CSci 2021, Fall 2018   Written Exercise Set 3

Problems and solutions version 2

Problem 1: Arrays in Assembly

In this problem, the C code for a main function that calls the function `matrix_calculation` is provided below. The function `matrix_calculation` has its assembly given below that. Based on the output we have given you from `main`, it is your job to figure out the values you can fill in for `a`, `b`, and `c` in `my_span` that lead to that output being produced. You can write your answer as "a = value, b = value, …". In the process of figuring this out, you may find it helpful to write C code for `matrix_calculation`. Writing C code is not required, and you can get full credit for just the correct values, but if you turn in C code for `matrix_calculation`, it will increase the possibility of getting partial credit.

```
int main(int argc, char **argv) {
  int my_span[3][3] = {{1,1,1},{a,b,-5},{2,11,c}};

  int result_val = matrix_calculation(my_span, 1, 2, 3);

  printf("result_val is %d\n", result_val);
}

matrix_calculation:
        pushq   %rbx
        movl    $0, %r10d
        movl    $0, %eax
        jmp     .L2
.L3:
        movslq  %esi, %r8
        leaq    (%r8,%r8,2), %r9
        leaq    0(,%r9,4), %r8
        addq    %rdi, %r8
        movslq  %r10d, %r11
        movslq  %edx, %r9
        leaq    (%r9,%r9,2), %rbx
        leaq    0(,%rbx,4), %r9
        addq    %rdi, %r9
        movl    (%r9,%r11,4), %r9d
        imull   (%r8,%r11,4), %r9d
        addl    %r9d, %eax
        addl    $1, %r10d
.L2:
        cmpl    %ecx, %r10d
        jl      .L3
        popq    %rbx
        ret
```

The given output for main() is: `result_val is 42`

Problem 1: Solution

The original C code that gave the above assembly is:

```
int matrix_calculation(int mat[3][3], int vec1, int vec2, int num_col){
  int dot_product = 0;

  for(int j = 0; j < num_col; j++){
    dot_product += mat[vec1][j]*mat[vec2][j];
  }

  return dot_product;
}
```

Here is a walk-through of what each instruction does, using the variable names from the C code above.

```
matrix_calculation:
        pushq   %rbx        # save caller's rbx on stack
        movl    $0, %r10d  #  int j = 0
        movl    $0, %eax   #  dot_product = 0
        jmp     .L2        # jump to loop condition
.L3:
        movslq  %esi, %r8           # r8 = vec1 = 2
        leaq    (%r8,%r8,2), %r9    # r9 = r8 + r8*2 = 6
        leaq    0(,%r9,4), %r8      # r8 = r9*4 = 24
        addq    %rdi, %r8           # r8 = r8 + rdi = mat[vec1]
        movslq  %r10d, %r11         # r11 = j
        movslq  %edx, %r9           # r9 = vec2 = 3
        leaq    (%r9,%r9,2), %rbx   # rbx = r9 + r9*2 = 9
        leaq    0(,%rbx,4), %r9     # r9 = rbx*4 = 36
        addq    %rdi, %r9           # r9 = rdi + r9 = mat[vec2]
        movl    (%r9,%r11,4), %r9d  # r9d = r9+r11*4 = mat[vec2][j]
        imull   (%r8,%r11,4), %r9d  # r9d = r9d*(r8+r11*4) = r9d*mat[vec1][j]
        addl    %r9d, %eax          # add r9d to dot_product
        addl    $1, %r10d           # increment j
.L2:
        cmpl    %ecx, %r10d   # num_col is argument rcx
        jl      .L3           # jump to loop if less than
        popq    %rbx          # restore caller's rbc
        ret                   # return, return value in rax
```

Once we see what matrix_calculation is doing, we get a better understanding of how to find a, b, and c. Comparing the function's computation with our desired output gives the equation $a*2+b*11+c*-5 = 42$. Since this is one equation with three unknowns, there are many possible solutions, even given the constraint that the variables are integers. A simple approach is to try choosing small values for the variables and simplifying, as long as we avoid an equation with divisibility problems. For instance we can set b to 0 giving $a*2+c*-5 = 42$. Then setting a to 0 would leave $c*-5 = 42$ which is not solvable for integer c, but the next simplest choice is $a = 1$, leaving $c*-5 = 40$. This means our final answer will be a = 1, b = 0, and c = -8. By making different guesses you could have found other equally good solutions.

Problem 2: Translate to Y86-64

The given C code is for a function that checks if all the elements of a linked list add up to 100.

```
struct linked_list{
    long val;
    struct linked_list *next;
};

int sum_to_100(struct linked_list *start){
    struct linked_list *current_node = start;
    long total = 0;
    while( current_node != NULL ){
        total += current_node->val;
        current_node = current_node->next;
    }
    if( total == 100 ){
        return 1;
    }
    else{
        return 0;
    }
}
```

The x86-64 assembly given below was compiled with GCC from the C code given above.

```
sum_to_100:
        movq        $0, %rax
        jmp         .L2
.L3:
        addq        (%rdi), %rax
        movq        8(%rdi), %rdi
.L2:
        testq        %rdi, %rdi
        jne         .L3
        cmpq        $100, %rax
        jne         .L5
        movq        $1, %rax
        ret
.L5:
        movq        $0, %rax
        ret
```

Write Y86-64 code that matches the given C and x86-64. You should mostly be able to translate instruction by instruction, but some x86-64 instructions will need to be implemented by multiple Y86-64 instructions.

Problem 2: Solution

The Y86-64 that matches the assembly and C code from the previous page is given below:

```
sum_to_100:
LFB0:
        irmovq    $0, %rax       # movq  $0, %rax
        jmp       L2             # jmp L2
L3:
        mrmovq    (%rdi), %rcx   # addq  (%rdi), %rax
        addq      %rcx, %rax
        mrmovq    8(%rdi), %rdi  # movq  8(%rdi), %rdi
L2:
        andq      %rdi, %rdi     # testq %rdi, %rdi
        jne       L3             # jne L3
        irmovq    $100, %r9      # cmpq  $100, %rax
        subq      %r9, %rax
        jne       L5             # jne  L5
        irmovq    $1, %rax       # movq  $1, %rax (2)
        ret                      # ret
L5:
        irmovq    $0, %rax       # movq  $0, %rax (2)
        ret                      # ret
```

The main difference between the X86-64 and Y86-64 is the need to specify the operand types of moves, breaking single moves in X86-64 to multiple moves in Y86-64, and the lack of a cmpX or testX command.

Problem 3: Structs in Assembly (Based on practice problem 3.44)

Answer the following questions for each given struct, according to the x86-64 rules for structure layout and alignment: **(a)** What is the size of the struct in bytes? **(b)** What is the offset of the third field (always named `t`)? **(c)** What is the required alignment of the struct? **(d)** Is there a way to order the fields such that fewer bytes are wasted due to alignment? If so, give an order that has the smallest possible size.

```
struct P1 {char c; int i; char t; int j;};

struct P2 {long l; char *p; int t; short s;};

struct P3 {int i[3]; char c[3]; short t[2];};

struct P4 {char c; int i; double *t; short s; float f[2];}
```

Draw the layout of the struct P4 as a sequence of bytes in memory. Draw each byte labeled by the letter of the variable that the byte is part of, or write X for unused bytes. An example is given below:

```
struct P{int i; char c[2];};

  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
  i   i   i   i   c   c   X   X
+---+---+---+---+---+---+---+---+
```

Notes on drawing the struct:

Make sure to include bytes that are left unused due to alignment. Also, notice how the variable c, an array, is only represented in the total number of bytes, not distinguishing between the elements within the array. Lastly, please use the variable names given in the struct above.

Problem 3: Solution

P1: (a) P1 is 16 bytes. (b) t has an offset of 8 bytes. (c) Required alignment is 4 bytes since the largest type (`int`) requires 4 bytes. (d) This could be reduced to 12 bytes with the following order:`int i, int j, char c, char t`. This will reduce the necessary padding from 6 bytes to 2 bytes.

```
  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  c   X   X   X   i   i   i   i   t   X   X   X   j   j   j   j
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

P2: (a) P2 is 24 bytes. (b) t has an offset of 16 bytes. (c) Required alignment is 8 bytes since the largest types (`long and char *`) both require 8 bytes. This is why the extra two bytes of padding at the end of P2 are needed. (d) The size of P2 cannot be reduced.

```
  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  l   l   l   l   l   l   l   l   p   p   p   p   p   p   p   p
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+


 16  17  18  19  20  21  22  23
+---+---+---+---+---+---+---+---+
  t   t   t   t   s   s   X   X
+---+---+---+---+---+---+---+---+
```

P3: (a) P3 is 20 bytes. (b) t has an offset of 16 bytes. (c) Required alignment is 4 bytes since the largest type (`int`) requires 4 bytes. (d) The size of P3 cannot be reduced.

```
  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  i   i   i   i   i   i   i   i   i   i   i   i   c   c   c   X
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+


 16  17  18  19
+---+---+---+---+
  t   t   t   t
+---+---+---+---+
```

P4: (a) P4 is 32 bytes. (b) t has an offset of 8 bytes. (c) Required alignment is 8 bytes since the largest type (`double *`) requires 8 bytes. This is why 4 bytes of padding are needed at the end of P4. (d) The size of P4 can be reduced to 24 bytes with the following order: `double * t, float f[2], int i, short s, char c`. Only 1 byte of padding is needed with this ordering.

```
  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  c   X   X   X   i   i   i   i   t   t   t   t   t   t   t   t
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+


 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  s   s   X   X   f   f   f   f   f   f   f   f   X   X   X   X
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Problem 4: Memory Allocation

The following C program has many errors with memory allocation using `malloc()` and `free()`, which might prevent the code from compiling, or make it crash or have other undesirable behavior when run. Identify all these errors. For each error, briefly explain the issue and suggest a way to fix it. The code is divided in to four sections: A, B, C, and D. All of the dynamically allocated memory should be `free`d before beginning the next section. There may be more than one error in each section.

```
struct pet {
    char name[64];
    int species;
    int friendliness;
    float weight;
};

int main() {
// A
    int static_array[5] = {1, 2, 3, 4, 5};
    int *dynamic_array = static_array;
    *dynamic_array = 10;
    free(dynamic_array);

// B
    int static_array_2[5] = {1, 2, 3, 4, 5};
    int *dynamic_array_2 = (int *)malloc(5 * sizeof(int));
    for(int i = 0; i <= 5; i++) {
        dynamic_array_2[i] = static_array_2[5 - i];
    }
    free(dynamic_array_2);

// C
    struct pet max = {"Max", 1, 15, 60.5};
    struct pet *max_ptr = (struct pet*)malloc(sizeof(struct pet));
    *max_ptr = max;
    max_ptr->name = "Max";
    free(*max_ptr);

// D
    struct pet** my_pets = (struct pet**)malloc(10 * sizeof(struct pet));
    for(int i = 0; i < 10; i++) {
        my_pets[i] = (struct pet*)malloc(sizeof(struct pet));
    }

    free(*my_pets);
}
```

Problem 4: Solution

A: There is one error in part A. Calling `free` on memory that was not allocated with `malloc` will result in undefined behavior. There is a chance that the program will crash, but that won't always be the case. It is likely that a compiler would not catch this type of issue. This is similar to freeing memory that was already freed. Think about Hands-on Assingment 4, and what would happen if a user of your allocator asked to free a pointer that wasn't on the heap.

A simple solution is to remove the line `free(dynamic_array)`.

B: Part B doesn't have an issue with `free` or `malloc`. The error is indexing both arrays out of bounds. When `i=0`, `static_array_2` is accessed at index 5, and when `i=5`, `dynamic_array_2` is also indexed at 5. This is out of bounds for both arrays and may potentially cause a segfault.

A possible fix is:

```
int *dynamic_array_2 = (int *)malloc(5 * sizeof(int));
for(int i = 0; i < 5; i++) {
    dynamic_array_2[i] = static_array_2[4 - i];
}
```

C: The issue in part C is a small issue here that a compiler would likely catch. The `free` function expects to have a pointer (memory address) as its argument. However, `*max_ptr` is the structure that `max_ptr` is pointing to, not the address of the structure. To properly free this memory, use the call

```
free(max_ptr)
```

D: Part D has 2 errors. The first error is with how the array `my_pets` is being allocated. Since we want an array of pointers, the size passed into `malloc` should be the size of the pointer and not the size of the structure itself. This call is significantly over-allocating memory. Since the structure is 76 bytes long, this call allocates $10 * 76 = 760$ bytes, however, we only need pointers that are only 8 bytes long, we actually only need to allocate 80 bytes.

The other issue is with how the array is being freed. Right now, only the first pointer in the array is being freed, while the rest of the memory stays around until the end of the program. This is a standard case of memory leak. To properly free all memory, first loop through the array to free all the pointer in the array. Then free the pointer to the array itself. In total, this will result in calling `free` 11 times (which is exactly how many times `malloc` was called). A good rule of thumb is to call `free` once for each time `malloc` is called.

A better approach would have been:

```
struct pet** my_pets = (struct pet**)malloc(10 * sizeof(struct pet *));
for(int i = 0; i < 10; i++) {
    my_pets[i] = (struct pet*)malloc(sizeof(struct pet));
}
for(int i = 0; i < 10; i++) {
    free(my_pets[i]);
}
free(my_pets);
```