Harsh Patel
AU1940184

Que-1

(b)

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

int x = 0;

pthread_mutex_t mutex;

void* routine() {
    for (int i = 0; i < 9; i++) {
        pthread_mutex_lock(&mutex);
        x++;

        printf("%c \t", x+64);
        printf("%d \t", x);
        printf("%c \t", x+96);

        pthread_mutex_unlock(&mutex);
        // read mails
        // increment
        // write mails
    }
}

int main(int argc, char* argv[]) {
    pthread_t t1, t2, t3;

    pthread_mutex_init(&mutex, NULL);

    if (pthread_create(&t1, NULL, &routine, NULL) != 0) {
        return 1;
    }

    if (pthread_create(&t2, NULL, &routine, NULL) != 0) {
        return 2;
    }

    if (pthread_create(&t3, NULL, &routine, NULL) != 0) {
        return 3;
    }


    if (pthread_join(t1, NULL) != 0) {
        return 5;
    }

    if (pthread_join(t2, NULL) != 0) {
        return 6;
    }

    if (pthread_join(t3, NULL) != 0) {
        return 7;
    }


    pthread_mutex_destroy(&mutex);


    return 0;
}
```

Que-2

(b)

Buddy Algorithm for Memory Allocation

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

int size;           // Size for the vector of pairs

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

map<int, int> mp;

void initialize(int z)
{
    int x = ceil(log(z) / log(2));    // for the max num of power of 2

    size = x + 1;

    for(int i = 0; i <= x; i++)
        free_list[i].clear();

    free_list[x].push_back(make_pair(0, z - 1));
}

void allocate(int z)
{

    int x = ceil(log(z) / log(2));  //calculating index in free list if block is avaible

    if (free_list[x].size() > 0)        //if blcok is avaible
    {
        pair<int, int> temp = free_list[x][0];

        free_list[x].erase(free_list[x].begin());           //removing block from the free list
        cout << "Memory from " << temp.first
            << " to " << temp.second << " is allocated"
            << "\n";


        mp[temp.first] = temp.second -
                                temp.first + 1;
    }
    else
    {
        int i;
        for(i = x + 1; i < size; i++)
        {

            if(free_list[i].size() != 0)          //finding the block size greater than the
index
                break;
        }

        if (i == size)              //if no block is avaible
        {
            cout << "Failed to allocate memory \n";
        }
        else            //if blcok is avaible
        {
            pair<int, int> temp;
            temp = free_list[i][0];

            free_list[i].erase(free_list[i].begin());       //removing first block for
splitting into halves
            i--;

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

                pair<int, int> pair1, pair2;        //dividing blocks into halves
                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);

                free_list[i].push_back(pair1);

                free_list[i].push_back(pair2);      //push blocks into free list
                temp = free_list[i][0];

                free_list[i].erase(free_list[i].begin());
            }
            cout << "Memory from " << temp.first
                << " to " << temp.second
                << " is allocated" << "\n";

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

int main()
{

    initialize(256);
    allocate(32);
```

```
        allocate(18);
        allocate(42);
        allocate(68);
        allocate(94);

        return 0;
}
```

Buddy Algorithm for Memory Deallocation

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

int size;        // Size for the vector of pairs

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

map<int, int> mp;

void initialize(int z)
{

        int x = ceil(log(z) / log(2));    // for the max num of power of 2

        size = x + 1;

        for(int i = 0; i <= x; i++)
                free_list[i].clear();

        free_list[x].push_back(make_pair(0, z - 1));
}

void allocate(int z)
{

        int x = ceil(log(z) / log(2));  //calculating index in free list if block is avaible

        if (free_list[x].size() > 0)        //if blcok is avaible
        {
                pair<int, int> temp = free_list[x][0];

                free_list[x].erase(free_list[x].begin());          //removing block from the free list
                cout << "Memory from " << temp.first
```

```
            << " to " << temp.second << " is allocated"
            << "\n";


        mp[temp.first] = temp.second -
                                temp.first + 1;
    }
    else
    {
        int i;
        for(i = x + 1; i < size; i++)
        {

            if(free_list[i].size() != 0)        //finding the block size greater than the
index
                break;
        }

        if (i == size)          //if no block is avaible
        {
            cout << "Failed to allocate memory \n";
        }

        else            //if blcok is avaible
        {
            pair<int, int> temp;
            temp = free_list[i][0];

            free_list[i].erase(free_list[i].begin());           //removing first block for
splitting into halves
            i--;

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

                pair<int, int> pair1, pair2;            //dividing blocks into halves
                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);
                free_list[i].push_back(pair1);

                free_list[i].push_back(pair2);            //push blocks into free list
                temp = free_list[i][0];

                free_list[i].erase(free_list[i].begin());
            }
            cout << "Memory from " << temp.first
                << " to " << temp.second
                << " is allocated" << "\n";

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

void deallocate(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 x = ceil(log(mp[id]) / log(2));

    int i, buddyNumber, buddyAddress;

    // Add the block in free list
    free_list[x].push_back(make_pair(id,
                                            id + pow(2, x) - 1));
    cout << "Memory block from " << id
        << " to "<< id + pow(2, x) - 1
        << " freed\n";

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

    if (buddyNumber % 2 != 0)
```

```
            buddyAddress = id - pow(2, x);
        else
            buddyAddress = id + pow(2, x);

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

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

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

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

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

        // Remove the key existence from map
        mp.erase(id);
    }
```

```
// Driver code
int main()
{

        initialize(128);
        allocate(16);
        allocate(32);
        allocate(64);
        allocate(16);
        deallocate(0);
        deallocate(9);
        deallocate(32);
        deallocate(16);

        return 0;
}
```

Que-3

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#define MaximumItems 5    // Maximum items that will be poduced by producer
#define Buffer_Size 5     // Size of the buffer

sem_t Empty;
sem_t full;
int i,in = 0;
int out = 0;
int buffer[Buffer_Size];
pthread_mutex_t mutex;

void *producer(void *pno)
{
   int item;

   for(i = 0; i < MaximumItems; i++) {
      item = rand();                  // Producing an random item
      sem_wait(&Empty);
      pthread_mutex_lock(&mutex);
      buffer[in] = item;
      printf("Producer %d: Insert Item %d at %d\n", *((int *)pno),buffer[in],in);
```

```
        in = (in+1)%Buffer_Size;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}


void *consumer(void *cno)
{

    for(i = 0; i < MaximumItems; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        printf("Consumer %d: Remove Item %d from %d\n",*((int *)cno),item, out);
        out = (out+1)%Buffer_Size;
        pthread_mutex_unlock(&mutex);
        sem_post(&Empty);
    }
}


int main()
{

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

    int k[5] = {1,2,3,4,5}; //Will give numbering the producer and consumer

    for(i = 0; i < 5; i++) {
        pthread_create(&pro[i], NULL, (void *)producer, (void *)&k[i]);
    }

    for(i = 0; i < 5; i++) {
        pthread_create(&con[i], NULL, (void *)consumer, (void *)&k[i]);
    }

    for(i = 0; i < 5; i++) {
        pthread_join(pro[i], NULL);
    }
    for(i = 0; i < 5; i++) {
        pthread_join(con[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&Empty);
    sem_destroy(&full);


    return 0;

}
```