PLIKI WYKONYWALNE I ICH ANALIZA

Biblioteki statyczne i dynamiczne + binutils

Anna Bogusz, Grzegorz Burzyński

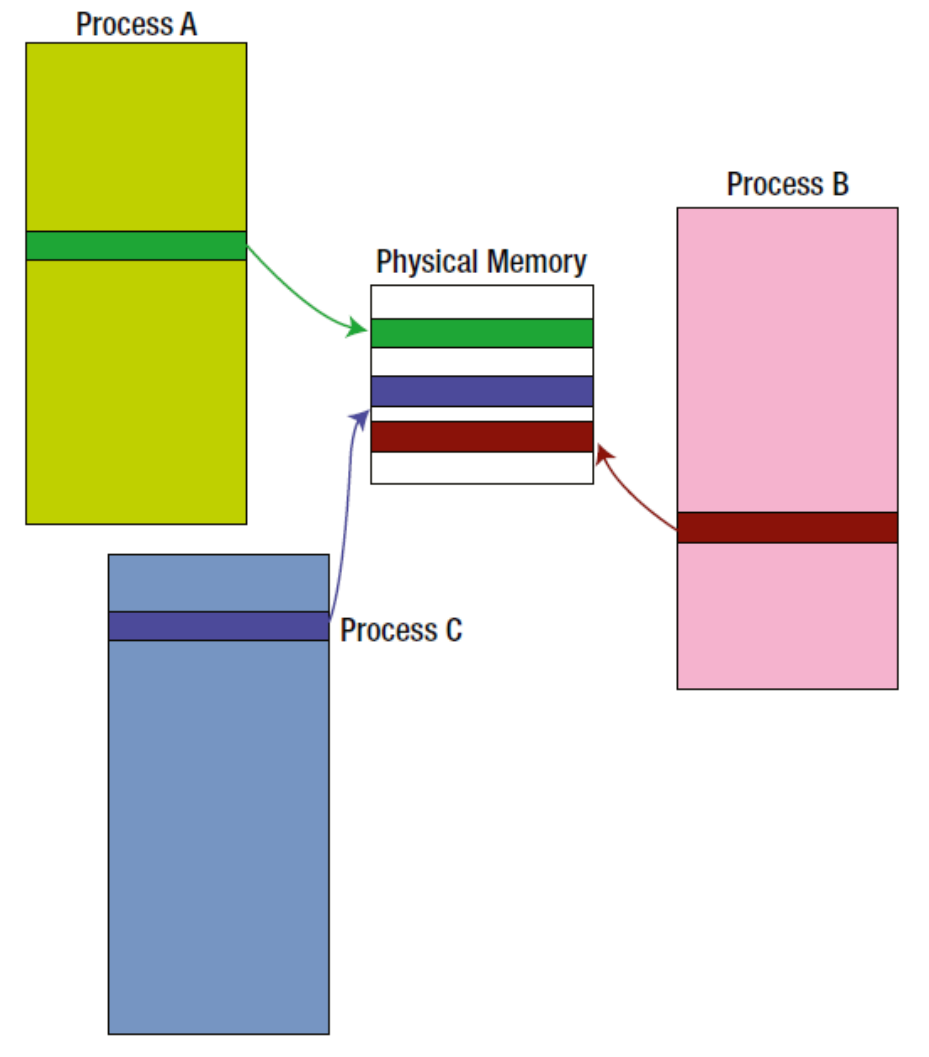
28 marca 2017

Akademia Górniczo-Hutnicza
Wydział Fizyki i Informatyki Stosowanej
Programowanie niskopoziomowe

Wprowadzenie do plików wykonywalnych

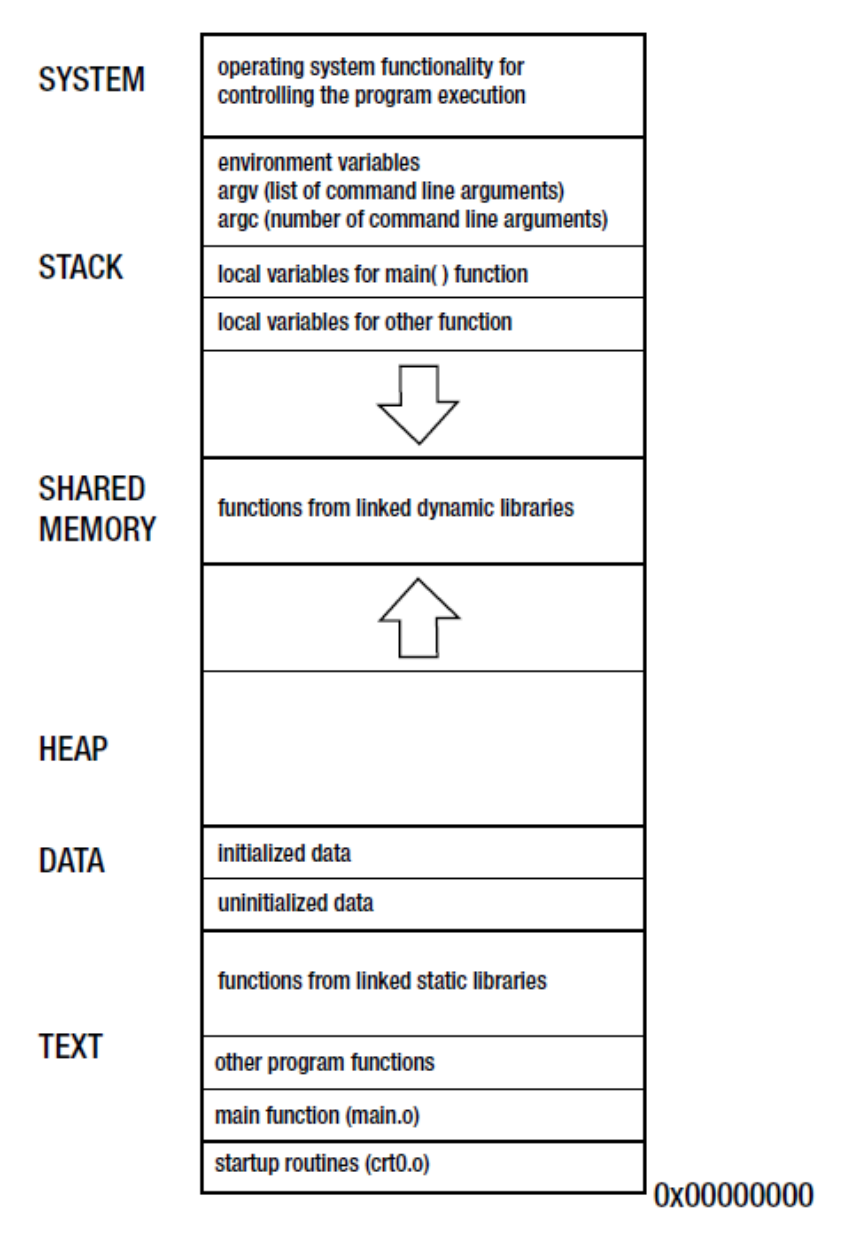
PAMIĘĆ WIRTUALNA - PRZYPOMNIENIE

Pamięć wirtualna - mechanizm zarządzania pamięcią komputera zapewniający procesowi wrażenie pracy w jednym, dużym, ciągłym obszarze pamięci operacyjnej.



Rysunek: Koncept pamięci wirtualnej

MAPA PAMIĘCI PROCESU



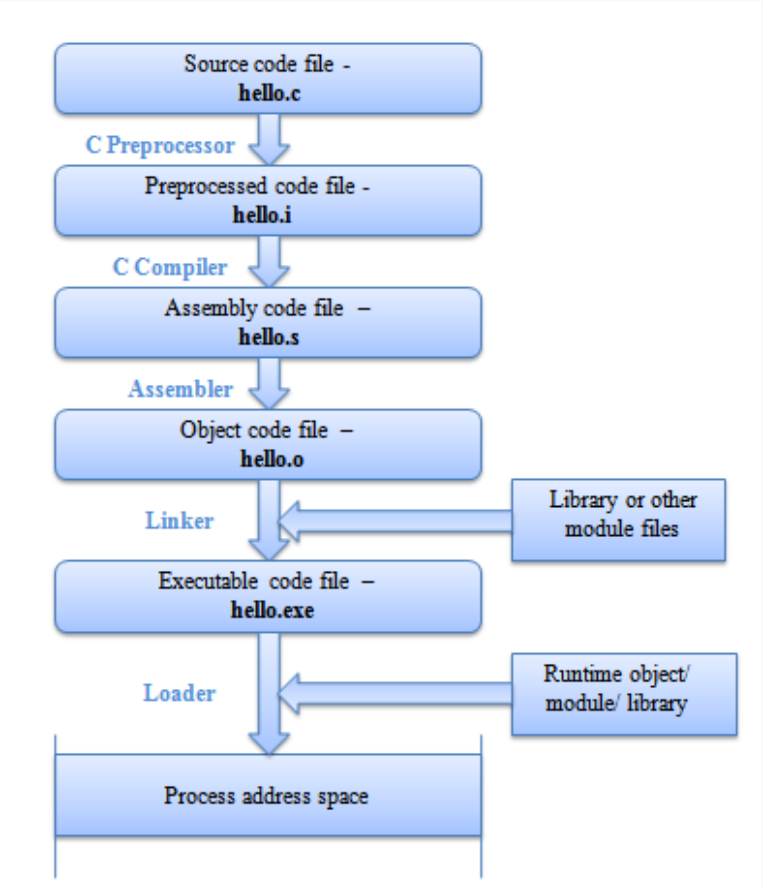
Proces to działająca instancja programu.

ROLE W PROCESIE BUDOWY I URUCHAMIANIA PROGRAMU

- **Pliki wykonywalne** - przechowują ”szkic” mapy pamięci programu
- **Linker** - tworzy plik wykonywalny, łącząc pliki binarne, wypełniając poszególne sekcje
- **Loader** - ładuje program, tworzy faktyczną mapę pamięci procesu

Schemat prawdziwy dla najpopularniejszych współczesnych systemów operacyjnych (Windows, Linux).

Przed uruchomieniem programu, musimy przejść wszystkie etapy budowania pliku wykonywalnego. Dopiero odpowiednio przygotowany **plik binarny**, o określonym i zrozumiałym formacie, będzie mógł być uruchomiony przez **loader**.



Rysunek: Od źródeł do egzekucji

NA CZYM SIĘ SKUPIMY?

- Wyniki kompilacji i ich struktura

NA CZYM SIĘ SKUPIMY?

- Wyniki kompilacji i ich struktura
- Co zrobi z nimi linker

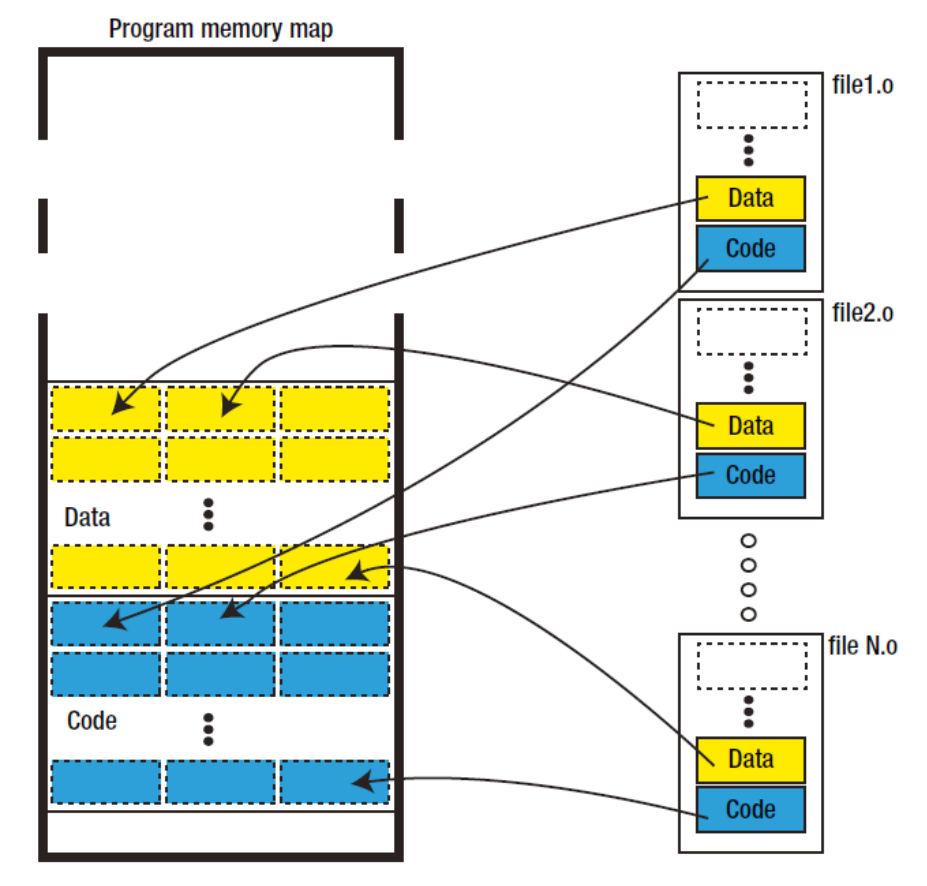
NA CZYM SIĘ SKUPIMY?

- Wyniki kompilacji i ich struktura
- Co zrobi z nimi linker
- Jak wynikiem linkowania posłuży się loader

Przykład 01 - zawartość pliku obiektowego

- rezultat tłumaczenia pojedynczego pliku źródłowego
- podstawowymi składnikami pliku obiektowego są symbole oraz sekcje
- kefelek, którego użyjemy w trakcie linkowania
- nie określa dokładnie gdzie jego sekcje znajdą się w pliku wykonywalnym

Zadaniem linkera jest ”poskładać” pliki obiektowe (oraz biblioteki) do pliku wykonywalnego (lub biblioteki). Rozumie on strukturę plików obiektowych i tworzy nową strukturę na ich bazie.

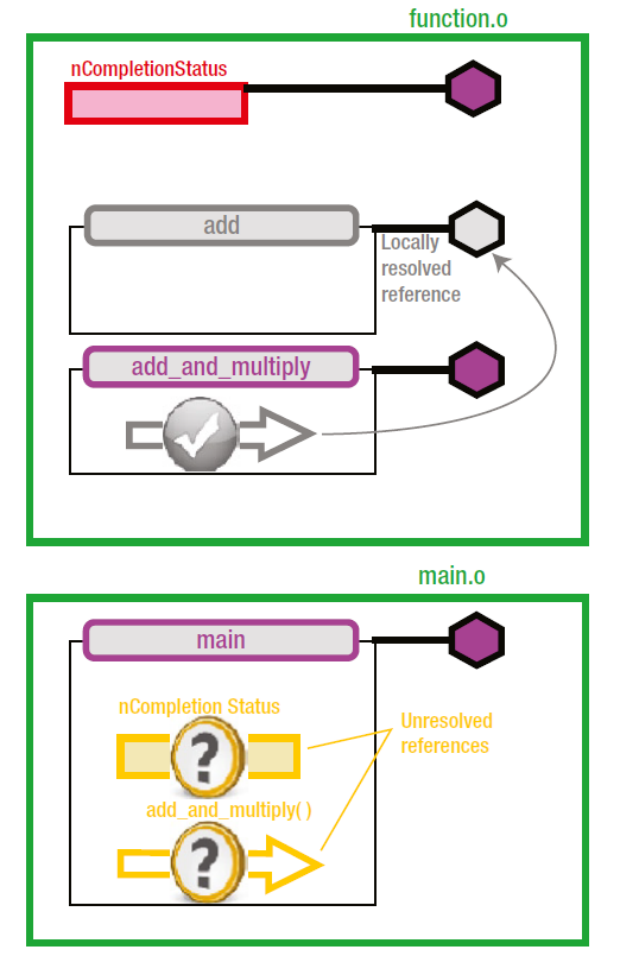


Rysunek: Działanie linkera

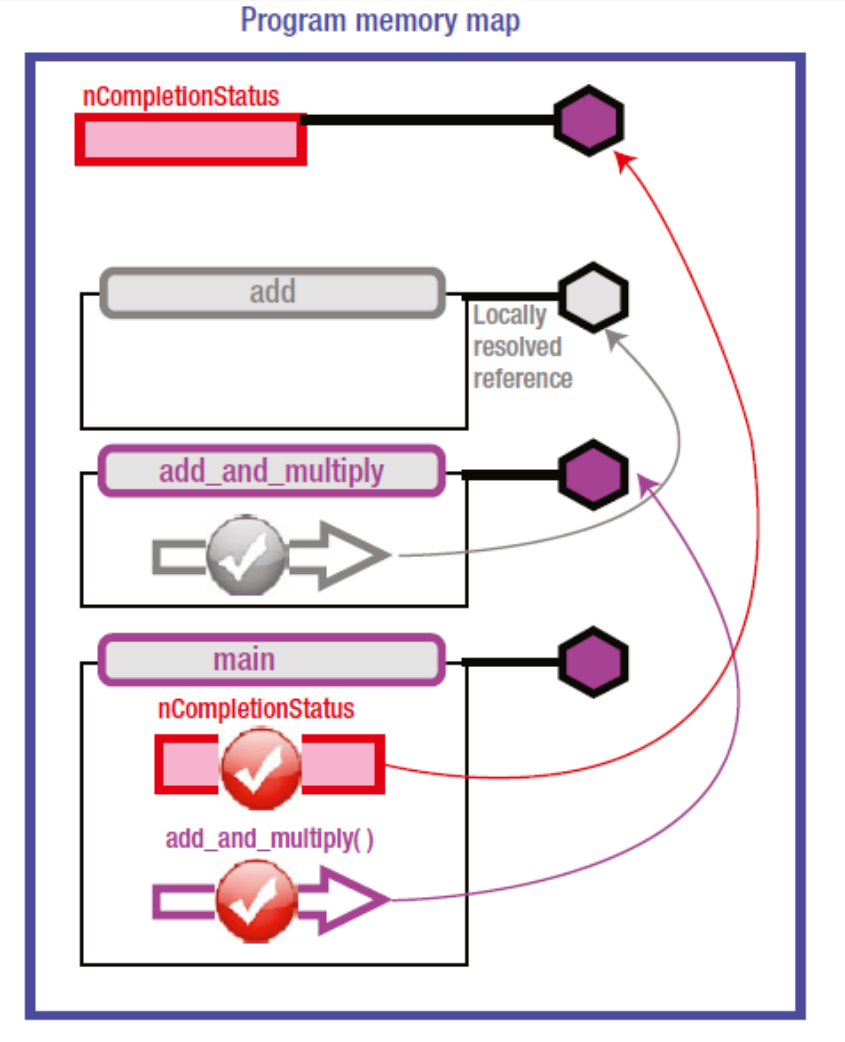
CZY LINKER JEST POTRZEBNY?

- składanie sekcji nie zawsze jest trywialne
- rozbicie budowania na kompilację i linkowanie sprzyja ponownemu użyciu modułów bez konieczności ich rekompilacji

- **Relokacja** - przenoszenie sekcji z plików obiektowych do pliku docelowego
- **Rozwiązywanie referencji** - bardziej złożony od pierwszego etapu; ma na celu rozwiązanie referencji pomiędzy modułami, tak aby stworzyć homogeniczną strukturę

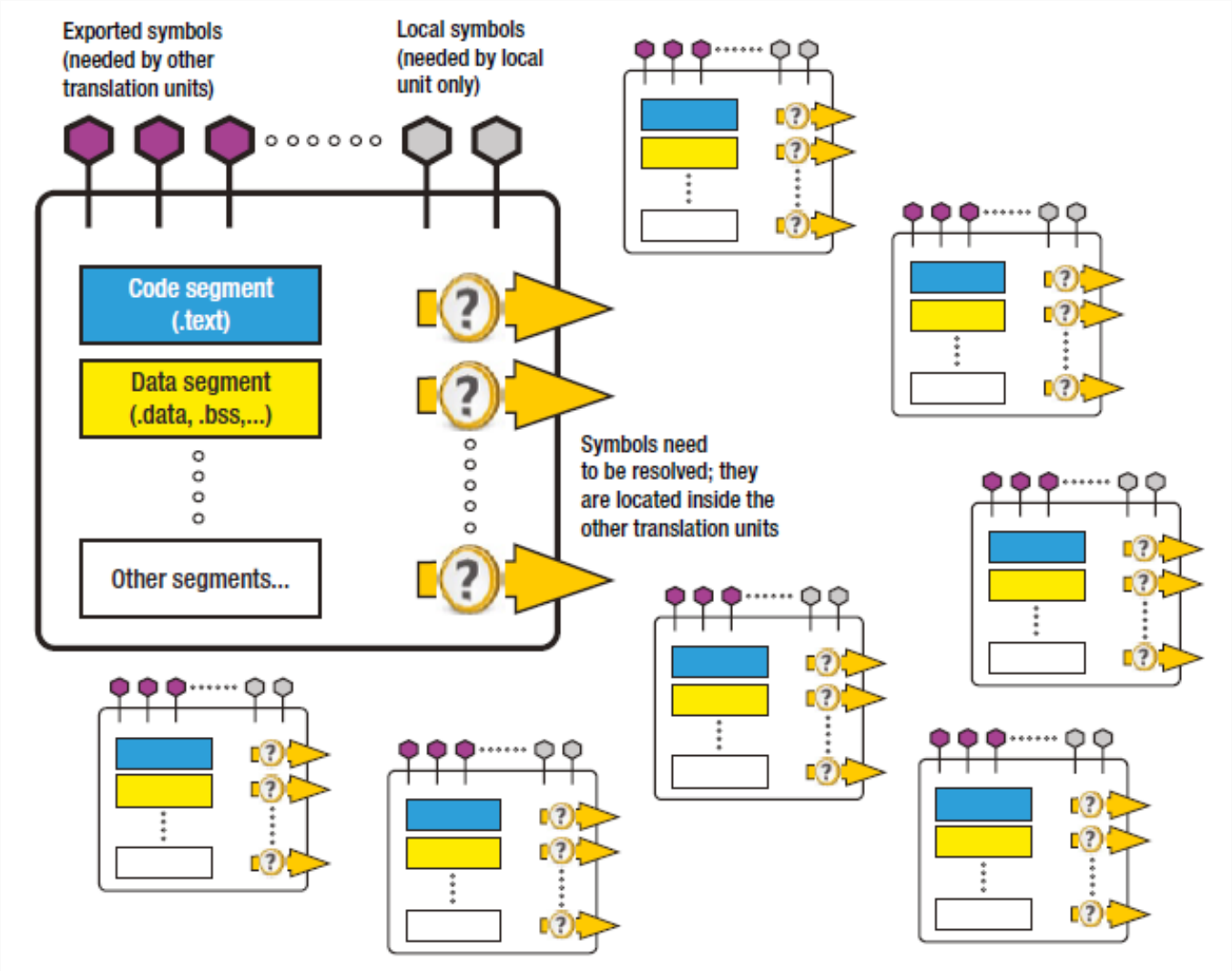


Rysunek: Pliki obiektowe przed linkowaniem



Rysunek: Zlinkowany plik wykonywalny

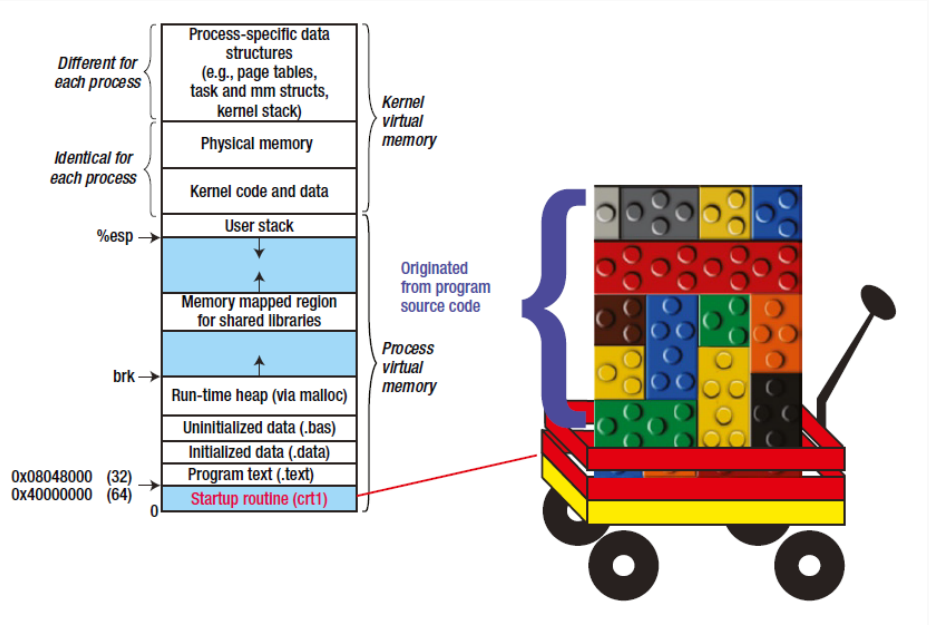
Przykład 02 - linkowanie



Rysunek: Widok linkera

PO LINKOWANIU

Po linkowaniu otrzymujemy plik gotowy do uruchomienia, w formacie odpowiednim dla danego systemu. W przypadku Linuxa jest to **ELF**. Plik wykonywalny posiada dodany entry point w postaci rutyny startowej.



Rysunek: Mapa pamięci programu na podstawie pliku wykonywalnego

FORMATY PLIKÓW WYKONYWALNYCH

Aby ujednolicić sposób, w jaki informacje na temat wykonywalnego programu są przechowywane, powstawały różne formaty plików.

Obecnie dwoma najpopularniejszymi są:

- **ELF** (Executable and Linkable Format) - unix-like
- **PE** (Portable Executable) - systemy z rodziny Microsoft Windows

Inne formaty:

- a.out (assembler output)
- COFF

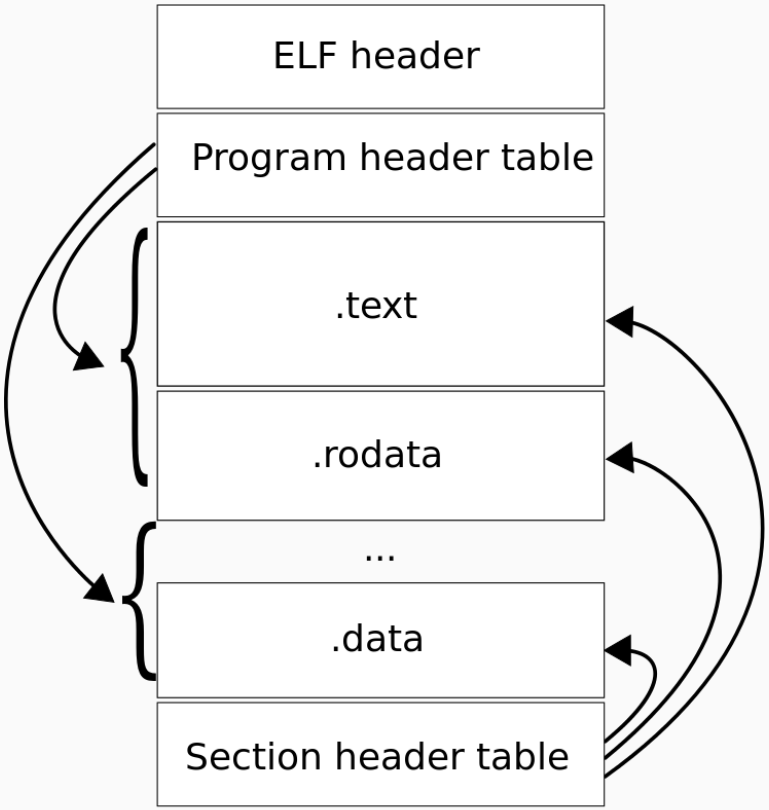
ELF został pierwotnie zdefiniowany w specyfikacji ABI Unixa - System V Release, później w Tool Interface Standard (1995 rok).

W założeniach ELF jest:

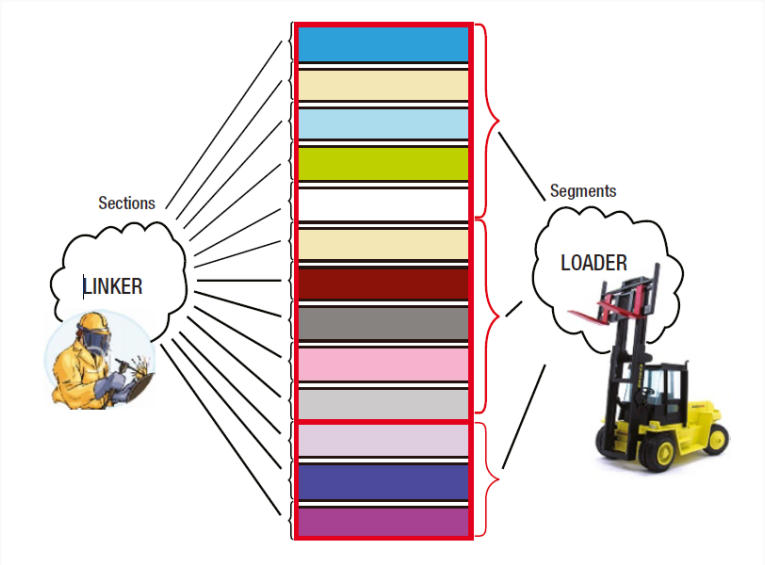
- elastyczny, rozszerzalny
- niezwiązany z konkretnym typem procesora

Jego przenośna natura pozwoliła na zaadaptowanie go w wielu środowiskach, np.:

- Linux, Solaris, FreeBSD, NetBSD (i inne unix-like)
- Windows (?!)
- konsole Playstation >= 2
- Android >= 5.0 (Android Runtime)



Rysunek: Budowa pliku ELF



Rysunek: Linker vs. loader

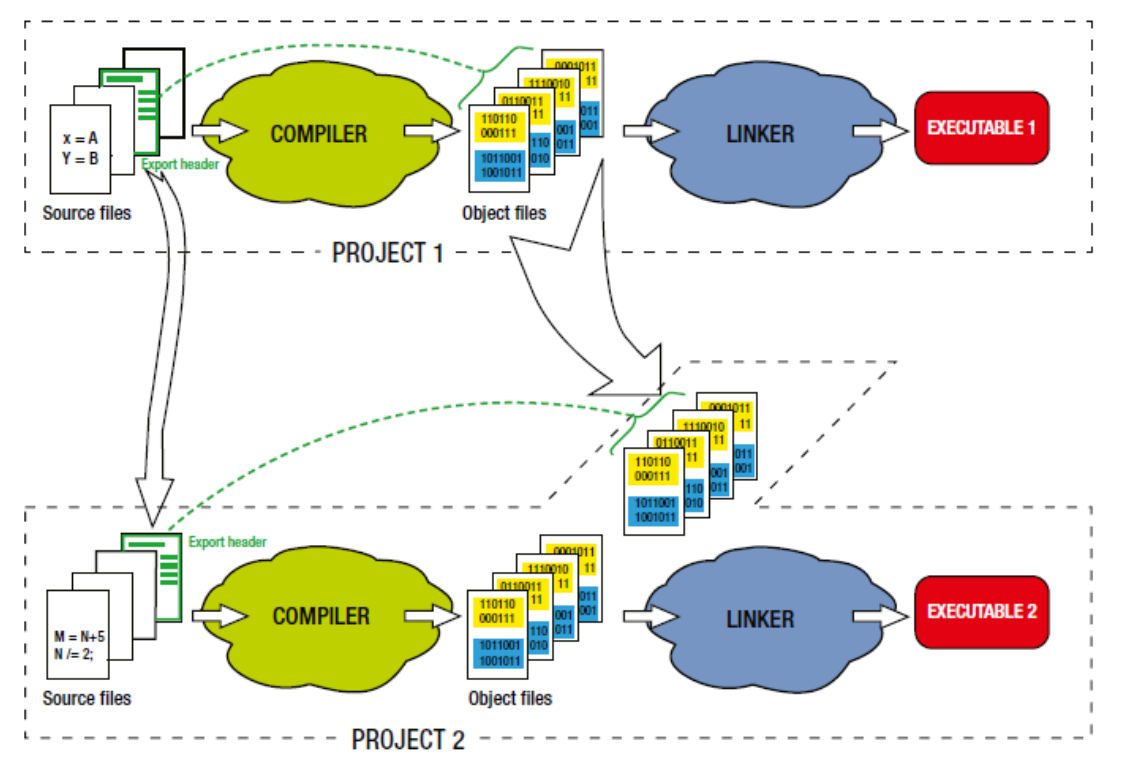
Sekcje zawierają się w ciągłych obszarach. Nie mogą na siebie nachodzić. Możliwe sekcje to nie tylko .text i .data, czy .bss, które są nieodzowne dla uruchomienia programu. Inne przykładowe sekcje:

- .ctors/.dtors - zainicjowane wskaźniki do konstruktorów C++
- .debug - informacje do debugowania
- .dynamic - informacje na temat dynamiczne linkowania
- .line - informacje łączące kod maszynowy z liniami w kodzie źródłowym
- .init/.fini - instrukcje do wykonania przy inicjalizacji i kończeniu egzekucji
- .symtab - tabela symboli

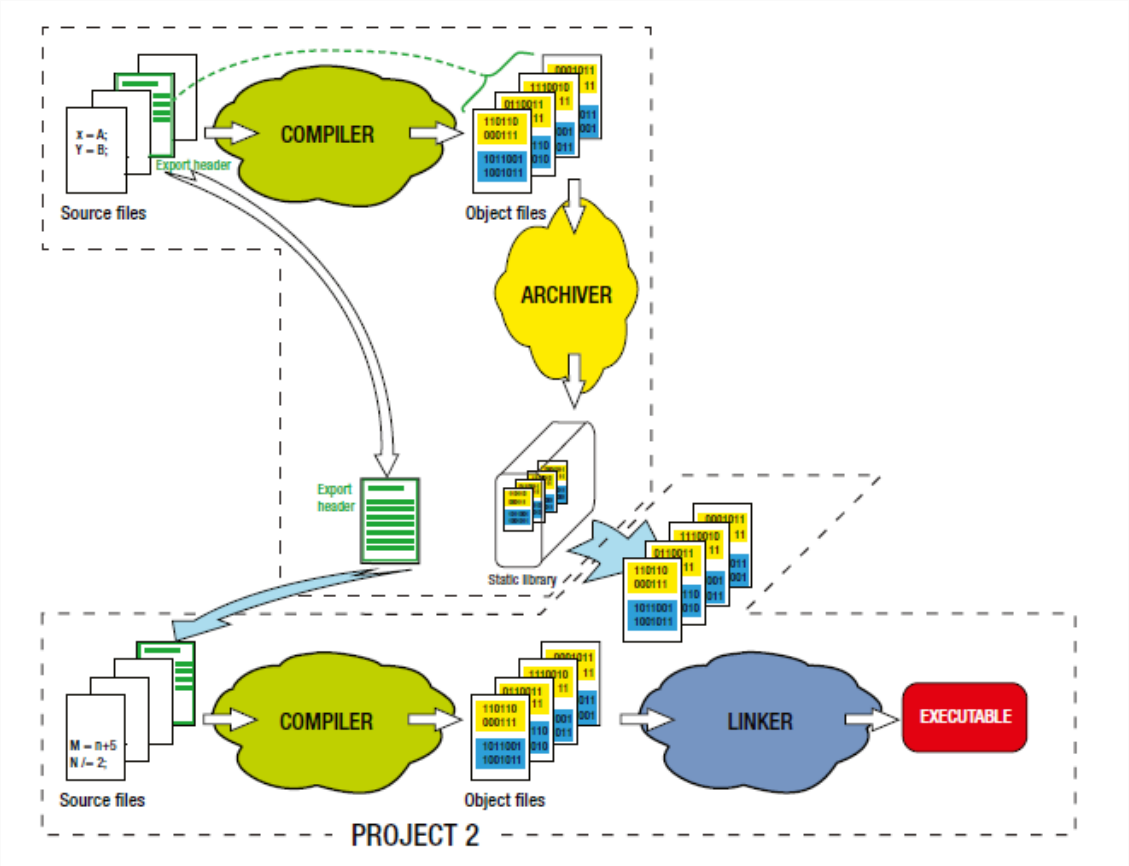
ELF udostępnia szeroką gamę rodzajów symboli, np.:

- ”N” - symbol debugowy
- ”T” - symbol w sekcji .text (kod)
- ”U” - undefined
- ”W” - weak symbol

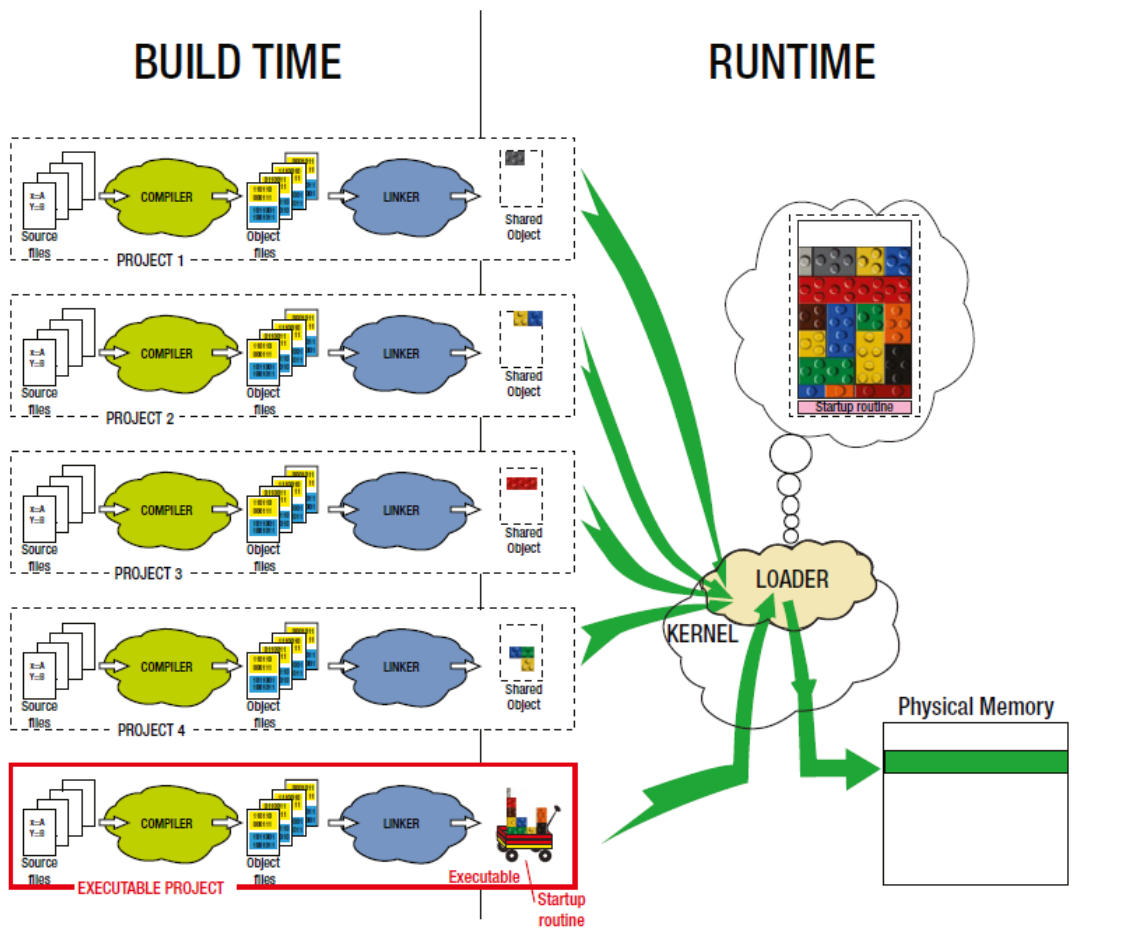
BIBLIOTEKI



Rysunek: Naiwne podejście do ponownego użycia plików obiektowych

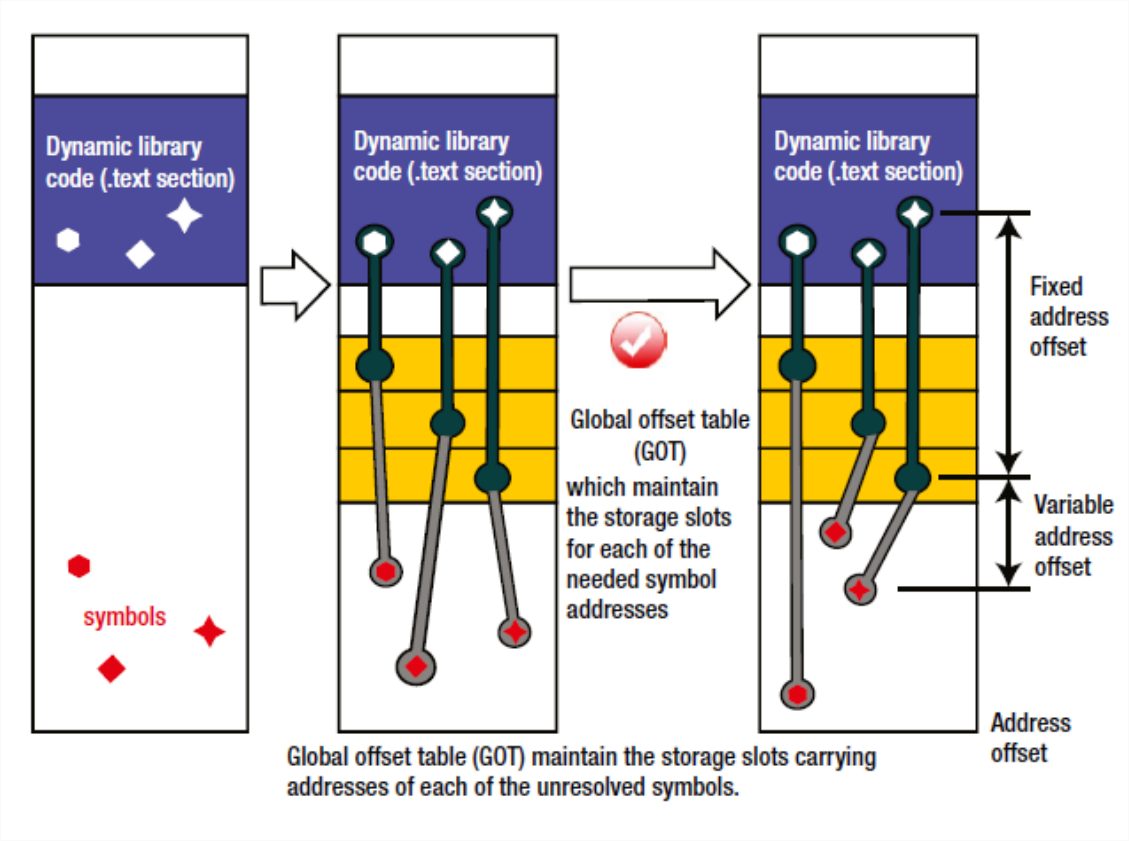


Rysunek: Użycie biblioteki statycznej



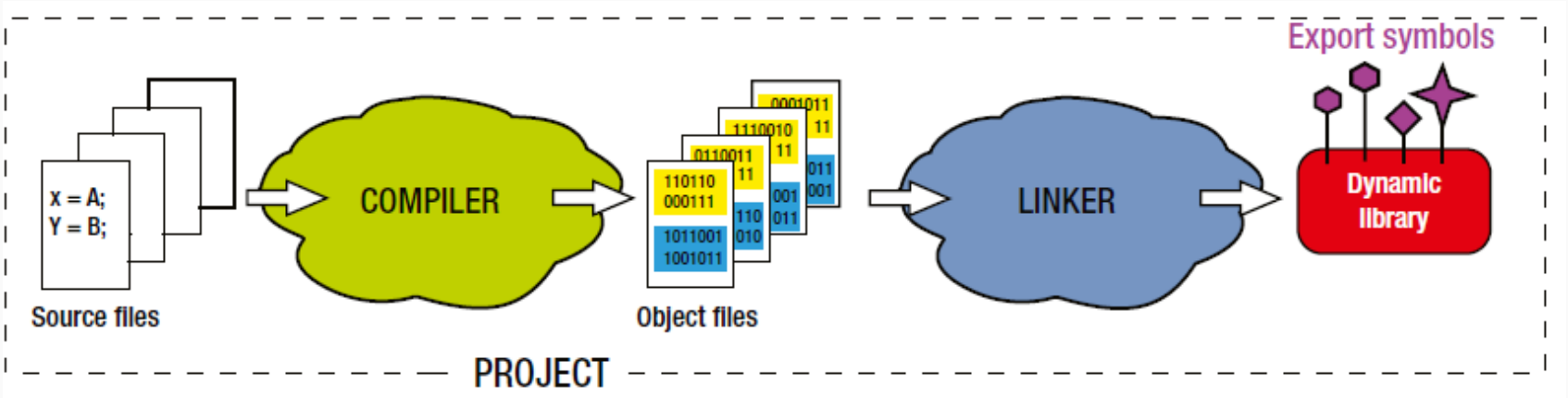
Rysunek: Biblioteka dynamiczna

Początkowo dynamiczne biblioteki były ładowane osobno dla każdego procesu (Load Time Reloaction). W celu zniwelowania tej wady wymyślono **PIC** (Position Independent Code).



Rysunek: Koncepcja PIC

Więcej o bibliotekach dynamicznych

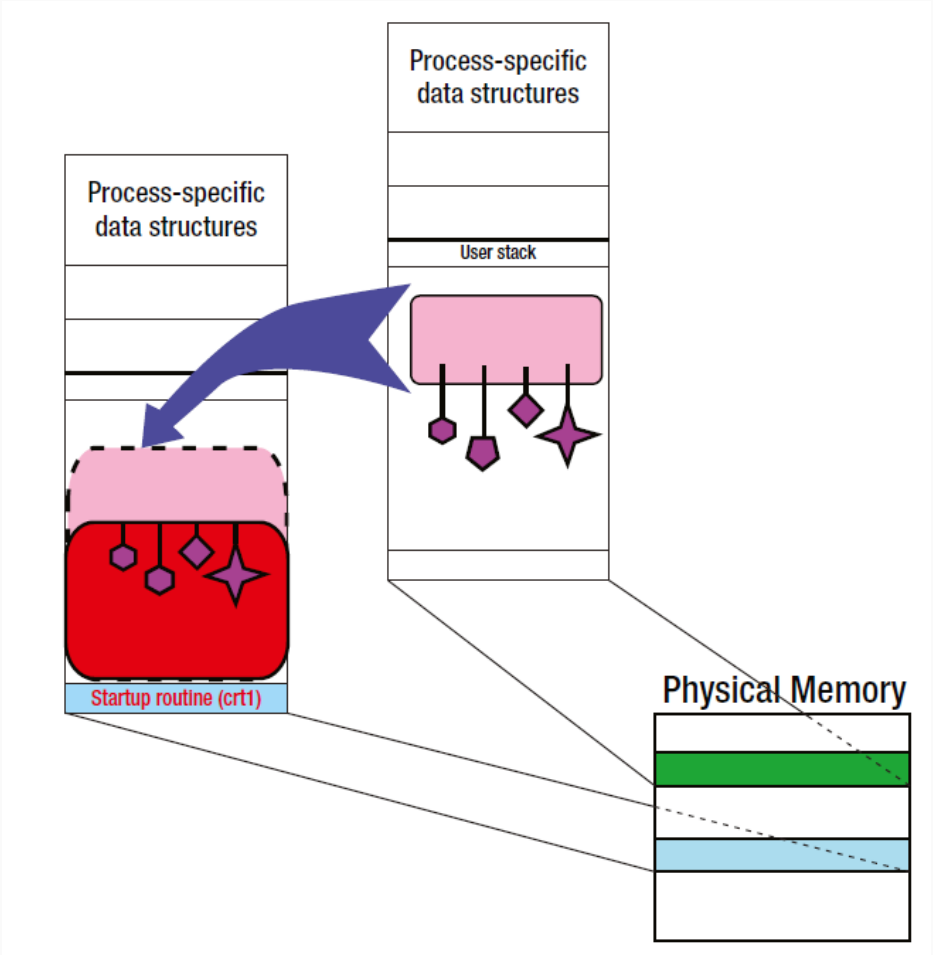


Biblioteka dynamiczna, tak jak plik wykonywalny, musi przejść nie tylko kompilację, ale też linkowanie. Domyślnie linker pozwala na nierozwiązane referencje - można zapobiec temu flagą –no-allow-shlib-undefined.

BUDOWANIE KLIENTA BIBLIOTEKI

- Podczas budowania klienta biblioteki (pliku wykonywalnego), linker sprawdza jedynie, czy **symbole**, których potrzebuje klient są eksportowane przez bibliotekę
- Pozwala to na linkowanie z jedną biblioteką, a ładowanie **innej** - eksportującej te same symbole

W trakcie uruchomienia pliku wykonywalnego, dynamiczny linker poszukuje bibliotek, z którymi plik został wcześniej zlinkowany. Istnieją określone reguły tych poszukiwań.



Rysunek: Linkowanie dynamiczne

MOŻLIWOŚĆ ŁADOWANIA SYMBOLI RUNTIME

Normalnym trybem jest ładowanie biblioteki dynamicznej **statically aware** (w load time). Możliwe jest jednak skorzystanie z ładowania w runtime. Dokonujemy tego programowo.

```
1  #include <dlfcn.h>
2  ...
3  double (*cosine)(double);
4  char *error;
5
6  // dlopen opens a library and prepares it for
     ↪  use
7  void* handle = dlopen("libm.so.6", RTLD_LAZY);
8
9  // check if an error occured
10 if (!handle) {
11     fputs(dlerror(), stderr);
12     exit(1);
13 }
14
15 // symbol lookup
16 cosine = dlsym(handle, "cos");
17 if ((error = dlerror()) != NULL)  {
18     fputs(error, stderr);
19     exit(1);
20 }
```

Listing 1: Ładowanie biblioteki runtime

- Uważaj na name mangling (C++)

- Uważaj na name mangling (C++)
- Najlepiej implementuj ABI jako funkcje C

- Uważaj na name mangling (C++)
- Najlepiej implementuj ABI jako funkcje C
- Dostarczaj header z pełną definicją ABI

- Uważaj na name mangling (C++)
- Najlepiej implementuj ABI jako funkcje C
- Dostarczaj header z pełną definicją ABI
- Eksportuj tylko najważniejsze symbole

- Uważaj na name mangling (C++)
- Najlepiej implementuj ABI jako funkcje C
- Dostarczaj header z pełną definicją ABI
- Eksportuj tylko najważniejsze symbole
- Używaj przestrzeni nazw (C++)

Compiler	void h(int)	void h(int, char)	void h(void)
Intel C++ 8.0 for Linux	_Z1hi	_Z1hic	_Z1hv
HP aC++ A.05.55 IA-64			
IAR EWARM C++ 5.4 ARM			
GCC 3.*x* and higher			
IAR EWARM C++ 7.4 ARM	_Z<number>hi	_Z<number>hic	_Z<number>hv
GCC 2.9*x*	h__Fi	h__Fic	h__Fv
HP aC++ A.03.45 PA-RISC			
Microsoft Visual C++ v6-v10 (mangling details)	?h@@YAXH@Z	?h@@YAXHD@Z	?h@@YAXXZ
Digital Mars C++			
Borland C++ v3.1	@h$qi	@h$qizc	@h$qv
OpenVMS C++ V6.5 (ARM mode)	H__XI	H__XIC	H__XV
OpenVMS C++ V6.5 (ANSI mode)		CXX$__7H__FIC26CDH77	CXX$__7H__FV2CB06E8
OpenVMS C++ X7.1 IA-64	CXX$_Z1HI2DSQ26A	CXX$_Z1HIC2NP3LI4	CXX$_Z1HV0BCA19V
SunPro CC	__1cBh6Fi_v_	__1cBh6Fic_v_	__1cBh6F_v_
Tru64 C++ V6.5 (ARM mode)	h__Xi	h__Xic	h__Xv
Tru64 C++ V6.5 (ANSI mode)	__7h__Fi	__7h__Fic	__7h__Fv
Watcom C++ 10.6	W?h$n(i)v	W?h$n(ia)v	W?h$n()v

Rysunek: Name mangling convention w różnych kompilatorach

KONCEPT PLUGINÓW

- Dodawanie lub usuwanie pluginów nie powinno wymagać rekompilacji programu
- Pluginy powinny udostępniać ustalony interfejs (ABI)
- Program może dostosować się do swojego otoczenia i używać ”podsuniętej” biblioteki

- Program może dawać użytkownikowi możliwość wyboru, którego pluginu chce użyć (np. poprzez GUI)

Przykłady użycia pluginów w istniejącym oprogramowaniu:

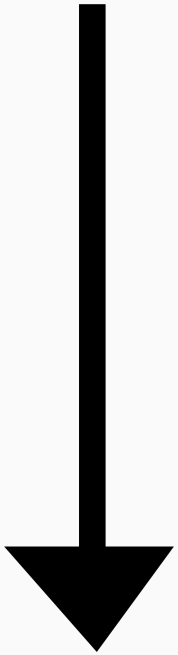
- Wtyczki z filtrami do Adobe Photoshopa
- Pluginy VST do procesowania audio
- Pluginy do IDE (np. Visual Studio)
- Wtyczki do przeglądarek internetowych

REGUŁY WYSZUKIWANIA BIBLIOTEK

REGUŁY WYSZUKIWANIA BIBLIOTEK

System operacyjny określa reguły, w zgodzie z którymi biblioteki są wyszukiwane, kiedy ich potrzebujemy. Kolejność w Linuxie:

- Preloaded libraries - zmienna *LD_PRELOAD* lub */etc/ld.so.preload*
- *LD_LIBRARY_PATH*
- *runpath* (pole *DT_RUNPATH* w ELFie)
- */etc/ld.so.cache* - konfigurowalne przez */etc/ld.so.conf*
- */lib* i */usr/lib*



WERSJONOWANIE BIBLIOTEK

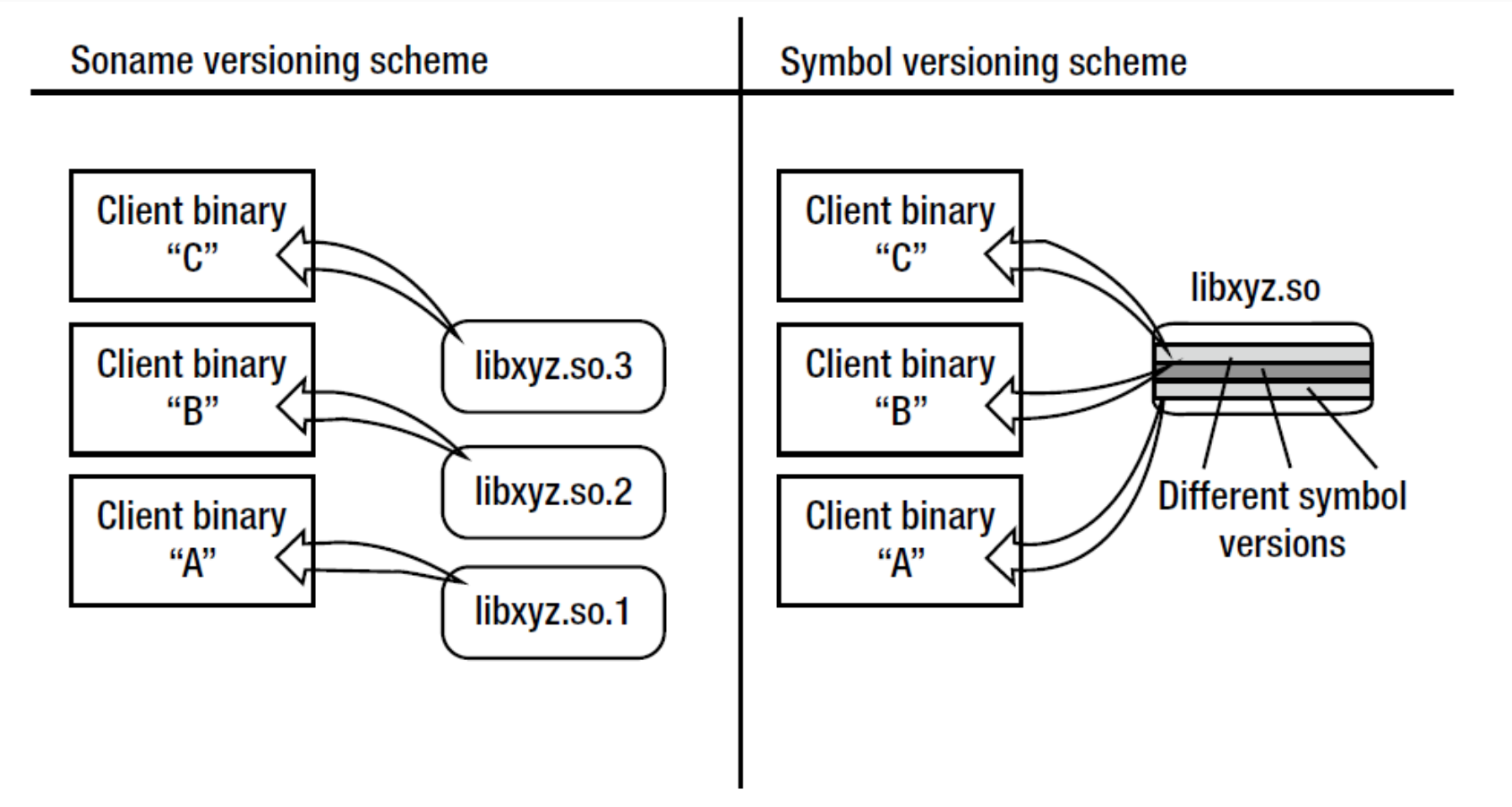
Schemat wersjonowania bibliotek:

$$lib < nazwa\_biblioteki > .so. < version\_info >$$

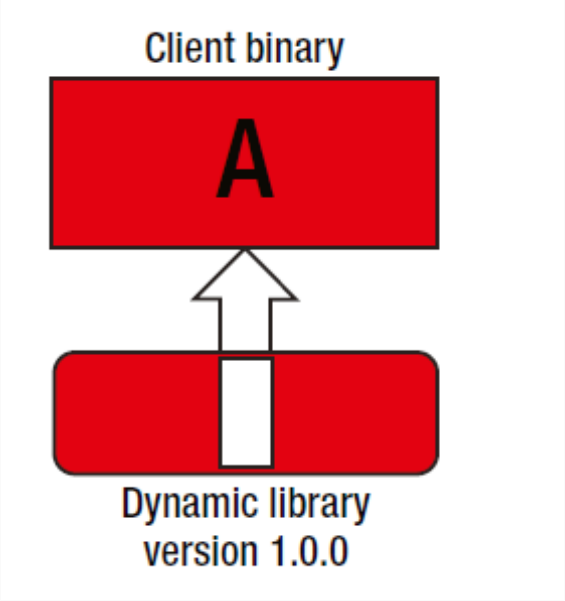
Dodatkowo, wersja składa się z trzech składowych:

$$< major > . < minor > . < patch >$$

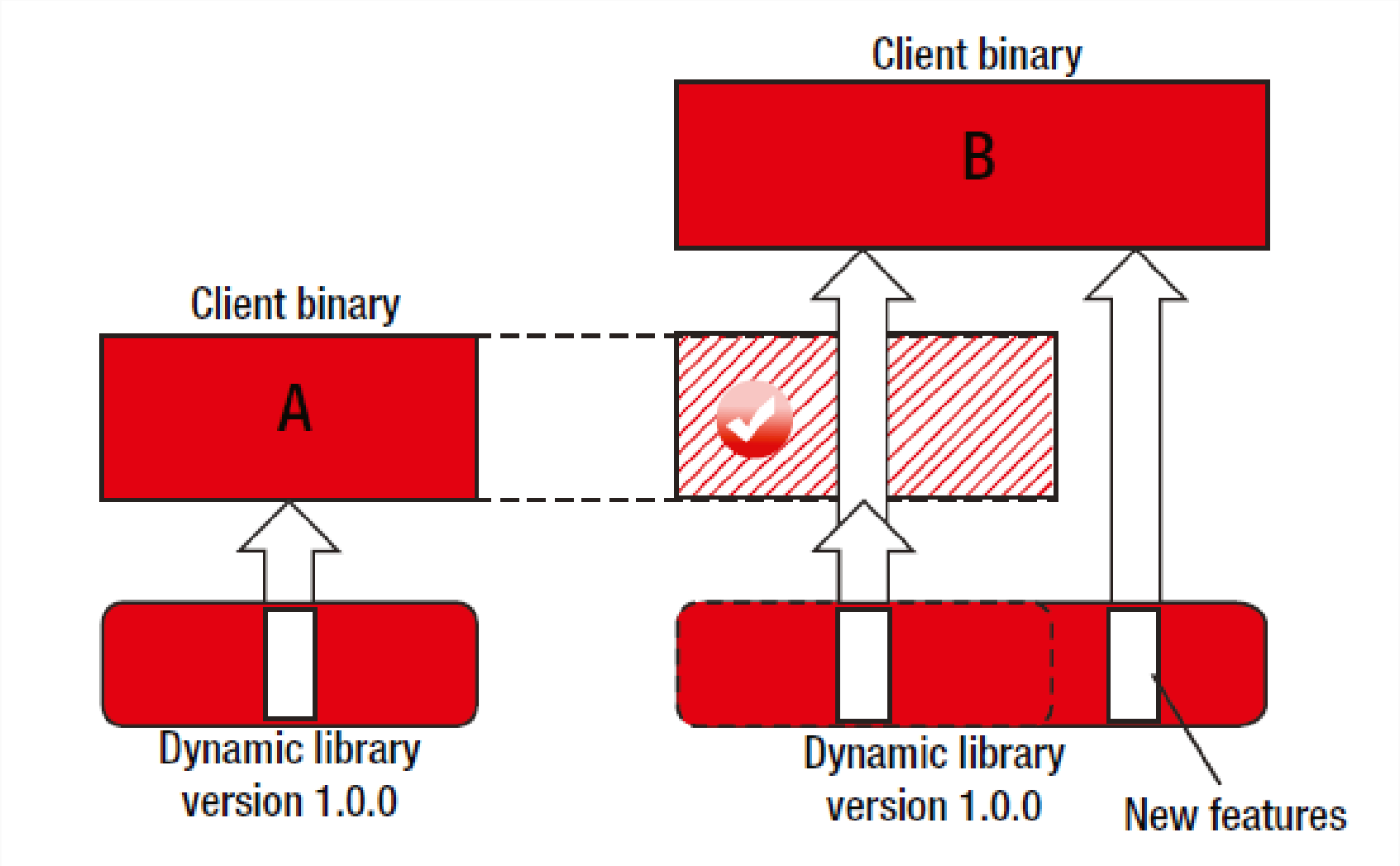
Ogólnie przyjętą regułą jest, aby te same wersje $< major >$ były ze sobą zgodne. W celu regulowania tego, używa się pola *DT_SONAME*, zawierającego tylko część $< major >$ wersji.



Rysunek: Porównanie wersjonowania soname i symboli

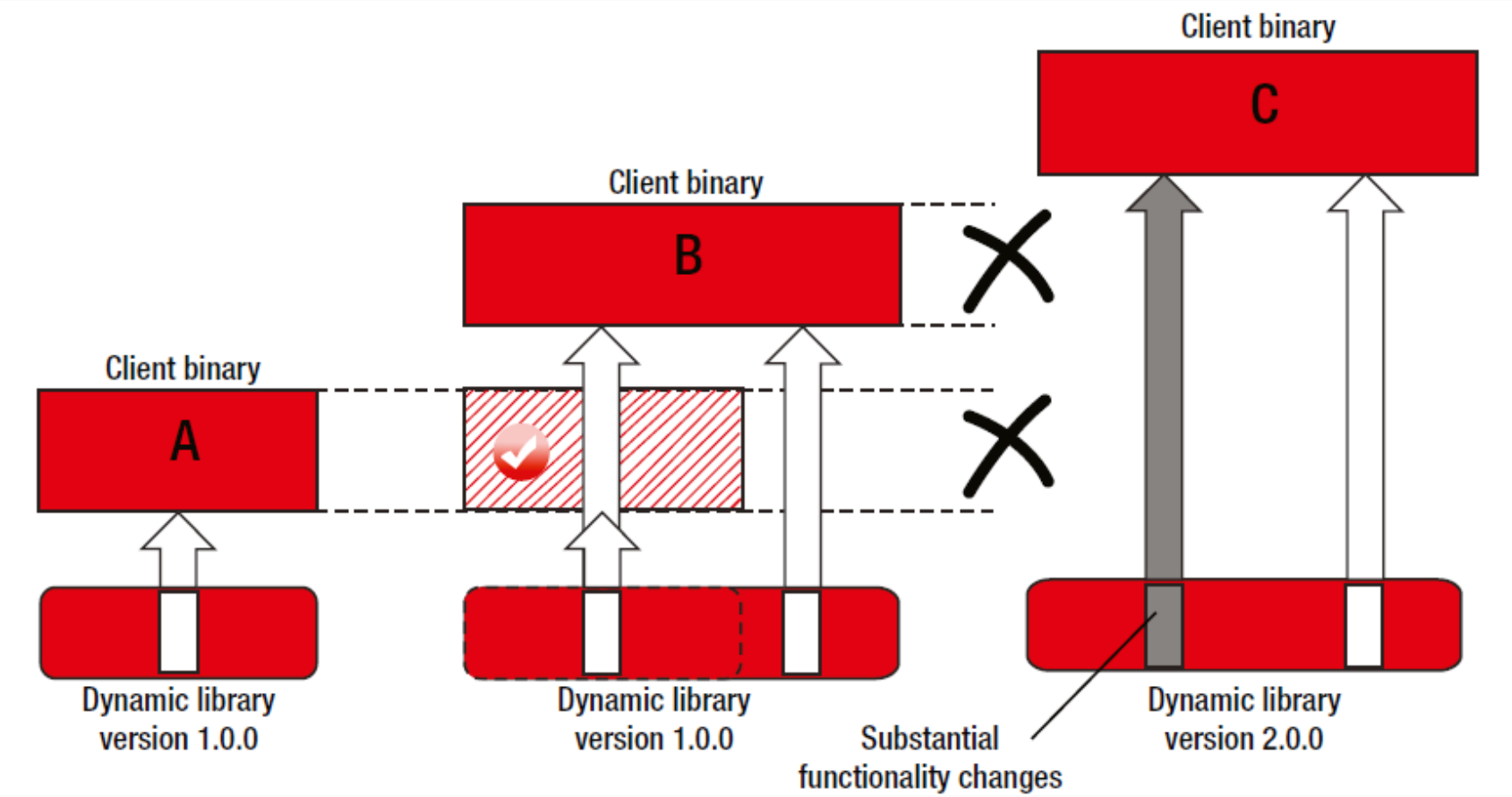


Rysunek: Linkowanie pierwszej wersji z aplikacją



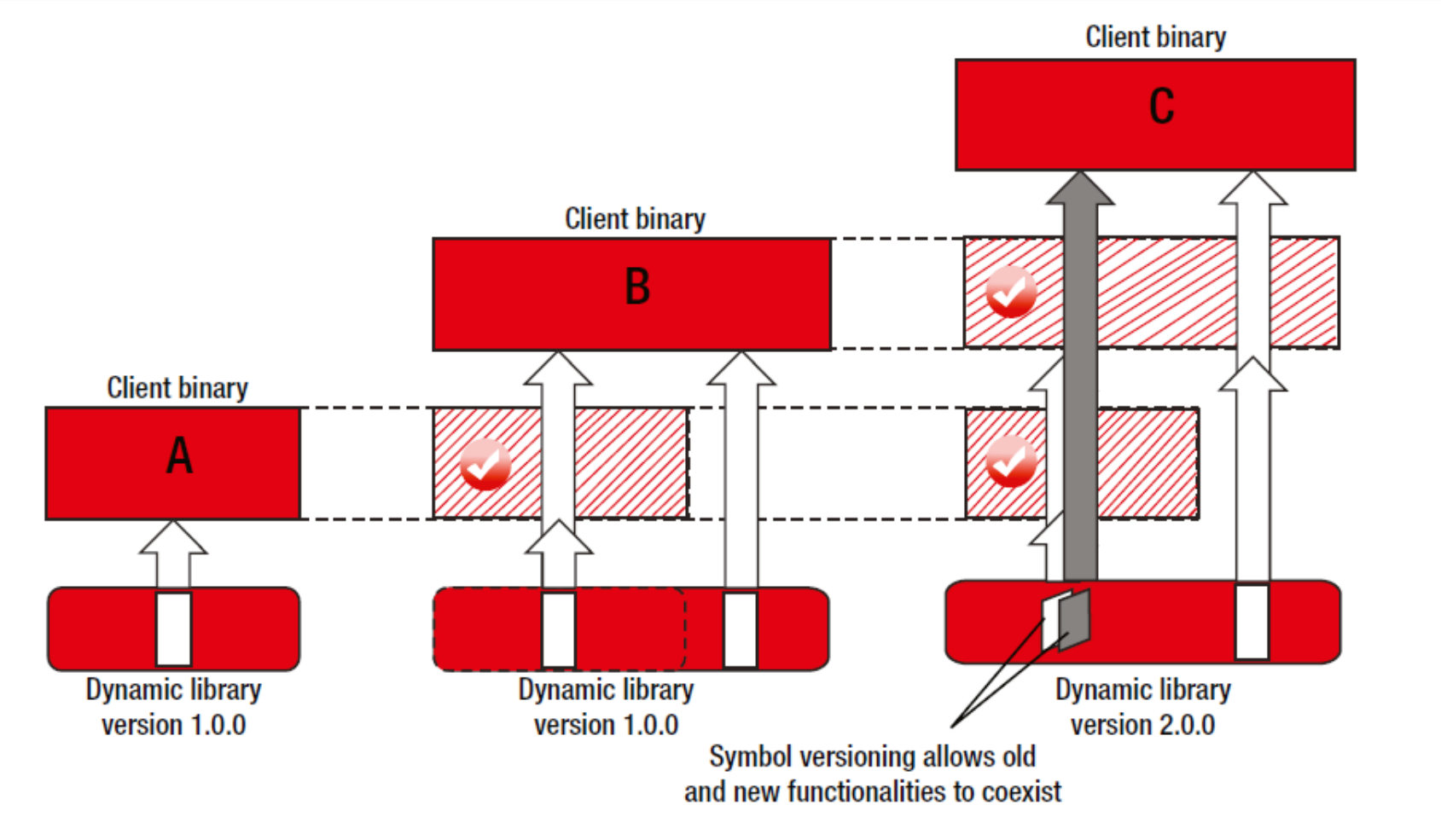
Rysunek: Linkowanie z nową minor wersją

PRZYKŁADOWY SCENARIUSZ WERSJONOWANIA



Rysunek: Linkowanie z nową major wersją, brak kompatybilności

PRZYKŁADOWY SCENARIUSZ WERSJONOWANIA



Rysunek: Koegzystencja symboli z obu major wersji

ŹRÓDŁA

ŹRÓDŁA

- Advanced C and C++ Compiling, An engineering guide to compiling, linking and libraries using C and C++ by Milan Stevanovic
- Learning Linux Binary Analysis by Ryan ”elfmaster” O’Neill
- Tool Interface Standard, Executable and Linking Format Specification
- How To Write Shared Libraries by Ulrich Drepper