Zad. 8

niedziela, 14 maja 2023 22:08

Zadanie 8. Na podstawie [1, §7.12] opisz proces **leniwego wiązania** (ang. *lazy binding*) symboli. Czym różni się **kod relokowalny** (ang. *Position Independent Code*) skompilowany z opcją «`-fpic`» od kodu nierelokowalnego? Jakie dane przechowują sekcje **procedure linkage table** «`.plt`» i **global offset table** «`.got`»? W jaki sposób korzysta z nich *konsolidator dynamiczny*? Czemu sekcja «`.got`» jest modyfikowalna, a sekcje kodu i «`.plt`» są tylko do odczytu? Jakie są przewagi *leniwego wiązania* nad *gorliwym wiązaniem*?

Figure 7.19(a) shows how the GOT and PLT work together to lazily resolve the run-time address of function addvec the first time it is called:

Step 1. Instead of directly calling addvec, the program calls into PLT[2], which is the PLT entry for addvec.

Step 2. The first PLT instruction does an indirect jump through GOT[4]. Since each GOT entry initially points to the second instruction in its corresponding PLT entry, the indirect jump simply transfers control back to the next instruction in PLT[2].

Step 3. After pushing an ID for addvec (0x1) onto the stack, PLT[2] jumps to PLT[0].

Step 4. PLT[0] pushes an argument for the dynamic linker indirectly through GOT[1] and then jumps into the dynamic linker indirectly through GOT[2]. The dynamic linker uses the two stack entries to determine the run-time location of addvec, overwrites GOT[4] with this address, and passes control to addvec.

Figure 7.19(b) shows the control flow for any subsequent invocations of addvec:

Step 1. Control passes to PLT[2] as before.

Step 2. However, this time the indirect jump through GOT[4] transfers control directly to addvec.

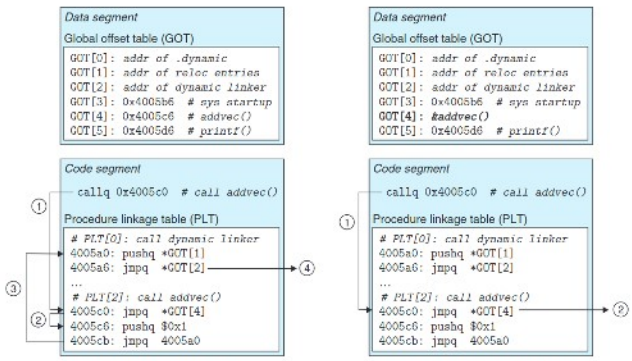


Figure 7.19 Using the PLT and GOT to call external functions. The dynamic linker resolves the address of addvec the first time it is called.

opis procesu

relokowalny vs nierelokowalny

Code that can be loaded without needing any relocations is known as *position-independent code (PIC)*. Users direct GNU compilation systems to generate PIC code with the `-fpic` option to GCC. Shared libraries must always be compiled with this option.

.got

of the data segment. The GOT contains an 8-byte entry for each global data object (procedure or global variable) that is referenced by the object module. The compiler also generates a relocation record for each entry in the GOT. At load time, the dynamic linker relocates each GOT entry so that it contains the absolute address of the object. Each object module that references global objects has its own GOT.

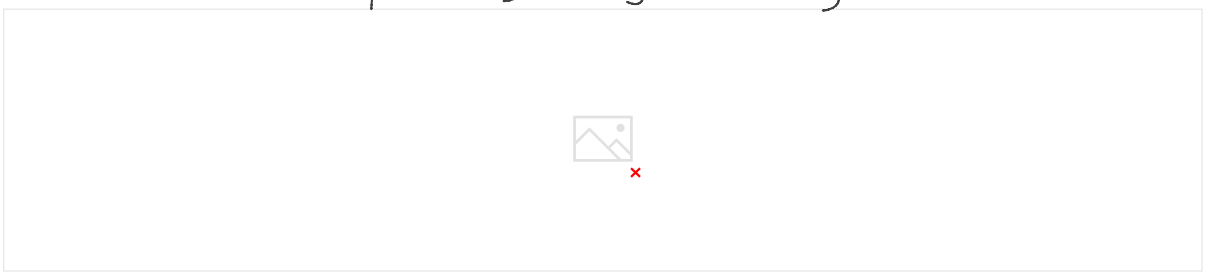
Global offset table (GOT). As we have seen, the GOT is an array of 8-byte address entries. When used in conjunction with the PLT, GOT[0] and GOT[1] contain information that the dynamic linker uses when it resolves function addresses. GOT[2] is the entry point for the dynamic linker in the ld-linux.so module. Each of the remaining entries corresponds to a called function whose address needs to be resolved at run time. Each has a matching PLT entry. For example, GOT[4] and PLT[2] correspond to addvec. Initially, each GOT entry points to the second instruction in the corresponding PLT entry.

.plt

• plt

Procedure linkage table (PLT). The PLT is an array of 16-byte code entries. `PLT[0]` is a special entry that jumps into the dynamic linker. Each shared library function called by the executable has its own PLT entry. Each of

przewagi lazy bindingu



• got jest modyfikowalna, żeby móc w niej zapisać adresy po pierwszym wywołaniu funkcji

• plt jest read-only zapewne ze względów bezpieczeństwa