Architektury systemów komputerowych

Lista zadań nr 7

Na zajęcia 15–16 kwietnia 2019

Zadania z tej listy należy rozwiązywać na komputerze z systemem operacyjnym Linux dla platformy x86-64. Prowadzący zakłada, że zainstalowana dystrybucja będzie bazowała na Debian-ie (np. Ubuntu lub Mint).

Do poniższej listy załączono na stronie przedmiotu pliki źródłowe wraz z plikiem `Makefile`.

UWAGA! W trakcie prezentacji rozwiązań należy zdefiniować i wyjaśnić pojęcia, które zostały oznaczone **wytłuszczoną** czcionką.

Zadanie 1. Poniżej podano zawartość pliku «`swap.c`»:

```
1  extern int buf[];
2
3  int *bufp0 = &buf[0];
4  static int *bufp1;
5
6  static void incr() {
7    static int count = 0;
8    count++;
9  }
10 void swap() {
11   int temp;
12   incr();
13   bufp1 = &buf[1];
14   temp = *bufp0;
15   *bufp0 = *bufp1;
16   *bufp1 = temp;
17 }
```

Wygeneruj **plik relokowalny** «`swap.o`», po czym na podstawie wydruku polecenia «`readelf -t -s`» dla każdego elementu tablicy symboli podaj:

- adres symbolu względem początku sekcji,
- typ symbolu – tj. «`local`», «`global`», «`extern`»,
- rozmiar danych, na które wskazuje symbol,
- numer i nazwę sekcji – tj. «`.text`», «`.data`», «`.bss`» – do której odnosi się symbol.

Co przechowują sekcje «`.strtab`» i «`.shstrtab`»?

Zadanie 2. Rozważmy program składający się z dwóch plików źródłowych:

```
1 /* mismatch-a.c */
2 void p2(void);
3
4 int main(void) {
5   p2();
6   return 0;
7 }
```

```
1 /* mismatch-b.c */
2 #include <stdio.h>
3
4 char main;
5
6 void p2(void) {
7   printf("0x%x\n", main);
8 }
```

Po uruchomieniu program drukuje pewien ciąg znaków i kończy działanie bez zgłoszenia błędu. Czemu tak się dzieje? Skąd pochodzi wydrukowana wartość? Zauważ, że zmienna «`main`» w pliku «`mismatch-a.c`» jest niezainicjowana. Co by się stało, gdybyśmy w funkcji «`p2`» przypisali wartość pod zmienną «`main`»? Co by się zmieniło gdybyśmy w pliku «`mismatch-b.c`» zainicjowali zmienną «`main`» w miejscu jej definicji?

Zadanie 3. Zapoznaj się z narzędziami do analizy **plików relokowalnych** w formacie `ELF` i bibliotek statycznych, tj. «`objdump`», «`readelf`» i «`ar`»; a następnie odpowiedz na następujące pytania:

1. Ile plików zawierają biblioteki «`libc.a`» i «`libm.a`» (katalog «`/usr/lib/x86_64-linux-gnu`»)?
2. Czy wywołanie kompilatora z opcją «`-Og`» generuje inny kod wykonywalny niż «`-O0 -g`»?
3. Z jakich bibliotek współdzielonych korzysta interpreter języka «`Python`» (plik «`/usr/bin/python`»)?

Zaprezentuj w jaki sposób można dojść do odpowiedzi korzystając z podanych poleceń.

Zadanie 4. Rozważmy program składający się z dwóch plików źródłowych:

```c
/* str-a.c */
#include <stdio.h>

char *somestr(void);

int main(void) {
  char *s = somestr();
  s[5] = '\0';
  puts(s);
  return 0;
}
```

```c
/* str-b.c */
char *somestr(void) {
  return "Hello, world!";
}
```

Po uruchomieniu program kończy się z błędem dostępu do pamięci. Dlaczego? Gdzie została umieszczona stała znakowa `"Hello, world!"`? Popraw ten program tak, by się poprawnie zakończył. Gdzie został umieszczony ciąg znaków po poprawce? Nie wolno modyfikować sygnatury procedury «`somestr`» i pliku «`str-a.c`», ani korzystać z dodatkowych procedur.

Zadanie 5. Posiłkując się narzędziem «`objdump`» podaj rozmiary sekcji «`.data`» i «`.bss`» plików «`bar.o`» i «`foo.o`». Wskaż rozmiar i pozycje symboli względem początków odpowiednich sekcji. Wyjaśnij znaczenie opcji «`-fno-common`» przekazywanej do kompilatora.

```c
/* bar.c */
int bar = 42;
short dead[15];
```

```c
/* foo.c */
long foo = 19;
char code[17];
```

Na czym polega **częściowa konsolidacja** z użyciem opcji «`-r`» do polecenia «`ld`»? Czym różni się sposób wygenerowania plików «`merge-1.o`» i «`merge-2.o`»? Na podstawie **mapy konsolidacji** porównaj pozycje symboli i rozmiary sekcji w plikach wynikowych. Z czego wynikają różnice skoro konsolidator nie dysponuje informacjami o typach języka C?

Zadanie 6. Plik wykonywalny powstały w wyniku kompilacji poniższych plików źródłowych powinien być nie dłuższy niż 1KiB. Na podstawie **nagłówka pliku ELF** wskaż w zdeasemblowanym pierwszą instrukcję, którą wykona procesor po wejściu do programu. Na podstawie **nagłówków programu** wskaż pod jaki adres wirtualny zostanie załadowany **segment** z sekcją «`.text`».

```c
/* start.c */
int is_even(long);

void _start(void) {
  asm volatile(
    "syscall"
    : /* no output */
    : "a" (0x3c /* exit */),
      "D" (is_even(42)));
}
```

```c
/* even.c */
int is_odd(long n);

int is_even(long n) {
  if (n == 0)
    return 1;
  else
    return is_odd(n - 1);
}
```

```c
/* odd.c */
int is_even(long n);

int is_odd(long n) {
  if (n == 0)
    return 0;
  else
    return is_even(n - 1);
}
```

Wskaż w kodzie źródłowym miejsca występowania **relokacji**. Zidentyfikuj je w wydruku polecenia «`objdump -r -d`», po czym pokaż jak zostały wypełnione w pliku wykonywalnym. Na podstawie rozdziału §7.7 podręcznika „Computer Systems: A Programmer's Perspective" zreferuj algorytm relokacji. Wydrukuj tabele relokacji plików relokowalnych, fragment mapy konsolidacji opisujący złączoną sekcję «`.text`», po czym oblicz ręcznie wartości, które należy wpisać w miejsce relokacji.

Zadanie 7. W trakcie tłumaczenia poniższego kodu na asembler kompilator umieścił tablicę skoków dla instrukcji przełączania w sekcji «.rodata».

```
1  int relo3(int val) {
2    switch (val) {
3      case 100:
4        return val;
5      case 101:
6        return val + 1;
7      case 103:
8      case 104:
9        return val + 3;
10     case 105:
11       return val + 5;
12     default:
13       return val + 6;
14   }
15 }
```

```
0000000000000000 <relo3>:
 0: 8d 47 9c              lea    -0x64(%rdi),%eax
 3: 83 f8 05              cmp    $0x5,%eax
 6: 77 15                 ja     1d <relo3+0x1d>
 8: 89 c0                 mov    %eax,%eax
 a: ff 24 c5 00 00 00 00  jmpq   *0x0(,%rax,8)
11: 8d 47 01              lea    0x1(%rdi),%eax
14: c3                    retq
15: 8d 47 03              lea    0x3(%rdi),%eax
18: c3                    retq
19: 8d 47 05              lea    0x5(%rdi),%eax
1c: c3                    retq
1d: 8d 47 06              lea    0x6(%rdi),%eax
20: c3                    retq
21: 89 f8                 mov    %edi,%eax
23: c3                    retq
```

Dla sekcji «.text» i «.rodata» określ pod jakimi miejscami znajdują się relokacje, a następnie podaj zawartość tablicy relokacji «.rela.text» i «.rela.rodata», tj. listę rekordów składających się z:

- przesunięcia relokacji względem początku sekcji,
- typu relokacji,
- nazwy symbolu i przesunięcia względem początku symbolu.

W wyniku konsolidacji pliku wykonywalnego zawierającego procedurę «relo3», została ona umieszczona pod adresem 0x1000, a tablica skoków pod 0x2000. Oblicz wartości, które należy wstawić w miejsca relokacji.

Zadanie 8. Na podstawie rozdziału §7.12 podręcznika „Computer Systems: A Programmer's Perspective" opisz proces **leniwego wiązania** (ang. *lazy binding*) symboli i odpowiedz na następujące pytania:

- Czym charakteryzuje się **kod relokowalny** (ang. *Position Independent Code*)?
- Do czego służą sekcje «.plt» i «.got» – jakie dane są tam przechowywane?
- Czemu sekcja «.got» jest modyfikowalna, a sekcje kodu i «.plt» są tylko do odczytu?
- Co znajduje się w sekcji «.dynamic»?

Zaprezentuj leniwe wiązanie na podstawie programu «lazy». Uruchom go pod kontrolą debuggera GDB, ustaw punkty wstrzymań (ang. *breakpoint*) na linię 4 i 5. Po czym wykonując krokowo program (stepi) pokaż, że gdy procesor skacze do adresu procedury «puts» zapisanego w «.got» – za pierwszym wywołaniem jest tam przechowywany inny adres niż za drugim.

Wskazówka: Wykorzystaj rysunek 7.19. Dobrze jest zainstalować sobie GDB dashboard[1].

[1] https://github.com/cyrus-and/gdb-dashboard