Systemy komputerowe

Lista zadań nr 5

Karolina Szlęk 300411

Zadanie 1

Zadanie 1. Zdefiniuj pojęcie **wyrównania danych** w pamięci (ang. *data alignment*). Dlaczego dane typów prostych (np. `short`, `int`, `double`) powinny być w pamięci odpowiednio wyrównane? Jaki związek z wyrównywaniem danych mają **wypełnienia** (ang. *padding*) w danych typu strukturalnego. Odpowiadając na powyższe pytanie podaj przykład struktury, której rozmiar w bajtach (wyliczony przez operator `sizeof`) jest większy niż suma rozmiaru pól składowych. Czemu służy **wypełnienie wewnętrzne** (ang. *internal padding*) a czemu **wypełnienie zewnętrzne** (ang. *external padding*)?

Wyrównania danych w pamięci (ang.data alignment) - polega na tym, że dane w pamięci muszą być na adresie podzielnym przez typ reprezentowany przez te dane.
Na przykład mamy zmienną int x. Musi być ona na adresie podzielnym przez 4.

Chcemy umieścić w określonym miejscu w pamięci jakiś typ. Jeśli to miejsce nie jest podzielne, przez typ, który chcemy tam umieścić, to musimy dodać **wypełnienie (ang.padding)**.

Wypełnienie (ang.pading) - przerwa w pamięci, która nic nie zawiera.

Pading w naszym przypadku jest potrzebny, żeby adres był podzielny.
Teraz przejdźmy do **internal padding** i **external padding**.
Do prawidłowego wyrównania danych stosuje się tak zwane wypełnienia w danych typu strukturalnego. Dzielą się one na wypełnienie **wewnętrzne** (**internal**) i **zewnętrzne** (**external)**.

W przypadku structów, struct stworzony musi być na adresie podzielnym przez największy typ, który on w sobie znajduje – to właśnie **wypełnienie wewnętrzne(ang.internal padding)** oraz jego długość musi być podzielna przez ten największy typ – to właśnie **wypełnienie zewnętrzne(ang.external padding)**.

Przykład struktury, która ma większy obliczony sizeof, niż suma rozmiaru pól składowych.

```
struct S0{
        char c;
       int32_t i;
} ;
```

Zrobiliśmy wypełnienie (padding) po zmiennej c, więc:
sizeof(S0) == 8

Natomiast suma rozmiaru składowych to:
sizeof(char) + sizeof(int32_t) == 5

Zadanie 2

Zadanie 2. Dana jest funkcja o sygnaturze «int16_t you(int8_t index)» i fragmencie kodu podanym poniżej. Funkcja ta została skompilowana z flagą -O0, a jej kod asemblerowy również jest podany. Nieznana jest natomiast funkcja «int16_t set_secret(void)». Jaki argument należy podać wywołując you, by odkryć wartość sekretu?

```
1  int16_t you(int8_t index) {
2  struct {
3      int16_t tbl[5];
4      int8_t tmp;
5      int16_t secret;
6  } str;
7
8  str.secret = set_secret();
9  ...
10 return str.tbl[index];
11 }
```

```
4  you:    pushq   %rbp
5          movq    %rsp, %rbp
6          subq    $24, %rsp
7          movl    %edi, %eax
8          movb    %al, -20(%rbp)
9          call    set_secret
10         movw    %ax, -2(%rbp)
11         ...
12         movsbl  -20(%rbp), %eax
13         cltq
14         movzwl  -14(%rbp,%rax,2), %eax
15         leave
16         ret
```

Wskazówka: Instrukcja cltq rozszerza rejestr %eax do %rax zachowując znak. Pamiętaj, że zadeklarowane zmienne muszą być odpowiednio wyrównane.

```
int16_t you(int8_t index) {
   struct {
      int16_t tbl[5];        // 2*5 =    10 bajtów
      int8_t tmp;            // 1 bajt
                             // padding 1 bajt
      int16_t secret;        // 2 bajty
   } str;

   str.secret = set_secret();
    ...
   return str.tbl[index];
}
```

```
you:    pushq %rbp
        movq %rsp, %rbp
        subq $24, %rsp
        movl %edi, %eax
        movb %al, -20(%rbp)          // tu wrzucamy index
        call set_secret
        movw %ax, -2(%rbp)           // zapisujemy wynik set_secret we właściwym miejscu w
strukturze
        // ...
        movsbl -20(%rbp), %eax       // wrzucamy index do rax'a
        cltq
        movzwl -14(%rbp, %rax,2), %eax // zwracamy [rbp-14+rax*2]
        leave
        ret
```

Z 14 linijki wiemy, że funkcja you zwróci nam to, co znajduje się w -14(%rbp, %rax, 2).
W %rax umieszczony został index.
Dodatkowo wiemy, że w linijce 10 otrzymaną wartość secret umieszczamy w -2(%rbp).
Więc jeżeli chcemy zwrócić secret musi zajść równanie:

%rbp-14+%rax*2 = r%bp-2,

stąd %rax=6.

Zatem funkcji you musimy podać argument 6.

Zadanie 3

Zadanie 3 (2pkty). Przeczytaj poniższy kod w języku C i odpowiadający mu kod w asemblerze, a następnie wywnioskuj jakie są wartości stałych «A» i «B».

Zadanie 3 (2pkty). Przeczytaj poniższy kod w języku C i odpowiadający mu kod w asemblerze, a następnie wywnioskuj jakie są wartości stałych «A» i «B».

```
1  typedef struct {
2    int32_t x[A][B];
3    int64_t y;
4  } str1;
5
6  typedef struct {
7    int8_t array[B];
8    int32_t t;
9    int16_t s[A];
10   int64_t u;
11 } str2;
12
13 void set_val(str1 *p, str2 *q) {
14   int64_t v1 = q->t;
15   int64_t v2 = q->u;
16   p->y = v1 + v2;
17 }
```

```
18 set_val:
19   movslq 8(%rsi),%rax
20   addq   32(%rsi),%rax
21   movq   %rax,184(%rdi)
22   ret
```

Wskazówka: Deklaracja int32_t x[A][B] powoduje, że x będzie A-elementową tablicą wartości typu int32_t [B]. Pamiętaj o wyrównaniu pól w strukturach.

```
typedef struct {
   int32_t x[A][B];       // A * B * 4
                          // Jeśli B*A % 2≠0, to dodajemy padding 4
   int64_t y;             // 8
} str1;


typedef struct {
   int8_t array[B];       // B
                          // Jeśli B % 8≠0, to dodajemy odpowiedni padding  ze zbioru {1, 2,
3}
   int32_t t;             // 4
                          // brak paddingu, ponieważ następna struktura jest mniejsza
   int16_t s[A];          // 2 * A
                          // Jeśli A % 4≠0, to dodajemy odpowiedni padding  ze zbioru {2, 4, 6}
   int64_t u;             // 8
} str2;
```

```
void set_val(str1 *p, str2 *q) {
   int64_t v1 = q->t;
   int64_t v2 = q->u;
   p->y = v1 + v2;
}
set_val:
   movslq 8(%rsi),%rax
   addq 32(%rsi),%rax
   movq %rax,184(%rdi)
   ret
```

Aby odwołać się do zmiennej t w strukturze q, w funkcji set_val, przesuwamy się o 8 od początku q. Więc B może być co najwyżej 8 (jeżeli będzie mniejsze niż 8, to za tablica array będzie paddingB). Odwołujemy się do u i przesuwamy się o 32. Padding może być też za tablicą s w strukturze q – paddingA.
Wiemy, że B + paddingB = 8

Zapiszmy więc

2 * A + paddingA = 20

Tylko jeżeli 2*A nie wypadnie na miejscu podzielnym przez 8, to paddingA będzie niezerowy. Skoro t zajmuje 4 bajty i zaczyna się na miejscu podzielnym przez 4, to s również zaczyna się na miejscu podzielnym przez 4. Za tablicą s może być potrzebny padding 2 lub 4 lub 6 (bo każdy element tablicy s spowoduje przesunięcie o 2). Również za tablicą x może być potrzebny padding. Daje nam to:

A * B * 4 + 4 = 184
Lub
A * B * 4 = 184

Rozpatrzmy więc przypadki:

I) paddingA = 2
A =9
i z równania 9*B*4*4=184 dostajemy
B =5

II) paddingA = 4
A = 8

Zarówno jak jest padding jak i wtedy, gdy go nie ma, B nie wychodzi całkowite.
A nie może być równe 8.

III) paddingA = 6
A = 7
7*B*4+4=184
7 * B * 4 = 184
Z żadnego z równań, nie dostajemy B całkowitego.
A nie może być równe 7.

Odp.: Rozwiązaniem są A=9 i B=5

Zadanie 4

Zadanie 4 (2pkty). Przeczytaj poniższy kod w języku C i odpowiadający mu kod w asemblerze, a następnie wywnioskuj jaka jest wartość stałej «CNT» i jak wygląda definicja struktury «a_struct».

```
1  typedef struct {
2    int32_t first;
3    a_struct a[CNT];
4    int32_t last;
5  } b_struct;
6
7  void test (int64_t i, b_struct *bp) {
8    int32_t n = bp->first + bp->last;
9    a_struct *ap = &bp->a[i];
10   ap->x[ap->idx] = n;
11 }
```

```
1  test:
2  movl   0x120(%rsi),%ecx
3  addl   (%rsi),%ecx
4  leaq   (%rdi,%rdi,4),%rax
5  leaq   (%rsi,%rax,8),%rax
6  movq   0x8(%rax),%rdx
7  movslq %ecx,%rcx
8  movq   %rcx,0x10(%rax,%rdx,8)
9  retq
```

Wiadomo, że jedynymi polami w strukturze «a_struct» są «idx» oraz «x», i że obydwa te pola są typu numerycznego ze znakiem.

```
typedef struct {
   int32_t first;              // 4      bajty
                               // padding, ponieważ a_struct może być większe niż int32_t
   a_struct a[CNT];            // sizeof a_struct * CNT
   int32_t last;               // 4      bajty
} b_struct;


void test (int64_t i, b_struct *bp) {
   int32_t n = bp->first + bp->last;
   a_struct *ap = &bp->a[i];
   ap->x[ap->idx] = n;
```

```
}
test:
   movl 0x120(%rsi),%ecx        // 288
   addl (%rsi),%ecx             // ecx = n, a n = bp.first + bp.last
   leaq (%rdi,%rdi,4),%rax      // i + 4*i
   leaq (%rsi,%rax,8),%rax      // 8*5i czyli i-te miejsce w tablicy a
   movq 0x8(%rax),%rdx          // rdx = ap->idx
   movslq %ecx,%rcx             //rcx = n
   movq %rcx,0x10(%rax,%rdx,8)          // 0x10 = 16 => 8 bajtów od b_struct->first + 8
bajtów od
                                          a_struct->idx i jesteśmy na początku idx => %rdx
* 8
                                          to jest x[ap->idx], czyli idx ma 8 bajtów
   retq
```

Widzimy że tablica a kończy się na 288 pozycji.
Odejmujemy 8 by ominąć first i $początkowy padding. Dostajemy 280.
Ponadto wiemy, że a_struct ma 40 bajtów, dlatego dzielimy 280/40. Wtedy uzyskamy wartość CNT. CNT = 280/40 = 7.

Definicja struktury <<a_struct>>:

```
typedef struct {
   int64_t idx;
   int64_t x[4];
} a_struct;
```

Zadanie 5

Zadanie 5. Zdefiniuj semantykę operatora «?:» z języka C. Jakie zastosowanie ma poniższa funkcja.

```
1 int32_t cread(int32_t *xp) {
2   return (xp ? *xp : 0);
3 }
```

Używając serwisu `godbolt.org` (kompilator `x86-64 gcc 8.2`) sprawdź, czy istnieje taki poziom optymalizacji (`-O0`, `-O1`, `-O2` lub `-O3`), że wygenerowany dla `cread` kod asemblerowy nie używa instrukcji skoku. Jeśli nie, to zmodyfikuj funkcję `cread` tak, by jej tłumaczenie na asembler spełniało powyższy warunek.

Wskazówka: Dążysz do wygenerowania kodu używającego instrukcji cmov. Końcowej instrukcji ret nie uważamy w tym zadaniu za instrukcję skoku.

Taka optymalizacja nie istnieje.

Nowy kod:

```
int32_t cread(int32_t *xp) {
        int32_t z = 0;
        int32_t *zp = &z;
        int32_t *res = (xp ? xp : zp);
        return *res; // zwrócimy wartość pod xp lub zp
}
```

Kod assemblerowy przy –O1 (wygenerowany)

cread:

```
movl    $0, -4(%rsp)
leaq    -4(%rsp), %rax
testq   %rdi, %rdi
cmove   %rax, %rdi     //ustaw %rdi=0, gdy ZF wyżej ustawione
movl    (%rdi), %eax
ret
```

Zadanie 7

Zadanie 7. W poniższej funkcji zmienna `field` jest polem bitowym typu `int32_t` o rozmiarze 4. Jaką wartość wypisze ta funkcja na ekranie i dlaczego? Co się stanie, gdy zmienimy typ pola `field` na `uint32_t`? Na obydwa te pytania odpowiedz analizując tłumaczenia tej funkcji na język asemblera.

```
1 void set_field(void) {
2 struct {
3   int32_t field : 4;
4 } s;
5   s.field = 10;
6   printf("Field value is: %d\n", s.field);
7 }
```

Wskazówka: Użyj poziomu optymalizacji «-O0». Dla wyższych poziomów optymalizacji kompilator zauważy, że deklaracja zmiennej «s» jest niepotrzebna i obliczy wartość wypisywaną przez «printf» podczas kompilacji.

10 zapisane binarnie to 1010, a w polu bitowym field bierzemy tylko 4 bity z prawej

strony, zatem mając typ ze znakiem int32_t będziemy mieli w s.field 1010 = -8 + 2 = -6. Zmieniając typ na uint32_t będziemy mieli 10, czyli tak jak chcemy.

Pole bitowe oznacza, że zamiast całej wartości bierzemy tylko jej n bitów, czyli wartość jest wtedy and-owana z maską bitową skladającą się z n jedynek i interpretowana jako liczba n-bitowa.

(10 binarnie to 1010)

Dla int32_t: 1010 = -8 + 2 = -6

Dla uint32_t: 1010 = 10

Dla int32_t:

```
.LC0:
        .string "Field value is: %d\n"
set_field:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movzbl  -4(%rbp), %eax      // zaznaczamy że na tym miejscu będzie eax
        andl    $-16, %eax          // -16 = 10000 zerujemy eax
        orl     $10, %eax           // 10 = 1010 wpisujemy do eax 1010
        movb    %al, -4(%rbp)       // al = najmłodszy bajt rax(eax) wpisz do -4(%rbp)
        movzbl  -4(%rbp), %eax      // wpisz -4(%rbp) do eax, eax wygląda 000...0001010
        sall    $4, %eax            // left shift o 4 bity da nam same 0 w eax, a te bity pójdą
                                       w rax, eax wygląda 00...0 10100000
        sarb    $4, %al             // right arithmetic shift (al to 1111)
        movsbl  %al, %eax           // zapisujemy al do eax, tak, że eax wygląda
                                       11111...111111010
        movl    %eax, %esi          // wpisujemy eax do esi, bo esi to część rsi a rsi to drugi
                                       argument
        movl    $.LC0, %edi         // wpisujemy string do pierwszego argumentu
```

```
    movl    $0, %eax
    call    printf
    nop
    leave
    ret
```

Dla uint32_t:

```
.LC0:
    .string "Field value is: %d\n"
set_field:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movzbl  -4(%rbp), %eax
    andl    $-16, %eax
    orl     $10, %eax
    movb    %al, -4(%rbp)
    movzbl  -4(%rbp), %eax
    andl    $15, %eax                   // and z 000...0 00001111
    movzbl  %al, %eax                   // wrzucamy %al bez znaku do %eax
    movl    %eax, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    nop
    leave
    ret
```

Zadanie 8

Zadanie 8. Język C dopuszcza deklaracje tablic wielowymiarowych z opuszczonym rozmiarem pierwszego wymiaru. Taka deklaracja może wystąpić w nagłówku funkcji, np. «`void process(int32_t A[][77], size_t len)`». Nie można natomiast opuszczać rozmiarów innych wymiarów, np. «`void bad(int32_t A[77][], size_t len)`» nie jest poprawną deklaracją. Wyjaśnij, dlaczego tak jest odwołując się do sposobu, w jaki kompilator tłumaczy odwołania do tablic z C na asembler.

A[SIZE1][SIZE2] oznacza, że mamy SIZE1 segmentów, każdy o długości SIZE2 (razy rozmiar typu tablicy...).

Mając A[][SIZE] znamy rozmiar segmentów, więc wiemy o ile bajtów trzeba przesunąć wskaźnik z A do A[x][y]: (x*SIZE+y).

Natomiast mając A[SIZE][] nie wiadomo o ile należy się przesuwać, bo nie wiadomo jaki rozmiar ma każdy segment.
Zatem void process(int32_t A[][77], size_t len) jest dobre. Kompilator odczyta, że jest to jakaś liczba tablic typu int32_t o wymiarach 77. Ale już void bad(int32_t A[77][], size_t len) nie będzie poprawne. Kompilator nie będzie wiedział o ile mamy się przesuwać.