Code
End-Sem Examination

Name Rushabh Shah
AU1940064

Q1)

```
#include <iostream>
#include <thread>
#include <mutex>
#include <semaphore.h>
#include <time.h>
#include <unistd.h>
#define THREAD_NUM 3
using namespace std;

sem_t smallalpha;
sem_t bigalpha;
sem_t num;

//prints big alpha
void pattern_bigalpha()
{

    char val;

    for (val = 'A'; val <= 'Z'; ++val)
    {
        sem_wait(&bigalpha);
        std::cout << val << "";
        sem_post(&num);
    }
}
//function prints small aplha
void pattern_aplha()
{

    for (int i = 1; i < 27; i++)
    {
        sem_wait(&num);
        std::cout << " " << i << " ";
        sem_post(&smallalpha);
    }
}
//function prints number value
void pattern_num()
{

    char val;
    for (val = 'a'; val <= 'z'; ++val)
    {
        sem_wait(&smallalpha);
        std::cout << val << " ";
        sem_post(&bigalpha);
    }
}

int main()
{
    //intialize semaphores
    sem_init(&smallalpha, 0, 0);
    sem_init(&bigalpha, 0, 1);
    sem_init(&num, 0, 0);

    std::thread smallalpha, bigalpha, num;

    smallalpha = std::thread(pattern_num);
    bigalpha = std::thread(pattern_bigalpha);
    num = std::thread(pattern_aplha);
//join threads
    bigalpha.join();
    num.join();
    smallalpha.join();
}
```

Q3)

```
//Rushabh shah AU1940064
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>

sem_t sem_em;
sem_t sem_fu;

pthread_mutex_t mutex;

int buffer[10];
int temp = 0;
//producer part
void *producer(void *args)
{
    while (1)
    {

        int x = rand() % 10;

        sem_wait(&sem_em); //checking wheater biffer is empty or ont
        pthread_mutex_lock(&mutex);
        buffer[temp] = x;
        temp++;
        printf("produced %d\n", x);
            pthread_mutex_unlock(&mutex);
        sem_post(&sem_fu); //increase the value of semphore to allow consumer if value
is 0 or less than 0
    }
}
//consumer part
void *consumer(void *args)
{
    while (1)
    {
        int y;

        sem_wait(&sem_fu); //check wheater if there is some items in buffer
        pthread_mutex_lock(&mutex);
        y = buffer[temp - 1];
        temp--;
        pthread_mutex_unlock(&mutex);
        sem_post(&sem_em);

        printf("consumed %d\n", y);
    }
}

int main(int argc, char *argv[])
{
    srand(time(NULL));
    //making four threads
    //equal number of producer nad consumers
    pthread_t threads[4];
    pthread_mutex_init(&mutex, NULL);
    //intialize counting semaphore
    sem_init(&sem_em, 0, 10); //wait for consumer because there are no products in c
    sem_init(&sem_fu, 0, 0);
    int i;
    //creating threads
    for (i = 0; i < 4; i++)
    {
        if (i > 0)
        {
            if (pthread_create(&threads[i], NULL, &producer, NULL) != 0) //checking error
while creating threads
            {
                perror("thread not crerated");
            }
        }
        else
        {
            if (pthread_create(&threads[i], NULL, &consumer, NULL) != 0)
            {
                perror("thread not crerated");
            }
        }
    }

    //joining threads
    for (i = 0; i < 4; i++)
    {
        if (pthread_join(threads
[i], NULL) != 0)
        {
            perror("not able to join threads");
        }
    }
    sem_destroy(&sem_em); //destroy semaphores
    sem_destroy(&sem_fu);
    pthread_mutex_destroy(&mutex); //destroy mutex
    return 0;
}
```

Q3)

```
//Rushabh shah AU1940064
#include<bits/stdc++.h>
using namespace std;

// Size of vector of pairs
int size;

// Global vector of pairs to track all
// the free nodes of various sizes
vector<pair<int, int>> arr[100000];

// Map used as hash map to store the
// starting address as key and size
// of allocated segment key as value
map<int, int> mp;

void BuddyAlgo(int s)
{
        //max 2 power
        int n = ceil(log(s) / log(2));
```

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

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

void alloc(int s)
{

    // Calculate index in free list
    // to search for block if available
    int x = ceil(log(s) / log(2));

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

        // Remove block from free list
        arr[x].erase(arr[x].begin());

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

        // Map starting address with
        // size to make deallocating easy
        mp[temp.first] = temp.second -
                        temp.first + 1;
    }
    else
    {
        int i;

        // If not, search for a larger block
        for(i = x + 1; i < size; i++)
        {

            // Find block size greater
            // than request
            if (arr[i].size() != 0)
                break;
        }

        // If no such block is found
        // i.e., no memory block available
        if (i == size)
        {
            cout << "Sorry, failed to alloc memory\n";
        }

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


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

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

                // Divide block into two halves
                pair<int, int> p1, p2;
                p1 = make_pair(temp.first,
                                        temp.first +
                                        (temp.second -
                                        temp.first) / 2);
                p2 = make_pair(temp.first +
                                        (temp.second -
                                        temp.first + 1) / 2,
                                        temp.second);

                arr[i].push_back(p1);

                // Push them in free list
                arr[i].push_back(p2);
                temp = arr[i][0];

                // Remove first free block to
```

```
                // further split
                arr[i].erase(arr[i].begin());
            }

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

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

void delloc(int id)
{

    // If no such starting address available
    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, buddynum, buddyadd;

    // 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
    buddynum = id / mp[id];

    if (buddynum % 2 != 0)
        buddyadd = id - pow(2, n);
    else
        buddyadd = 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 == buddyadd)
        {

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

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

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

    // remove from map
    mp.erase(id);
}

int main()
{
    BuddyAlgo(128);
    alloc(16);
    alloc(16);
    alloc(16);
    alloc(16);
    delloc(0);
    delloc(9);
    delloc(32);
    delloc(16);

    return 0;
}
```