

- czyta nagłówek-strukturę pliku (ELF, EXE) aby określić niezbędny rozmiar pamięci na kod i dane staczne (.text i .data),
- tworzy i przydziela odpowiednią przestrzeń adresową (obecnie wirtualną),
- kopiuje kod programu, dane statyczne i kod bibliotek statycznych do nowoprzydzielonej pamięci,
- kopiuje argumenty przekazane z linii komend na stos,
- jeśli wszystko się powiedzie wykonuje skok do procedury „rozruchowej” (start-up routine – dodawana podczas linkowania), która **m.in.** sprawdza czy dostępne są wymagane biblioteki współdzielone, kopiuje przekazane do programu parametry ze stosu do odpowiednich rejestrów i dopiero wtedy wywołuje funkcje main (call),
- zakończenie funkcji main powoduje powrót (ret) do procedury rozruchowej, która kończy działanie programu wywołując funkcję systemową nr 60 - exit (przekazanie numeru błędu, zwolnienie zasobów).

Biblioteki statyczne - prosty i szybki sposób wywoływania funkcji bibliotecznych

- Skompilowane do postaci kod maszynowego funkcje biblioteczne zostają połączone podczas linkowania z głównym programem w jeden plik.

  + jeden plik wykonywalny zawiera wszystkie niezbędne biblioteki (i to we właściwej wersji),

  - jeśli jednak zostanie wydana nowsza / poprawiona wersja bilbioteki, chcąc jej użyć trzeba na nowo zbudować plik wykonywalny (ponownie zlinkować każdy program który jej używa),

  - większy rozmiar pliku wykonywalnego (niż w przypadku bibliotek współdzielonych).

- Wszystkie funkcje biblioteczne, do których występują odwołania w programie są ładowane do pamięci RAM podczas uruchamiania programu.

  + wywołanie funkcji przebiega szybko,

  - cześć funkcji może nigdy nie być wywołana (np. obsługa specyficznych wyjątków/błędów), ale zajmuje miejsce w pamięci operacyjnej.

Bilibioteki dynamicznie ładowane (Dynamically Linked Libraries – DLL) i współdzielone (Shared Objects - SO)

- kod maszynowy biblioteki nie jest połączony z głównym programem w jeden plik (plik wykonywalny ma więc mniejszy rozmiar),

- przeważnie pliki z bibliotekami współdzielonymi znajdują się w miejscu dostępnym dla wielu programów (odpowiednio ustawiona, znana ścieżka dostępu).

Wersja 1 – prostsza:

- cała biblioteka ładowana jest do pamięci podczas uruchomienia programu,

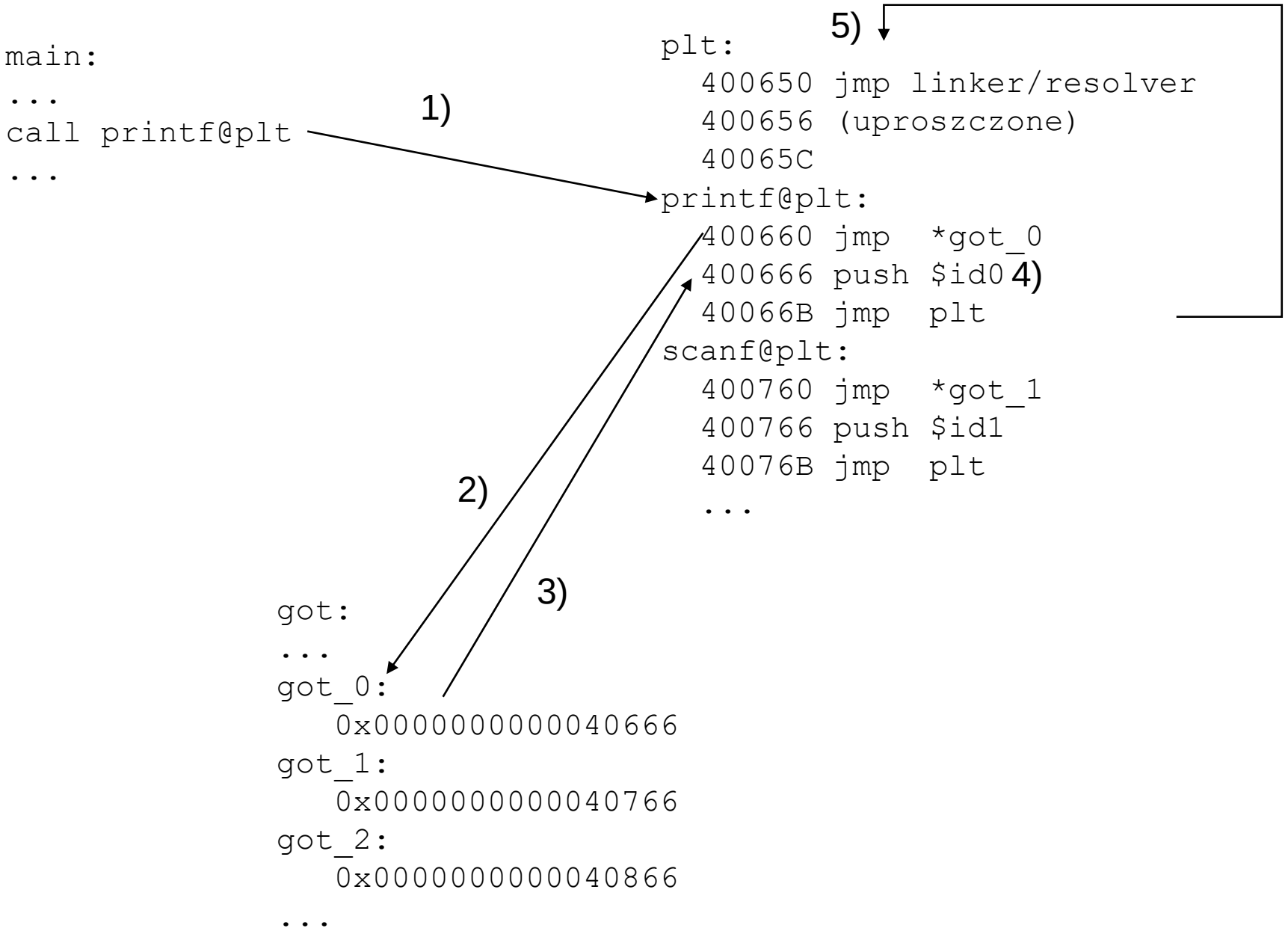
  wada: podobnie jak w wersji statycznej: pamięć mogą zajmować funkcje, które nigdy nie zostaną wywołane.

Wersja 2 – obecnie stosowana: ***Lazy/Late Binding (Loading, Linking…)*** itp.

- podczas uruchamiania ładowany do pamięci jest tylko program główny oraz tablice (w plikach ELF, Linux): *Procedure Linkage Table* - PLT oraz *Global Offset Table* – GOT, ew. procedura linkera dynamicznego,

- kod danej funkcji bibliotecznej ładowany jest do pamięci **dopiero podczas pierwszego wywołania** tej funkcji w programie.

Stan początkowy – przed pierwszym wywołaniem funkcji



Bilibioteki dynamicznie ładowane (Dynamically Linked Libraries – DLL) i współdzielone (Shared Objects - SO)

Zasada działania:

W programie wszystkie wywołania funkcji bibliotecznych odbywają się pośrednio – poprzez tablicę linkowania PLT, oraz tablicę GOT – zawierającą adresy funkcji.

Pierwsze wywołanie funkcji:

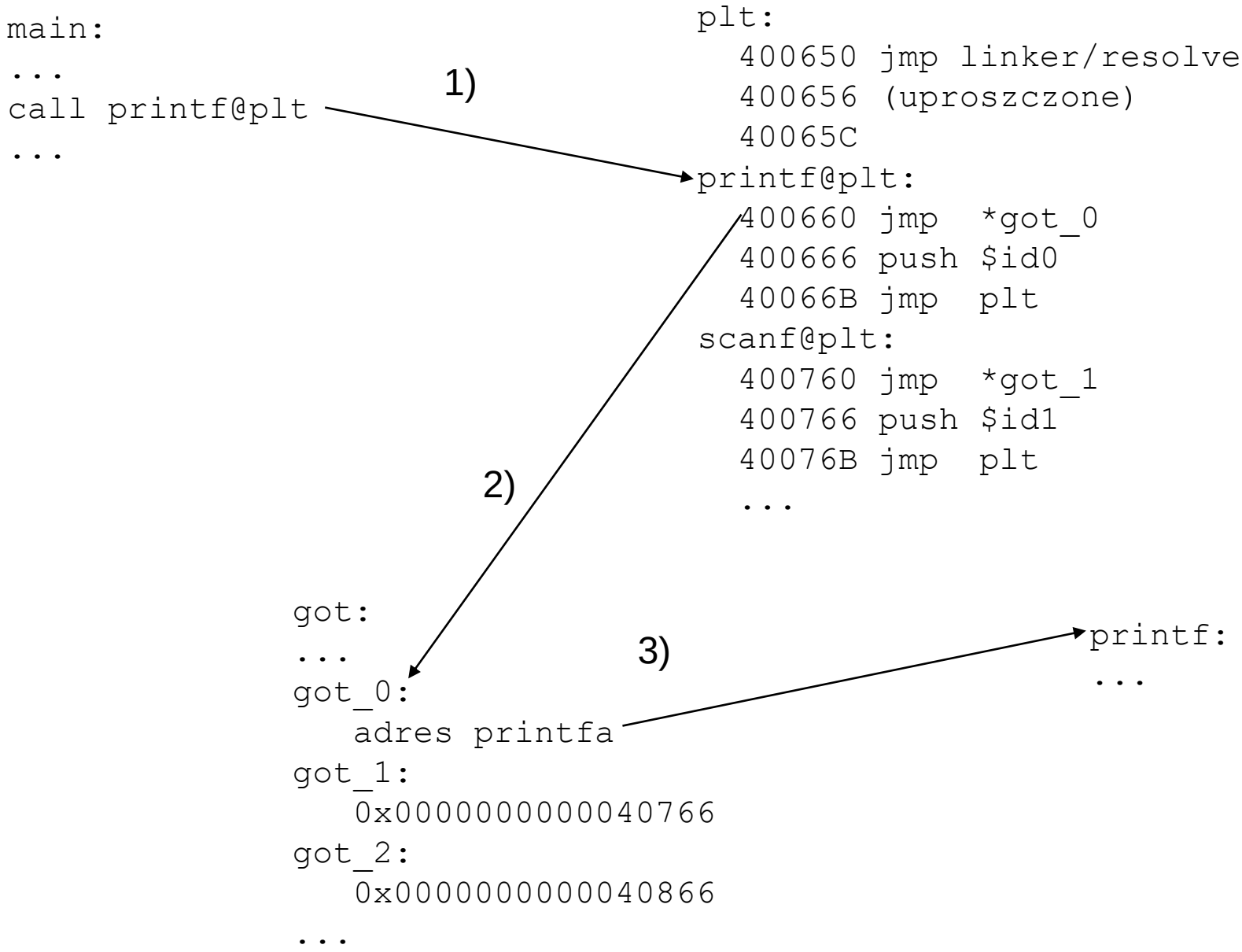
1) instrukcja call w programie głównym nie wywołuje bezpośrednio funkcji, ale prostą procedurę (*stub*) w tablicy PLT.

2) Wykonywany jest skok pod adres odczytany z odpowiedniego miejsca tablicy GOT.

3) Przy pierwszym wywołaniu danej funkcji adresem tym jest adres kolejnej instrukcji w PLT.

4) Na stosie umieszczany jest numer-identyfikator żądanej funkcji bibliotecznej, a następnie wywoływany linker dynamiczny (5), który ładuje do pamięci żądaną funkcję oraz zamienia odpowiadający jej adres w GOT na właściwy: prowadzący do miejsca w pamięci, gdzie funkcję załadowano. Funkcja zostaje uruchomiona.

Kolejne wywołanie tej samej funkcji bibliotecznej



Bilibioteki dynamicznie ładowane (Dynamically Linked Libraries – DLL) i współdzielone (Shared Objects - SO)

- Rozwiązanie jest wolniejsze od wersji statycznej: szczególnie pierwsze wykonanie funkcji trwa dłużej – ponieważ uruchomiony musi być „linker” ładujący funkcję z pliku do pamięci. Podczas kolejnego wywołania narzutem jest dodatkowy skok (pośredni).

- Do pamięci ładowane są tylko funkcje, które faktycznie muszą być wykonane.

- Jedna, współdzielona przez wiele programów biblioteka może być łatwo* zastąpiona nowszą wersją - nie jest wymagane ponowne linkowanie wszystkich korzystających z niej programów.

- Kod bibliotek dynamicznych musi być napisany w specjalny sposób – jako *Position Independend Code* (PIC) – relokowalny, czyli taki, który może być załadowany w (prawie) dowolny obszar pamięci. Docelowe miejsce ładowania funkcji nie jest znane przed uruchomieniem programu. Różne programy mogą swoje kopie tej samej, współdzielonej funkcji umieszczać w innym miejscu w pamięci.

  Kod PIC nie może zawierać adresów absolutnych (tzn. numerów komórek pamięci) np. w dostępie do sekcji .data. W X86-64 bazuje natomiast na adresowaniu względnym, opartym o licznik rozkazów %rip.

*o ile nie wystąpią problemy z kompatybilnością, „*dependency hell”* itp.