Lab 3
C Memory Management and Introduction to RISC-V

Qadis Chaudhry

October 27, 2021

Exercise 1

1. Contiguous memory refers to a set of memory that is already allocated for user processes while dynamically allocated memory is memory that is defined by the user for a specific task. The main difference between the two is the fact that dynamically allocated memory is able to change size while contiguous is fixed, often referred to as static memory. Accessing them will take different amounts of time based on the task which is being computed. In a general sense, the contiguous memory will be able to be accessed faster since it is fixed whereas when accessing dynamically allocated memory, the computer has to make sure not only that the correct data is being extracted, but also at the right time since the memory space can change size at any time. This affects the speed at which data can be accessed and therefore, the overall speed of the program.

2.
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* array;
    array = (int*) malloc(20 * sizeof(int));
    free(array);
    return 0;
}
```

   If we attempt to print the array before it is freed, it will be printed with no issue. This is because the array is dynamically allocated in the memory and it can be manipulated as the programmer wishes. We can print the array as well within this time since it exists in memory. After freeing it however, the value printed will be undefined since there is technically nothing in the array for the program to print. This will result in a NULL expression or an error since there is nothing to print from the viewpoint of the program.

Exercise 2

1. Using the simulator we can step through each line of code and see how the registers change with every instruction. The first two lines,

```
addi x5, x0, 3
addi x6, x0, 11
```

   are simply assigning values to the registers `x5` and `x6`, where `x5` equals 3 and `x6` equals 11. The next line performs the AND operation on the values of registers `x5` and `x6`, and stores the result in `x7`.

```
and x7, x6, x5
```

   In binary, the value in `x5` would be 0011, and the value in `x6` is 1011. Performing bitwise AND on these two numbers will yield the result 0011 in binary, or 3 in decimal. This value is stored in `x7`. The last line,

```
slli x7, x7, x5
```

   takes the value stored in `x7` and shifts it left by the number of bits in register `x5`. Since the value in `x5` is 3, the number in `x7` will be shifted left by 3 digits which is the same as multiplying the number by 2^3 or 8. This will yield a value of 24, since 3 * 8 is 24, stored in register `x7`.

2. The largest value we can have for the constants is 32. This is because of the fact that when we are shifting bits, the largest value we can shift by is 32 since that would be equal to multiplying by 2^{32}. Since we are using 32 bit registers, any value higher than that will yield incorrect results as the numbers cannot be represented with the number of bits provided by the register.

3.

```
#include <stdio.h>

int main()
{
    int arr[9];
    arr[4] = arr[4] << arr[8];
    return 0;
}
```

   This is the program in C representing the assembly instructions given in the problem. According to the instructions, the pointer variable has the highest value of 9, meaning that the array must have be of at least 9 elements. This warrants the line `int arr[9]` as this initializes an array with 9 elements. From the actual instructions now we can see

that the first two lines are simply loading the values. Due to the byte addressing nature of arrays, by dividing the offset by 4 we can see which index these values are coming from if the base address of the array is `x27`. Register `x5` stores the value of `arr[4]` and register `x6` stores the value of `arr[8]`. The second to last line signifies the shifting of `x5` by the value stored in `x6`. The shifting operation can be done in C with the "`<<`" operator. The last line signifies that this result is stored back into index 4 of the array. Putting it all together, we get an expression of `arr[4] = arr[4] << arr[8]`.

4.

```
lui x5, 0x87654          // stores in left 20 bits of the address
addi x5, x5, 0x321       // adds the rest of the bits to the address
ld x22, 0(x5)            // loads the contents of x5 into memory with offset 0
                         assuming address is in x22
```

Exercise 3

1. The output of this function was "900." The first two lines act as loading instructions as they load values into the registers `x10` and `x11`. The last line is `ecall` and this gives calls to the OS. Since the register `x11` is responsible for return values, the program will output the value in the register when called. This may be useful in applications since we may need to call the kernel to execute some possess in the evolution of the program, the `ecall` instruction can achieve this purpose.

2.

```
addi x5, x0, 24          // put value of x in x5
addi x6, x0, 56          // put value of y in x6
addi x10, x0, 4          // sets call to print string
ecall                    // assuming "x * y = " is in x11
mul x7, x5, x6           // multiply x and y
addi x10, x0, 1          // set call to print integer
add x11, x0, x7          // put value of x7 in x11
ecall                    // prints value x * y from x11
```

3.

```
addi x26, x0, 32          // a = 32
addi x27, x0, 36          // b = 36
addi x28, x0, 10          // c = 10
bge x26, x27, Else        // if a ≥ b, move to else
slli x29, x28, 1          // c * 2 and store result in x29
addi x10, x0, 1           // set call to print integer
add x11, x0, x29          // put value of c * 2 in x11 to be printed
ecall                     // print value in x11
beq x0, x0, End           // end program
```

Else:

```
mul x26, x27, x27         // b * b and store the value in a
div x27, x26, 2           // a/2 and store value in b
addi x10, x0, 1           // sets call to print integer
add x11, x0, x28          // puts value to be printed, c, in x11
ecall                     // print value in x11
```

End: ...

Exercise 4

1. This will jump back to the `main` label since the instruction `j` represents jump and `main` refers to the label to which it jumps. This will lead to an infinite loop since the `main` label represents the top of the file and adding this to the bottom will jump to that label. This will continue on forever since the loop is being reinstated at the end of each iteration.

2. exercise4:

```
        add x6, x0, x0          // res = 0 stored in x6
        addi x5, x0, 99         // i = 99 stored in x5
        add x29, x6, x10        // x29 = res + a
        j Loop                  // jump to Loop
```

Loop:

```
        bge x29, x5, Exit       // if a + res is greater than i, exit
        add x6, x6, x10         // res = res + a
        add x5, x5, -1          // i = i - 1
        j Loop                  // Loop again
```

Exit:

```
        addi x1, x0, x6         // put x6 in return address
        jalr x0, 0(x1)          // return to caller
```

main:

```
        addi x10, x0, 10        // a = 10 stored in x10
        jal x1, exercise4       // jump and link to procedure
        addi x28, x0, x1        // put return address value in x28, b
        addi x10, x0, 1         // sets call to print integer
        add x11, x0, x28        // puts value to be printed, b, in x11
        ecall                   // print value in x11
```