Shail Patel

AU1940142

1. Print the following Pattern

A 1 a B 2 b C 3 c ... Y 25 y Z 26 z

Using any one of the following concepts

b. Multithreads (Hint: using 3 Threads)

```
#include<stdio.h>
#include<pthread.h>
#include<time.h>

pthread_t tid[3];
pthread_mutex_t mutex;
unsigned int gain_lock;
//prototypes for callback functions
int upperCase = 65;
int lowerCase = 97;
int number = 1;
int flag = 0;
void* printUpperCharacters(void*);
void* printLowerCharacters(void*);
void* printNumbers(void*);


void main(void)
{
    pthread_create(&tid[0],0,&printUpperCharacters,0);
    pthread_create(&tid[1],0,&printNumbers,0);
    pthread_create(&tid[2],0,&printLowerCharacters,0);    // create three
thread seperately for upperCharacters, lowerCharacters and numbers

    pthread_join(tid[0],NULL);
    pthread_join(tid[1],NULL);
    pthread_join(tid[2],NULL); // start three threads

}

void* printUpperCharacters(void *ptr)
{
```

```
     gain_lock = pthread_mutex_lock(&mutex); // gain lock for mutual exclusion
for the thread to run
     while(upperCase<=90)
     {
     if(flag%3==0){

        printf("%c ",upperCase);
        upperCase++;
        flag++;

     }
     else{
            gain_lock=pthread_mutex_unlock(&mutex); // release the lock
(mutual exclusion) for other thread to run
     }

     }
}

void* printLowerCharacters(void* ptr1)
{
    gain_lock = pthread_mutex_lock(&mutex); // gain lock for mutual exclusion
for the thread to run
    while(lowerCase<=122)
    {
      if(flag%3==2){
      printf("%c ",lowerCase);
        lowerCase++;
        flag++;

      }
      else{
            gain_lock=pthread_mutex_unlock(&mutex); // release the lock
(mutual exclusion) for other thread to run
      }


    }
}

void* printNumbers(void* ptr1)
{
    gain_lock = pthread_mutex_lock(&mutex); // gain lock for mutual exclusion
for the thread to run
   while(number <= 26)
    {
    if(flag%3==1){
     printf("%d ",number);
```

```
        number++;
        flag++;
    }
    else{
                gain_lock=pthread_mutex_unlock(&mutex); // release the lock
(mutual exclusion) for other thread to run

        }


    }
}
```

Q2. Describe and implement any one of the following [20]

b) Describe the Buddy's Algorithm for Memory Allocation and Deallocation along with an example and implement it in C or C++.

```
#include<bits/stdc++.h>
using namespace std;

int size;

vector<pair<int, int>> arr[100000];

map<int, int> mp;

void Buddy(int s)
{


    int n = ceil(log(s) / log(2));

    size = n + 1;
    for(int i = 0; i <= n; i++)
        arr[i].clear();

    //initially the whole block of size 128 is available
    arr[n].push_back(make_pair(0, s - 1));
}

void allocate(int s)
{
```

```
    int x = ceil(log(s) / log(2));


    if (arr[x].size() > 0)
    {
        pair<int, int> temp = arr[x][0];

        arr[x].erase(arr[x].begin());

        cout << "Memory from " << temp.first
            << " to " << temp.second
            << " allocated" << "\n";


        mp[temp.first] = temp.second -
                         temp.first + 1;
    }
    else
    {
        int i;


        for(i = x + 1; i < size; i++)
        {

            //find a block size greater then the request made
            if (arr[i].size() != 0)
                break;
        }

        // If no such block is found then signal that all blocks are full

        if (i == size)
        {
            cout << "Sorry, failed to allocate memory\n";
        }

        // If found
        else
        {
            pair<int, int> temp;
            temp = arr[i][0];

            // Remove first block to split it into equal halves
            arr[i].erase(arr[i].begin());
            i--;

            for(;i >= x; i--)
```

```
            {

                // Divide block into two halves
                pair<int, int> pair1, pair2;
                pair1 = make_pair(temp.first,
                                  temp.first +
                                  (temp.second -
                                  temp.first) / 2);
                pair2 = make_pair(temp.first +
                                  (temp.second -
                                  temp.first + 1) / 2,
                                  temp.second);

                arr[i].push_back(pair1);

                // Push them in free list
                arr[i].push_back(pair2);
                temp = arr[i][0];

                arr[i].erase(arr[i].begin());
            }

            cout << "Memory from " << temp.first
                << " to " << temp.second
                << " allocate" << "\n";

            mp[temp.first] = temp.second -
                             temp.first + 1;
        }
    }
}

void deallocate(int id)
{


    if(mp.find(id) == mp.end())
    {
        cout << "Sorry, invalid free request\n";
        return;
    }

    // Size of block to be searched
    int n = ceil(log(mp[id]) / log(2));

    int i, buddyNumber, buddyAddress;

    // Add the block in free list
```

```
    arr[n].push_back(make_pair(id,
                            id + pow(2, n) - 1));
    cout << "Memory block from " << id
        << " to "<< id + pow(2, n) - 1
        << " freed\n";

    // Calculate buddy number
    buddyNumber = id / mp[id];

    if (buddyNumber % 2 != 0)
        buddyAddress = id - pow(2, n);
    else
        buddyAddress = id + pow(2, n);

    // Search in free list to find it's buddy
    for(i = 0; i < arr[n].size(); i++)
    {

        // If buddy found and is also free
        if (arr[n][i].first == buddyAddress)
        {

            // Now merge the buddies to make them one into large free memory
block as it was before splitting
            if (buddyNumber % 2 == 0)
            {
                arr[n + 1].push_back(make_pair(id,
                id + 2 * (pow(2, n) - 1)));

                cout << "Coalescing of blocks starting at "
                    << id << " and " << buddyAddress
                    << " was done" << "\n";
            }
            else
            {
                arr[n + 1].push_back(make_pair(
                    buddyAddress, buddyAddress +
                    2 * (pow(2, n))));

                cout << "Coalescing of blocks starting at "
                    << buddyAddress << " and "
                    << id << " was done" << "\n";
            }
            arr[n].erase(arr[n].begin() + i);
            arr[n].erase(arr[n].begin() +
            arr[n].size() - 1);
            break;
        }
```

```
    }

    mp.erase(id);
}

// Driver code
int main()
{

    Buddy(128);
    allocate(18);
    allocate(13);
    allocate(16);
    allocate(16);
    deallocate(0);
    deallocate(10);
    deallocate(31);
    deallocate(14);

    return 0;
}
```

3. [Bonus] Describe what is Producer Consumer Problem and its solution in detail using Semaphores and Mutex and implement it in C.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>

#define MaxItems 3 // Maximum items a producer can produce or a consumer can
consume
#define BufferSize 3 // Size of the buffer

sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int buffer[BufferSize];
pthread_mutex_t mutex;

void *producer(void *pno)
{
    int item;
    for(int i = 0; i < MaxItems; i++) {
        item = rand();
        sem_wait(&empty);
```

```
        pthread_mutex_lock(&mutex);
        buffer[in] = item;
        printf("Producer %d: Item inserted in buffer %d at position %d\n",
*((int *)pno),buffer[in],in);
        in = (in+1)%BufferSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}
void *consumer(void *cno)
{
    for(int i = 0; i < MaxItems; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumer %d: Consumed the Item %d from position %d\n",*((int
*)cno),item, out);
        out = (out+1)%BufferSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}

int main()
{

    pthread_t pro[5],con[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty,0,BufferSize);
    sem_init(&full,0,0);

    int a[3] = {1,2,3};

    for(int i = 0; i < 3; i++) {
        pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]);
    }
    for(int i = 0; i < 3; i++) {
        pthread_create(&con[i], NULL, (void *)consumer, (void *)&a[i]);
    }

    for(int i = 0; i < 3; i++) {
        pthread_join(pro[i], NULL);
    }
    for(int i = 0; i < 3; i++) {
        pthread_join(con[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
```

```
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;

}
```