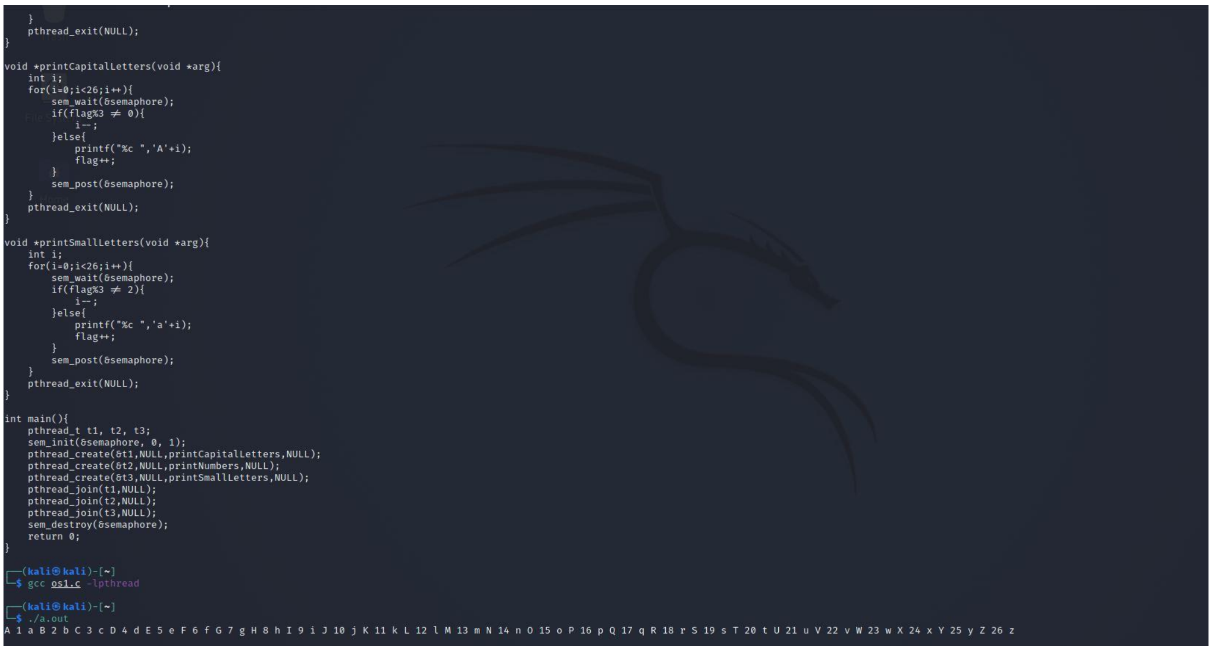
Yash Doshi - AU1941097

Q1)

Multithread: Here we are using the semaphore and threads for generating the output. We are using a binary semaphore with a shared variable flag. The flag variable works like the counter. Now we are creating three threads as t1, t2, and t3 respectively. Now we are using three function as 'printNumbers()', 'printCapitalLetters()' and 'printSmallNumbers()' for thread t1, t2, and t3 thread to execute them respectively. In the printCapitalLetters() we are using iterating over a for loop from 0 to 26. And in each iteration, we are waiting for the semaphore to get access to use the shared variable. After that, if the value of flag variable modulo 3 is zero then we print the loop index 'i + A'. Else we decrement the loop index 'i' just for having an additional iteration. The printNumber() and printSmallLetters work similar to printCapitalLetters(), which are used to print numbers and alphabets in lowercase respectively. In the main function, we are creating the three threads with an inbuilt function 'pthread_create()' and then we use pthread_join for each thread which works as wait() for all the threads to complete their tasks.



```
    }
    pthread_exit(NULL);
}

void *printCapitalLetters(void *arg){
    int i;
    for(i=0;i<26;i++){
        sem_wait(&semaphore);
        if(flag%3 ≠ 0){
            i--;
        }else{
            printf("%c ",'A'+i);
            flag++;
        }
        sem_post(&semaphore);
    }
    pthread_exit(NULL);
}

void *printSmallLetters(void *arg){
    int i;
    for(i=0;i<26;i++){
        sem_wait(&semaphore);
        if(flag%3 ≠ 2){
            i--;
        }else{
            printf("%c ",'a'+i);
            flag++;
        }
        sem_post(&semaphore);
    }
    pthread_exit(NULL);
}

int main(){
    pthread_t t1, t2, t3;
    sem_init(&semaphore, 0, 1);
    pthread_create(&t1,NULL,printCapitalLetters,NULL);
    pthread_create(&t2,NULL,printNumbers,NULL);
    pthread_create(&t3,NULL,printSmallLetters,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    pthread_join(t3,NULL);
    sem_destroy(&semaphore);
    return 0;
}

┌──(kali㉿kali)-[~]
└─$ gcc os1.c -lpthread

┌──(kali㉿kali)-[~]
└─$ ./a.out
A 1 a B 2 b C 3 c D 4 d E 5 e F 6 f G 7 g H 8 h I 9 i J 10 j K 11 k L 12 l M 13 m N 14 n O 15 o P 16 p Q 17 q R 18 r S 19 s T 20 t U 21 u V 22 v W 23 w X 24 x Y 25 y Z 26 z
```

Q2)

Buddy's Algorithm:

Yash Doshi - AU1941097

Buddy’s Algorithm

Memory Allocation: The Algorithm is used for the memory management for the process as they are requested. There is a free list of nodes of all possible powers of 2. When the request for the allocation is there we look for the smallest block which is bigger than the requested memory size and allocate to it if available. Else we look forward to the next bigger block. And once we reach the desired node we split it into two parts one to traverse down the tree and one for adding the next free list.

Example: Let say we have a memory block of size 64kb in our free list.

free list: {}, {}, {}, {}, {}, {(0,63)}

Request 28 byte: We will traverse up to the 64kb block since all others are not available. Then we will allocate that block and split the block into two parts as (0,31) and (32,63) respectively.

free list: {}, {}, {}, {{0,31}}, {(32,63)}, {}

And so on for all the requests.

Memory Deallocation:
Here we are also creating a map when we allocate memory with the buddy's algorithm. In this map, we are storing starting address as key and size as value. When we receive a valid deallocation request (the request for memory deallocation which is allocated). So then we will add the block to the free list and we will also the merge the consecutive free memory with it. Here we will use two variable buddy_number and buddy_address which help us to know the buddy of the any block.

Example: If we have above free list

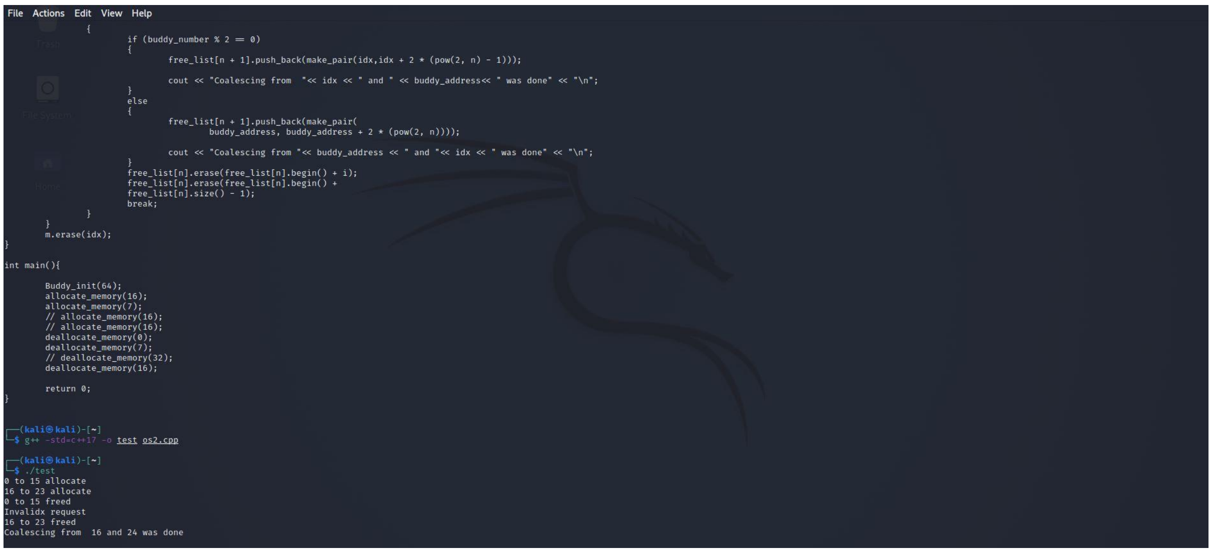
free list: {}, {}, {}, {}, { (0, 15), (32-47) }, {}

Deallocation Request: Index = 16

Yash Doshi - AU1941097

Here there will be deallocation with coalescing of (0-15) and (16-31) blocks and buddy_address of (16-31) is zero which is present in the free list

Resultant free list: {}, {}, {}, {}, { (32-47) }, { (0, 31) }, { }



```
                {
                        if (buddy_number % 2 == 0)
                        {
                                free_list[n + 1].push_back(make_pair(idx,idx + 2 * (pow(2, n) - 1)));

                                cout << "Coalescing from  "<< idx << " and " << buddy_address<< " was done" << "\n";
                        }
                        else
                        {
                                free_list[n + 1].push_back(make_pair(
                                        buddy_address, buddy_address + 2 * (pow(2, n))));

                                cout << "Coalescing from "<< buddy_address << " and "<< idx << " was done" << "\n";
                        }
                        free_list[n].erase(free_list[n].begin() + i);
                        free_list[n].erase(free_list[n].begin() +
                        free_list[n].size() - 1);
                        break;
                }
        }
        m.erase(idx);
}

int main(){

        Buddy_init(64);
        allocate_memory(16);
        allocate_memory(7);
        // allocate_memory(16);
        // allocate_memory(16);
        deallocate_memory(0);
        deallocate_memory(7);
        // deallocate_memory(32);
        deallocate_memory(16);

        return 0;
}
```

```
┌──(kali㉿kali)-[~]
└─$ g++ -std=c++17 -o test os2.cpp

┌──(kali㉿kali)-[~]
└─$ ./test
0 to 15 allocate
16 to 23 allocate
0 to 15 freed
Invalidx request
16 to 23 freed
Coalescing from  16 and 24 was done
```