End Sem OS

Nihar Patel

AU1940119

Code File

Q-1.

Print the following Pattern: A 1 a B 2 b C 3 c ... Y 25 y Z 26 z using multi-Threading in **C** language.

```
// C code to print A 1 a B 2 b C 3 c ... Y 25 y Z 26 z infinitely using pthread
#include <stdio.h>
#include <pthread.h>

// Declaration of thread condition variables
pthread_cond_t cond1 =
                        PTHREAD_COND_INITIALIZER;
pthread_cond_t cond2 =
                        PTHREAD_COND_INITIALIZER;
pthread_cond_t cond3 =
                        PTHREAD_COND_INITIALIZER;

// mutex which we are going to
// use avoid race condition.
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// done is a global variable which decides
// which waiting thread should be scheduled
int done = 1;
// Thread function
void *foo(void *n)
{
        char Calpha;
        char digit;
        char alpha;
        while(1) {

                // acquire a lock
                pthread_mutex_lock(&lock);

                if (done != (int)*(int*)n) {

    // value of done and n is not equal, hold wait lock on condition variable
                        if ((int)*(int*)n == 1) {

    pthread_cond_wait(&cond1, &lock);
                        } else if ((int)*(int*)n == 2) {

    pthread_cond_wait(&cond2, &lock);
                        }
                        else {

    pthread_cond_wait(&cond3, &lock);
                        }
```

```
}
// done is equal to n, then print n
if(done == 1){
    printf("%c ", Calpha);
    Calpha++;
}
else if(done == 1){
    printf("%c ", digit);
    digit++;
}
else {
    printf("%c ", alpha);
    alpha++;
}

// Now time to schedule next thread accordingly
// using pthread_cond_signal()
if (done == 3) {
        done = 1;
        pthread_cond_signal(&cond1);
}
else if(done == 1) {
        done = 2;
        pthread_cond_signal(&cond2);
} else if (done == 2) {
```

```
                done = 3;
                pthread_cond_signal(&cond3);
            }

            // Finally release mutex
            pthread_mutex_unlock(&lock);
        }
        return NULL;
}

// Driver code
int main()
{
        char last_char = z;
        pthread_t tid1, tid2, tid3;
        int n1 = 1, n2 = 2, n3 = 3;
        // Create 3 threads
        pthread_create(&tid1, NULL, foo, (void *)&n1);
        pthread_create(&tid2, NULL, foo, (void *)&n2);
        pthread_create(&tid3, NULL, foo, (void *)&n3);

        // infinite loop to avoid exit of a program/process
        while(alpha != last_char);

        return 0;
}
```

Q-2.

Buddy’s Algorithm for Memory Allocation and Deallocation implement in C++.

```
#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 Buddy(int s)
{

        // Maximum number of powers of 2 possible
        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 allocate(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 allocate memory\n";
            }

            // If found
```

```
else

{

          pair<int, int> temp;

          temp = arr[i][0];



          // Remove first block to split

          // it into 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];

                // Remove first free block to
                // further split
                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 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, 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 large free memory block
        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;
        }
    }
    // Remove the key existence from map
    mp.erase(id);
}

// Driver code
int main()
{
    Buddy(128);
    allocate(16);
    allocate(16);
    allocate(16);
    allocate(16);
    deallocate(0);
    deallocate(9);
    deallocate(32);
    deallocate(16);

    return 0;
}
```

Q-3.

Implement Producer Consumer Solution using Semaphores and Mutex in C.

```
#include<stdio.h>

#include<stdlib.h>


int mutex=1,full=0,empty=3,x=0;


int main()

{

   int n;

   void producer();

   void consumer();

   int wait(int);

   int signal(int);

   printf("\n1.Producer\n2.Consumer\n3.Exit");

   while(1)

   {

      printf("\nEnter your choice:");

      scanf("%d",&n);

      switch(n)

      {

         case 1:    if((mutex==1)&&(empty!=0))

                        producer();

                    else

                        printf("Buffer is full!!");
                    break;
            case 2:    if((mutex==1)&&(full!=0))
                        consumer();
                    else
                        printf("Buffer is empty!!");
                    break;
            case 3:
                    exit(0);
                    break;
        }
    }

    return 0;
}

int wait(int s)
{
    return (--s);
}

int signal(int s)
{
    return(++s);
}

void producer()
```

```
{
   mutex=wait(mutex);
   full=signal(full);
   empty=wait(empty);
   x++;
   printf("\nProducer produces the item %d",x);
   mutex=signal(mutex);
}

void consumer()
{
   mutex=wait(mutex);
   full=wait(full);
   empty=signal(empty);
   printf("\nConsumer consumes item %d",x);
   x--;
   mutex=signal(mutex);
}
```

----X----