Assignment 3

Kalpesh Krishna (140070017)
Kumar Ayush (140260016)

1 Question 1

1.1 Part a) and b)

We obtain the following output on running `memory.c`,

```
kalpesh@kalpesh-Inspiron-3542:assign3$ ./a.out
bss 1 - 0x601054
bss 2 - 0x601058
data segment 1 - 0x601048
data segment 2 - 0x60104c
stack 1 - 0x7ffd3dd72458
stack 2 - 0x7ffd3dd7245c
heap 1 - 0xf57010
heap 2 - 0xf57030
heap 1 (pointer) - 0x7ffd3dd72460
heap 2 (pointer) - 0x7ffd3dd72468
```

This is consistent with the output of `objdump -h a.out`,

```
 12 .text         00000262  0000000000400490  0000000000400490  00000490
    2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 23 .data         00000018  0000000000601038  0000000000601038  00001038
    2**3
                  CONTENTS, ALLOC, LOAD, DATA
 24 .bss          00000010  0000000000601050  0000000000601050  00001050
    2**2
                  ALLOC
```

1.2 Part c)

- **Stack** - As expected, the variables have been allocated on contiguous memory locations on the stack (they are of type `int` and each have size 4 bytes)
- **Heap** - The two integer variables are not on contiguous memory locations since we used two separate `malloc()` calls. Notice that the addresses are much smaller when compared to the stack addresses, indicating that the heap generally occupies the lower address spaces.
- **bss** - The addresses we obtain are contiguous and consistent within the `bss` segment range from 0x601050 to 0x601060 (exclusive).

- **data segment** - The addresses we obtain are contiguous and consistent within the `data` segment range from 0x601038 to 0x601050 (exclusive).

1.3 Part d)

In this part we allocate exactly 32 bytes (eight integers) in each of the segments. The first `size` call is before the allocation and the second `size` call is after allocation.

```
kalpesh@kalpesh-Inspiron-3542:assign3$ gcc memory2.c
kalpesh@kalpesh-Inspiron-3542:assign3$ size ./a.out
   text    data     bss     dec     hex filename
   1115     552       8    1675     68b ./a.out
kalpesh@kalpesh-Inspiron-3542:assign3$ gcc memory2.c
kalpesh@kalpesh-Inspiron-3542:assign3$ size ./a.out
   text    data     bss     dec     hex filename
   1115     584      40    1739     6cb ./a.out
kalpesh@kalpesh-Inspiron-3542:assign3$
```

Notice both the `data` and `bss` segments have increased by 32 bytes. Uninitialized global or static variables are allocated in the `bss` segment whereas initialized global or static variables are allocated in the `data` segment.

1.4 Part e)

We obtain the virtual address section of the text segment using the `objdump -h` command.

```
kalpesh@kalpesh-Inspiron-3542:assign3$ objdump -h memory | grep 'text'
 12 .text         00000272  0000000000400490  0000000000400490  00000490
    2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .fini         00000009  0000000000400704  0000000000400704  00000704
    2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE

kalpesh@kalpesh-Inspiron-3542:assign3$ objdump -h memory2 | grep 'text'
 12 .text         00000172  0000000000400400  0000000000400400  00000400
    2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .fini         00000009  0000000000400574  0000000000400574  00000574
    2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE

kalpesh@kalpesh-Inspiron-3542:assign3$
```

For the first program, the size is 0x272 bytes, or 626 bytes. The text segment begins at 0x400490 and ends at 0x400704 (exclusive). 2 extra bytes are needed due to alignment (reference).
Similarly, for the second program, the size is 0x172, or 370 bytes. The text segment begins at 0x400400 and ends at 0x400574 (exclusive). 2 extra bytes are needed due to alignment.

2 Question 2

The /proc filesystem contains a map between virtual memory pages and physical memory pages in the /proc/$(PID)/pagemap file. This map can be used to obtain a mapping from virtual addresses to physical addresses.
We utilize a function found on StackOverflow (reference) which reads the pagemap file and carries out the mapping process. This function is based on the pagemap documentation given here. We describe the function here. **sudo access is necessary to run this code**.

```
FILE *pagemap;
intptr_t paddr = 0;
int offset = (vaddr / sysconf(_SC_PAGESIZE)) * sizeof(uint64_t);
uint64_t e;
```

pagemap is a reference to the actual file, paddr will store the final physical address. offset refers to the offset within the pagemap file. Each entry is 64 bits long. We need to find the current virtual page number and move to that address.

```
lseek(fileno(pagemap), offset, SEEK_SET)
```

Moves the file read pointer to the offset location, the place where the virtual page should be found.

```
fread(&e, sizeof(uint64_t), 1, pagemap)
```

64 bits are read into e.

```
e & (1ULL << 63)
```

This bitwise operation checks whether the bit 63 is non zero, which indicates the presence of a page. (This is mentioned in Documentation/vm/pagemap.txt).

```
paddr = e & ((1ULL << 54) - 1); // pfn mask
paddr = paddr * sysconf(_SC_PAGESIZE);
// add offset within page
paddr = paddr | (vaddr & (sysconf(_SC_PAGESIZE) - 1));
```

This snippet reads bits 0-53, the location of page frame (again mentioned in Documentation/vm/pagemap.txt). To get an actual address, it's necessary to multiply this by the page size and add the offset present in the virtual address.

2.1 Results

Here is a sample output.

```
kalpesh@kalpesh-Inspiron-3542:part2$ gcc q2.c
kalpesh@kalpesh-Inspiron-3542:part2$ ./a.out
Virtual address - 0x601074
Physical address - (nil)
kalpesh@kalpesh-Inspiron-3542:part2$ sudo ./a.out
Virtual address - 0x601074
Physical address - 0x68791074
kalpesh@kalpesh-Inspiron-3542:part2$
```

Here a data segment variable's (initialized with static) pointer is used. Notice that without root access, the function is unable to access the pagemap file.