Operační systémy

Překlad programu

Petr Krajča

Katedra informatiky
Univerzita Palackého v Olomouci

Překlad programu

1. preprocessor – expanduje makra, odstraní nepotřebný kód, načte požadované hlavičkové soubory (např. `math.h`) – deklarace struktur, deklarace prototypů, atd.
2. překladač – generuje kód v assembleru
3. assembler – vygeneruje objektový kód (`foo.c` $\Longrightarrow$ `foo.obj`/`foo.o`)
4. linker – sloučí několik souborů s objektovým kód + knihovny do spustitelného formátu

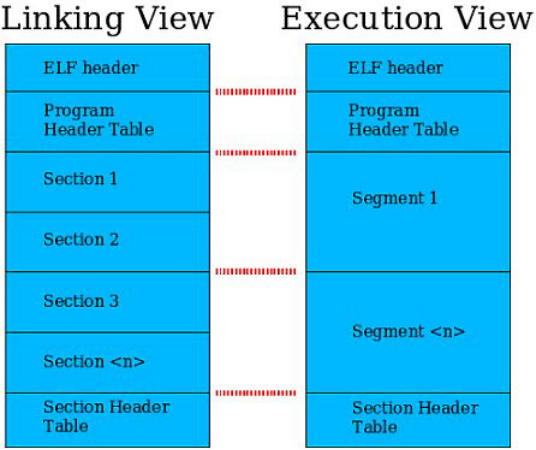
Poznámky

- některé kroky mohou být sloučeny nebo vypuštěny
- některé vyšší programovací jazyky jsou překládany do nižšího jazyka (např. C)
- oddělený překlad do objektových souborů a jejich spojení $\Longrightarrow$
  - možnost kombinovat různé programovací jazyky (různé jazyky, ale stejné objektové soubory)
  - komplikuje interprocedurální optimalizace (není možné optimalizovat napříč zdrojovými soubory)

Objektový soubor

- formát specifický pro každý OS
- obecně obsahuje
  - hlavička – informace o souboru
  - objektový kód – strojový kód + data
  - exportované symboly – seznam poskytovaných symbolů (např. funkce nedeklarované jako `static`)
  - importované symboly – seznam symbolů použitých v tomto souboru
  - informace pro přemístění – seznam míst, které je potřeba upravit v případě přesunutí kódu
  - debugovací informace
- rozdělení na sekce
  - kód
  - data jen pro čtení (konstanty, `const`)
  - inicializovaná data (globální proměnné, statické proměnné)
- navíc obsahuje informace o neinicializovaných datech, apod.
- možnost sdílet jednotlivé části mezi instancemi programu
- formát často sdílený i binárními soubory

Různé pohledy na objektový kód



Linkování (1/2)

- spojí jednotlivé objektové soubory do spustitelného formátu (sloučí jednotlivé sekce)
- postará se o správné umístění kódu a vyřešení odkazů na chybějící funkce a proměnné
- připojení knihoven (hlavičkové soubory většinou neobsahují žádný kód!)

Staticky linkované knihovny 

- archiv objektových souborů (+ informace o symbolech) 
- výhody: jednoduchá implementace, nulová režie při běhu aplikace, žádné závislosti
- nevýhody: velikost výsledného binárního souboru, aktualizace knihovny $\Longrightarrow$ nutnost rekompilace

Dynamicky linkované knihovny

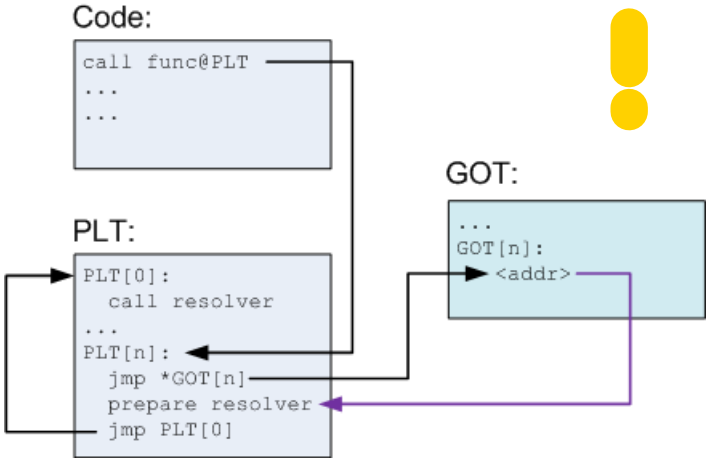
- knihovna je načtena až při spuštění programu
- sdílení kódu mezi programy
- nutnost provázat adresy v kódu s knihovnou
- nutná spolupráce OS

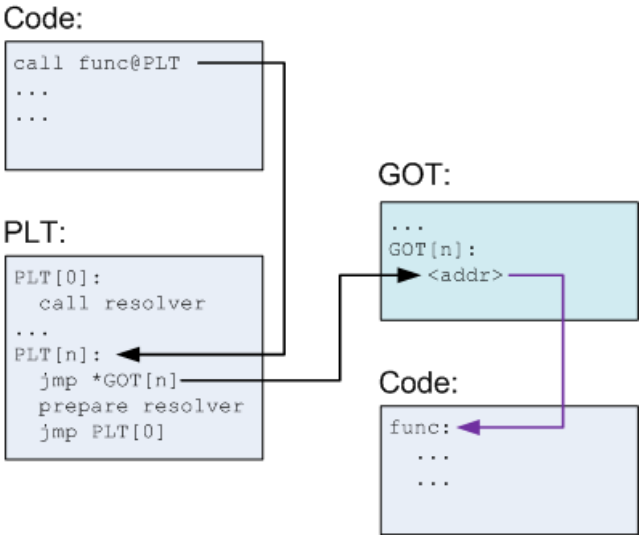
Dynamicky linkované knihovny (1/2)

- problém: umístění knihovny v paměti

Řešení v Unixu

- sdílené knihovny (shared objects, `foo.so`)
- $\Longrightarrow$ position independent code (PIC) – kód, který lze spustit bez ohledu na adresu v paměti
- x86 používá často relativní adresování (i tak PIC pomalejší než běžný kód)
- při spuštění dynamický linker (`ld.so`) provede přenastavení všech odkazů na vnější knihovny
- Global Offset Table (GOT) – tabulka sloužící k výpočtu absolutních adres (nepřímá adresace)
- Procedure Linkage Table (PLT) – tabulka absolutních adres funkcí
  - na začátku PLT obsahuje volání linkeru
  - při volání funkce se provede skok do PLT
  - nastaví se informace o funkci pro linker a ten se zavolá
  - linker najde adresu funkce, nastaví záznam v PLT
  - linker zavolá funkci
  - další volání se provádí bez účasti linkeru $\Longrightarrow$ adresa v PLT


Code:
call func@PLT
...
...
PLT:
PLT[0]:
call resolver
...
PLT[n]:
jmp *GOT[n]
prepare resolver
jmp PLT[0]
GOT:
...
GOT[n]:
<addr>


Code:
call func@PLT
...
...
PLT:
PLT[0]:
call resolver
...
PLT[n]:
jmp *GOT[n]
prepare resolver
jmp PLT[0]
GOT:
...
GOT[n]:
<addr>
Code:
func:
...
...

Dynamicky linkované knihovny (2/2)

Řešení ve Windows

- Dynamic-link library (DLL)
- Windows nepoužívá PIC $\Longrightarrow$ každá knihovna má svou adresu v paměti
- v případě kolize nutnost přesunu + přepočet absolutních adres
- každý program obsahuje *import address table (IAT)* – tabulka adres volaných funkcí (nepřímá adresace)
- inicializace při spuštění
- volání přes `call [adresa]` nebo *thunk table*

```
...kód...
00401002 CALL 00401D82
...thunk table...
00401D82 JMP DWORD PTR DS:[40204C]
...adresy funkcí...
40204C > FC 3D 57 7C   ; adresa
```

- vyhledávání funkcí podle čísla nebo jména (binární vyhledávání)

Dynamicky nahrávané knihovny

- možnost explicitně nahrát knihovnu za běhu
- implementace pluginu
- mechanismus podobný dynamickému linkování
- Unix: dlopen, dlsym (vyhledá funkci podle jména)
- Windows: LoadLibrary, GetProcAddr
- kombinace: zpožděné načítání knihoven

Virtuální stroje

- virtualizace systému vs. virtualizace procesu
- program se nepřekládá do strojového kódu cílového procesoru 
- bytecode: instrukční sada virtuálního procesoru (virtuálního stroje, VM)
- bytecode $\Longrightarrow$ interpretace jednotlivých instrukcí nebo překlad do instrukční sady cílového procesoru běhovým prostředím
- přenositelný kód nezávislý na konkrétním procesoru
- možnost lépe kontrolovat běh kódu (oprávnění, přístupy)
- režie interpretace/překladu
- VM může řešit i komplexnější úlohy než běžný CPU (správa paměti, výjimky, atd.)
- příklady: Java Virtual Machine (& Java Byte Code), Common Language Runtime (& Common Intermediate Language), UCSD Pascal (p-code), LLVM, atd.

JIT překlad

- běhové prostředí generuje kód dané architektury za běhu (von Neumannova architektura!)
- možnost optimalizace pro konkrétní typ CPU
- optimalizace podle aktuálně prováděného kódu (profilování)

Zásobníkové virtuální stroje

- jednoduchá instrukční sada $\Longrightarrow$ snadná implementace
- potřeba více instrukcí, nicméně kratší kód
- JVM, CLR

Registrové virtuální stroje

- efektivní překlad do instrukční sady (pipelined) procesorů
- odolnější proti chybám
- LLVM – optimalizace přes Single Static Assignment; Parrot – Raku (Perl 6); Dalvik – dříve Android, minimální spotřeba paměti

Java Virtual Machine a Java Bytecode (1/2)

- 1995: SUN prog. jazyk Java 1.0
- překlad Java $\Longrightarrow$ Java Bytecode (JBC)
- JBC vykonáván pomocí Java Virtual Machine (JVM)
- implementace JVM není definovaná (pouze specifikuje chování), JBC lze
  - interpretovat
  - přeložit do strojového kódu daného stroje (JIT i AOT)
  - provést pomocí konkrétního CPU
- JVM – virtuální zásobníkový procesor
- malý počet instrukcí (< 256)
- zásobník obsahuje rámce (rámec je vytvořen při zavolání funkce)
  - lokální proměnné, mezivýpočty
  - *operand stack* – slouží k provádění výpočtů
- heap s automatickou správou paměti

Java Virtual Machine a Java Bytecode (2/2)

- jednoduché i velmi komplexní operace (volání funkcí, výjimky)
- základní aritmetika s primitivními datovými typy (hodnoty menší než int převedeny na int)
- speciální operace pro práci s prvními argumenty, lokálními prostředími, jedničkou, nulou
- pouze relativní skoky

Příklad

```
public static void foo(int a, int b) {
    System.out.println(a + b);
}

Code:
0: getstatic       #21; //Field java/lang/System.out:Ljava/io/PrintStream;
3: iload_0
4: iload_1
5: iadd
6: invokevirtual   #27; //Method java/io/PrintStream.println:(I)V
9: return
```

Common Langauge Runtime

- Microsoft .NET implementuje obdobný přístup
- Common Langauge Runtime (CLR) + Common Intermediate Langauge (CIL) – běhové prostředí + bytecode
- koncepčně velice podobné JVM a JBC
- od začátku navržen s podporou více jazyků
- při prvním zavolání metody $\Longrightarrow$ překlad do strojového kódu CPU

Opustění běhového prostředí

- Java: Java Native Interface – rozhraní pro spolupráci s C++
- .NET: Platform Invocation Services (P/Invoke) – umožňuje spoustět kód z DLL