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

The distribution is done through the use of free lists. Now, for deallocation, we'll keep a separate data structure called a Map (with the segment's starting address as the key and the segment's size as the value) and update it whenever an allocation request comes in.

When a deallocation request arrives, we'll first look at the map to determine if it's a legitimate request. If that's the case, we'll add the block to the free list that keeps track of blocks of various sizes.

Then we'll check the free list to see if its friend is available; if so, we'll combine the blocks and place them on the free list above them (which records blocks twice the size); otherwise, we won't coalesce and will simply return.

The **buddyNumber** of a block is calculated by the formula:
(base_address-starting_address_of_main_memory)/block_size

The **buddyAddress** of a block is the starting index of its buddy block, given by the formula:
block_starting_address+block_size (if buddyNumber is even)
block_starting_address-block_size (if buddyNumber is odd)

Let us see how the algorithm proceeds by tracking a memory block of size 128 KB. Initially, the free list is: {}, {}, {}, {}, {}, {}, {}, { (0, 127) }

Allocation Request:16 bytes
List is: {}, {}, {}, {}, { (16, 31) }, { (32, 63) }, { (64, 127) }, {}

Allocation Request:16 bytes
Straight-up memory segment 16-31 will be allocated as it already exists.
List is: {}, {}, {}, {}, {}, { (32, 63) }, { (64, 127) }, {}

Allocation Request:16 bytes
No such block was found, so we will traverse up to block 32-63 and split it into blocks 32-47 and 48-63; we will add 48-63 to list tracking 16-byte

blocks and return 32-47 to a user.
List is: {}, {}, {}, {}, { (48, 63) }, {}, { (64, 127) }, {}

Allocation Request: 16 bytes
List is: {}, {}, {}, {}, {}, {}, { (64, 127) }, {}

Deallocation Request: StartIndex = 0
Deallocation will be done but no coalescing is possible as it buddyNumber is 0 and buddyAddress is 16 (via the formula), none of which is in the free list.
List is: {}, {}, {}, {}, { (0, 15) }, {}, { (64, 127) }, {}

Deallocation Request: StartIndex = 9
Result: Invalid request, as this segment was never allocated.
List is: {}, {}, {}, {}, { (0, 15) }, {}, { (64, 127) }, {}

Deallocation Request: StartIndex = 32
List is: {}, {}, {}, {}, { (0, 15), (32-47) }, {}, { (64, 127) }, {}

- **Deallocation Request:** StartIndex = 16
List is: {}, {}, {}, {}, { (32-47) }, { (0, 31) }, { (64, 127) }, {}

The time complexity of allocation is O(log(n)). In the worst-case scenario for deallocation, all of the allocated blocks could be 1 unit in size, requiring O(n) time to traverse the list for coalescing. In practise, however, such an allocation is extremely unusual, therefore it is usually much faster than linear time.

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

Description:

Both the producer and the consumer use a fixed-size buffer as a queue. The producer is in charge of generating data and storing it in the buffer. The consumer's job is to go through the data in this buffer one after the other.

Pseudocode Solution by using Semaphore and Mutex:

Initialization:

```
Mutex Mutex; // Used to provide mutual exclusion for critical section
Semaphore Empty = N; // Number of empty slots in buffer
Semaphore full = 0 // Number of slots filled
int in = 0; //at this index producer will put the next data
int out = 0; // BY this index, the consumer will consume the next data
int buffer[N];
```

Producer Code:

```
while(True) {
   // this will produce an item
   wait(Empty); // it will wait/sleep when there are no empty slots available
   wait(Mutex);
   buffer[in] = item
   in = (in+1)%buffer_size;
   signal(Mutex);
   signal(full); // Signal/wake to consumer that buffer has some   data and they can consume now
}
```

Consumer Code:

```
while(True) {
   wait(full); // it will wait/sleep when there are no full slots
   wait(Mutex);
   item = buffer[out];
   out = (out+1)%buffer_size;
   signal(Mutex);
   signal(Empty); // Signal/wake the producer that buffer slots are emptied and they can produce
more
}
```

Now we will implement this in our final solution using some methods !

sem_init -> Initialise the semaphore to some initial value
sem_wait -> Same as wait() operation
sem_post -> Same as Signal() operation
sem_destroy -> Destroy the semaphore to avoid memory leak

pthread_mutex_init -> Initialise the mutex
pthread_mutex_lock() -> Same as wait() operation
pthread_mutex_unlock() -> Same as Signal() operation
pthread_mutex_destroy() -> Destroy the mutex to avoid memory leak