OS Assignment-4(write-up)

Ans 1)

The question asked us to implement the dining philosophers problem such that there are 5 philosophers and 5 fork . Additional to that now there are 2 bowls at the middle of the table . For eating , a philosopher requires the two forks besides it and one of the bowl present in the middle of the table . Now , we have to model the philosophers as thread and each of them perform the two functions thinking() and eating() .
The code for the two functions are as follows :
thinking()

```
void thinking(int id) {
   printf("Philosopher %d is thinking right now.\n", id);
   sleep(1);
}
```

This prints philosopher of which id is currently thinking ..

eating()

```
void eating(int id) {
   printf("Philosopher %d is eating right now.\n", id);
   sleep(2);
}
```

This prints philosopher of which id is currently eating .

Also in the thinking() function we are giving a sleep of 1 and in the eating() function we are giving a sleep of 2 . This ensures that threads can come at random(philosophers) such that fairness is maintained .

Now we can see which fork a philosopher can take by making the two functions left and right such that it gives the fork number left to that philosopher and right of that philosopher .
The two functions are:
left()

```
int left(int p) {
   return p;
}
```

right()

```
int right(int p) {
   return (p + 1) % 5;
}
```

Now , we can make the function void* philosopher(void* args) for running the philosopher thread .

Now , to prevent deadlock , a case can happen in which each philosopher pick their left fork and now they all in thinking state .This would create a deadlock and the code will stuck .

Now to prevent this case , we can make the philosopher with even ID pick the left fork first and then the right fork . Similarly , we can make the philosopher with odd ID pick the right fork first and then the left fork . So, such case will never occur and deadlock will be prevented now .
We make mutex forks[5] for this and lock it for the respective cases . So the philosopher waits for the fork till it is unblocked by the other philosopher.
Now with this condition it will never happen that all the philosophers will be in the thinking state .
The code for this condition goes like :

```
pthread_mutex_t forks[5];
```

```
void* philosopher(void* args) {
   int id = *(int*)args;
   int left_fork = left(id);
   int right_fork = right(id);
   while (1) {
       thinking(id);
       if(id%2 == 0){
           printf("Philosopher %d is waiting for fork %d\n",id,left_fork)
;
           pthread_mutex_lock(&forks[left_fork]);
           printf("Philosopher %d is taking fork %d\n",id,left_fork) ;
           printf("Philosopher %d is waiting for fork %d\n",id,right_fork)
;
           pthread_mutex_lock(&forks[right_fork]);
           printf("Philosopher %d is taking fork %d\n",id,right_fork) ;
       }
       else{
           printf("Philosopher %d is waiting for fork %d\n",id,right_fork)
;
           pthread_mutex_lock(&forks[right_fork]);
           printf("Philosopher %d is taking fork %d\n",id,right_fork) ;
           printf("Philosopher %d is waiting for fork %d\n",id,left_fork)
;
           pthread_mutex_lock(&forks[left_fork]);
           printf("Philosopher %d is taking fork %d\n",id,left_fork) ;
       }
```

Now , for the bowls , we make one pthread_cond_t bowl_condition such that it waits for the signal till we get a bowl which is free .
Now we also create an array of integers having 0th index and 1st index as 1 which shows the bowls are first in the available state .
We also create a mutex such that only the part running at that time takes the critical section .

```
pthread_mutex_t mutex;
```

```
pthread_cond_t bowl_condition[2];
int bowl_available[2] = {1, 1};
```

Now , we start with bowl_id = -1 and we change this id if we get a bowl which is currently free .
So we first lock the part with the mutex we created , now we get in the loop which iterates till the bowl_id remains -1 , so it would iterate the array and if the value at that index is 1 , the philosopher acquires the bowl and locks it , otherwise we make the philosopher wait for the bowl 0 with the conditional variable using

```
pthread_cond_wait(&bowl_condition[0], &mutex);
```

Now , after the philosopher gets the signal that bowl[0] is available we make the philosopher acquire it with

```
pthread_mutex_lock(&bowl[0]) ;
```

After this we unlock the critical section by unlocking the mutex .

Now we unlock the bowl by unlocking the mutex of the bowl with id equal to bowl_id and make the value of the bowl’s index as 1 showing that the bowl is now available . Also if the bowl acquired was bowl 0 we make it signal the condition .

```
        pthread_mutex_unlock(&bowl[bowl_id]);
        printf("philosopher %d is dropping bowl %d\n",id,bowl_id)   ;
        pthread_mutex_lock(&mutex);
        bowl_available[bowl_id] = 1;
        pthread_mutex_unlock(&mutex);
        if(bowl_id == 0 ){
            pthread_cond_signal(&bowl_condition[0]);
        }
```

Now we make the forks unlock in the opposite of how we locked it that is right first and then left for the philosopher with even id and vice versa for the philosopher with odd ID . By these conditions there will be now deadlocks in the code .
In the int main() we make the threads of the philosophers and make it wait for all the threads . We also initialize all the mutexes and the conditional variables and destroy them .

```
int main() {
   pthread_t philosophers[5]; // 5 threads for philosophers ..
   int id[5];

   for (int i = 0; i < 5; i++) {
       if (pthread_mutex_init(&forks[i], NULL) != 0) {
           perror("Error initializing fork mutex");
           exit(EXIT_FAILURE);
       }
   }

   for (int i = 0; i < 2; i++) {
       if (pthread_cond_init(&bowl_condition[i], NULL) != 0) {
           perror("Error initializing bowl condition variable");
           exit(EXIT_FAILURE);
       }
       if (pthread_mutex_init(&bowl[i], NULL) != 0) {
           perror("Error initializing bowl mutex");
           exit(EXIT_FAILURE);
       }
```

```
    }

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("Error initializing main mutex");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 5; i++) {
        id[i] = i;
        if (pthread_create(&philosophers[i], NULL, philosopher, &id[i]) !=
0) {
            perror("Error creating philosopher thread");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < 5; i++) {
        if (pthread_join(philosophers[i], NULL) != 0) {
            perror("Error joining philosopher thread");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < 5; i++) {
        if (pthread_mutex_destroy(&forks[i]) != 0) {
            perror("Error destroying fork mutex");
        }
    }

    for (int i = 0; i < 2; i++) {
        if (pthread_mutex_destroy(&bowl[i]) != 0) {
            perror("Error destroying bowl mutex");
        }
        if (pthread_cond_destroy(&bowl_condition[i]) != 0) {
            perror("Error destroying bowl condition variable");
        }
    }

    pthread_mutex_destroy(&mutex);
```

```
   return 0;
}
```

Now the whole code goes like :

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

pthread_mutex_t forks[5];
pthread_mutex_t bowl[2];
pthread_mutex_t mutex;
pthread_cond_t bowl_condition[2];
int bowl_available[2] = {1, 1};

int left(int p) {
   return p;
}

int right(int p) {
   return (p + 1) % 5;
}

void eating(int id) {
   printf("Philosopher %d is eating right now.\n", id);
   sleep(2);
}

void thinking(int id) {
   printf("Philosopher %d is thinking right now.\n", id);
   sleep(1);
}

void* philosopher(void* args) {
   int id = *(int*)args;
   int left_fork = left(id);
   int right_fork = right(id);
   while (1) {
       thinking(id);
```

```
    if(id%2 == 0){
        printf("Philosopher %d is waiting for fork %d\n",id,left_fork)
;
        pthread_mutex_lock(&forks[left_fork]);
        printf("Philosopher %d is taking fork %d\n",id,left_fork) ;
        printf("Philosopher %d is waiting for fork %d\n",id,right_fork)
;
        pthread_mutex_lock(&forks[right_fork]);
        printf("Philosopher %d is taking fork %d\n",id,right_fork) ;
    }
    else{
        printf("Philosopher %d is waiting for fork %d\n",id,right_fork)
;
        pthread_mutex_lock(&forks[right_fork]);
        printf("Philosopher %d is taking fork %d\n",id,right_fork) ;
        printf("Philosopher %d is waiting for fork %d\n",id,left_fork)
;
        pthread_mutex_lock(&forks[left_fork]);
        printf("Philosopher %d is taking fork %d\n",id,left_fork) ;
    }

    int bowl_id = -1;
    pthread_mutex_lock(&mutex);
    while (bowl_id == -1) {
        for (int i = 0; i < 2; i++) {
            if (bowl_available[i]) {
                bowl_id = i;
                bowl_available[i] = 0;
                pthread_mutex_lock(&bowl[i]);
                printf("philosopher %d is taking bowl %d\n",id,bowl_id)
;
                break;
            }
        }
  //       printf("%d\n",bowl_id) ;
        if (bowl_id == -1) {
            // pthread_mutex_unlock(&forks[right_fork]);
            // pthread_mutex_unlock(&forks[left_fork]);
            pthread_cond_wait(&bowl_condition[0], &mutex);
            bowl_id = 0 ;
```

```
               pthread_mutex_lock(&bowl[0]) ;
           }
       }
       pthread_mutex_unlock(&mutex);

       eating(id);

       pthread_mutex_unlock(&bowl[bowl_id]);
       printf("philosopher %d is dropping bowl %d\n",id,bowl_id)  ;
       pthread_mutex_lock(&mutex);
       bowl_available[bowl_id] = 1;
       pthread_mutex_unlock(&mutex);
       if(bowl_id == 0 ){
           pthread_cond_signal(&bowl_condition[0]);
       }
       if(id%2 ==0){
           pthread_mutex_unlock(&forks[right_fork]);
           printf("Philosopher %d is dropping fork %d\n",id,right_fork) ;
           pthread_mutex_unlock(&forks[left_fork]);
           printf("Philosopher %d is dropping fork %d\n",id,left_fork) ;
       }
       else{
           pthread_mutex_unlock(&forks[left_fork]);
           printf("Philosopher %d is dropping fork %d\n",id,left_fork) ;
           pthread_mutex_unlock(&forks[right_fork]);
           printf("Philosopher %d is dropping fork %d\n",id,right_fork) ;
       }
   }
}

int main() {
   pthread_t philosophers[5]; // 5 threads for philosophers ..
   int id[5];

   for (int i = 0; i < 5; i++) {
       if (pthread_mutex_init(&forks[i], NULL) != 0) {
           perror("Error initializing fork mutex");
           exit(EXIT_FAILURE);
       }
   }
```

```
    for (int i = 0; i < 2; i++) {
        if (pthread_cond_init(&bowl_condition[i], NULL) != 0) {
            perror("Error initializing bowl condition variable");
            exit(EXIT_FAILURE);
        }
        if (pthread_mutex_init(&bowl[i], NULL) != 0) {
            perror("Error initializing bowl mutex");
            exit(EXIT_FAILURE);
        }
    }

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("Error initializing main mutex");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 5; i++) {
        id[i] = i;
        if (pthread_create(&philosophers[i], NULL, philosopher, &id[i]) !=
0) {
            perror("Error creating philosopher thread");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < 5; i++) {
        if (pthread_join(philosophers[i], NULL) != 0) {
            perror("Error joining philosopher thread");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < 5; i++) {
        if (pthread_mutex_destroy(&forks[i]) != 0) {
            perror("Error destroying fork mutex");
        }
    }

    for (int i = 0; i < 2; i++) {
        if (pthread_mutex_destroy(&bowl[i]) != 0) {
            perror("Error destroying bowl mutex");
        }
        if (pthread_cond_destroy(&bowl_condition[i]) != 0) {
            perror("Error destroying bowl condition variable");
        }
    }

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Q1) Why deadlocks can occur in the problem setup ?
Ans) Deadlock can happen if all the philosophers take the left fork/ right fork to them and then all of them are in thinking state .
Q2) How your proposed solution avoids deadlock ?
Ans) We made the philosophers with odd Ids take the right fork first and then the left forks . Similarly vice versa for the philosophers with even ids .
Q3) Fairness of the solution i.e. for your implementation, which and how many of the 5 philosopher threads are able to eat, and a rough estimate of how often a philosopher is able to eat (if at all).
Ans) the solution is fair as it lets at max tow philosophers eat at max as given in the question’s condition . Like 1 and 4 can eat together , 0 and 2 can eat together etc . Rough estimate is that after every 2-3 turns a philosopher is able to eat again .
Output :

```
Philosopher 1 is thinking right now.
Philosopher 3 is thinking right now.
Philosopher 0 is thinking right now.
Philosopher 4 is thinking right now.
Philosopher 2 is thinking right now.
Philosopher 3 is waiting for fork 4
Philosopher 3 is taking fork 4
Philosopher 3 is waiting for fork 3
Philosopher 3 is taking fork 3
Philosopher 2 is waiting for fork 2
Philosopher 2 is taking fork 2
Philosopher 2 is waiting for fork 3
Philosopher 1 is waiting for fork 2
Philosopher 4 is waiting for fork 4
Philosopher 0 is waiting for fork 0
Philosopher 0 is taking fork 0
Philosopher 0 is waiting for fork 1
Philosopher 0 is taking fork 1
philosopher 3 is taking bowl 0
Philosopher 3 is eating right now.
philosopher 0 is taking bowl 1
Philosopher 0 is eating right now.
philosopher 0 is dropping bowl 1
Philosopher 0 is dropping fork 1
Philosopher 0 is dropping fork 0
Philosopher 0 is thinking right now.
philosopher 3 is dropping bowl 0
Philosopher 3 is dropping fork 3
Philosopher 3 is dropping fork 4
Philosopher 3 is thinking right now.
Philosopher 2 is taking fork 3
philosopher 2 is taking bowl 0
Philosopher 2 is eating right now.
Philosopher 4 is taking fork 4
Philosopher 4 is waiting for fork 0
Philosopher 4 is taking fork 0
philosopher 4 is taking bowl 1
Philosopher 4 is eating right now.
Philosopher 0 is waiting for fork 0
Philosopher 3 is waiting for fork 4
philosopher 4 is dropping bowl 1
Philosopher 4 is dropping fork 0
Philosopher 4 is dropping fork 4
Philosopher 4 is thinking right now.
philosopher 2 is dropping bowl 0
Philosopher 2 is dropping fork 3
Philosopher 2 is dropping fork 2
Philosopher 2 is thinking right now.
Philosopher 3 is taking fork 4
Philosopher 3 is waiting for fork 3
Philosopher 3 is taking fork 3
```

```
philosopher 3 is taking bowl 0
Philosopher 3 is eating right now.
Philosopher 1 is taking fork 2
Philosopher 1 is waiting for fork 1
Philosopher 1 is taking fork 1
philosopher 1 is taking bowl 1
Philosopher 1 is eating right now.
Philosopher 0 is taking fork 0
Philosopher 0 is waiting for fork 1
Philosopher 4 is waiting for fork 4
Philosopher 2 is waiting for fork 2
philosopher 3 is dropping bowl 0
Philosopher 3 is dropping fork 3
Philosopher 3 is dropping fork 4
Philosopher 3 is thinking right now.
philosopher 1 is dropping bowl 1
Philosopher 1 is dropping fork 1
Philosopher 4 is taking fork 4
Philosopher 4 is waiting for fork 0
Philosopher 1 is dropping fork 2
Philosopher 1 is thinking right now.
Philosopher 2 is taking fork 2
Philosopher 2 is waiting for fork 3
Philosopher 2 is taking fork 3
philosopher 2 is taking bowl 0
Philosopher 2 is eating right now.
Philosopher 0 is taking fork 1
philosopher 0 is taking bowl 1
Philosopher 0 is eating right now.
Philosopher 3 is waiting for fork 4
Philosopher 1 is waiting for fork 2
philosopher 0 is dropping bowl 1
Philosopher 0 is dropping fork 1
Philosopher 0 is dropping fork 0
Philosopher 0 is thinking right now.
philosopher 2 is dropping bowl 0
Philosopher 4 is taking fork 0
philosopher 4 is taking bowl 0
Philosopher 2 is dropping fork 3
Philosopher 2 is dropping fork 2
Philosopher 4 is eating right now.
Philosopher 2 is thinking right now.
Philosopher 1 is taking fork 2
Philosopher 1 is waiting for fork 1
Philosopher 1 is taking fork 1
philosopher 1 is taking bowl 1
Philosopher 1 is eating right now.
Philosopher 0 is waiting for fork 0
Philosopher 2 is waiting for fork 2
philosopher 1 is dropping bowl 1
Philosopher 1 is dropping fork 1
Philosopher 1 is dropping fork 2
Philosopher 1 is thinking right now.
philosopher 4 is dropping bowl 0
Philosopher 4 is dropping fork 0
Philosopher 4 is dropping fork 4
Philosopher 4 is thinking right now.
Philosopher 2 is taking fork 2
Philosopher 2 is waiting for fork 3
Philosopher 2 is taking fork 3
philosopher 2 is taking bowl 0
Philosopher 2 is eating right now.
Philosopher 0 is taking fork 0
Philosopher 0 is waiting for fork 1
Philosopher 0 is taking fork 1
philosopher 0 is taking bowl 1
Philosopher 0 is eating right now.
Philosopher 3 is taking fork 4
Philosopher 3 is waiting for fork 3
Philosopher 1 is waiting for fork 2
Philosopher 4 is waiting for fork 4
philosopher 2 is dropping bowl 0
Philosopher 2 is dropping fork 3
Philosopher 2 is dropping fork 2
Philosopher 2 is thinking right now.
philosopher 0 is dropping bowl 1
Philosopher 0 is dropping fork 1
Philosopher 0 is dropping fork 0
Philosopher 0 is thinking right now.
Philosopher 3 is taking fork 3
```

```
philosopher 3 is taking bowl 0
Philosopher 3 is eating right now.
Philosopher 1 is taking fork 2
Philosopher 1 is waiting for fork 1
Philosopher 1 is taking fork 1
philosopher 1 is taking bowl 1
Philosopher 1 is eating right now.
```

We printed the statemented such that it known which philosopher is waiting for which fork and acquiring which . .

Ans 2)
In this question we needer to implement the question by taking the input from the user with number of passengers and the capacity of the car .
Now if the car has number of people equal to its capacity , we make the car run
This code runs in an infinite number of iterations such that the passengers that board are offboarded after the car do the unloading .

Now implementing the functions of the code :
1) void* car(void* args)
This runs for an infinite loop in which first the car do the loading in which it checks till the number of capacity in the car is equal to the capacity of the car such the car can reach the ready state .
After that condition is reached the function prints that the car is running ..
Then we make the code sleep for 1 sec such that after the delay we make the people unload() from the car . We do this in the sem_post and sem_wait of the sem_t loading_check and sem_t unloading_check respectively .Similarly here we make the code check till the current_capacity is 0 .

Then , we increase the car_semaphores value in the for loop as we decrease it in the passengers function .
Now we sem_wait the semaphores lock unloading check and the capacity_check semaphore .
The code for the same goes like :

```
void* car(void* args){
   while(1){
       load() ;
       sem_post(&loading_check) ;
       while(current_capacity != capacity) {
          // printf("passengers are boarding ..") ;
       }
       sem_wait(&loading_check) ;
       sem_wait(&capacity_check) ;
       printf("Car is running ..\n") ;
       sleep(1) ;
       unload() ;
       sem_post(&unloading_check) ;

       while(current_capacity != 0 ){

           //printf("passengers are unboarding ..") ;

       }

       sem_post(&lock) ;

       for(int i = 0 ;i< capacity ;i++) {

           sem_post(&car_semaphore) ;



       }

       sem_wait(&lock) ;

       sem_wait(&unloading_check) ;

       sem_post(&capacity_check) ;
```

```
    }


    pthread_exit(NULL) ;
}
```

Finally we exit the thread using pthread_exit(NULL) .

2) void* passenger(void* args)
This also goes till an infinite number of iterations .
In this we decrease the semaphore value of the car as described above and store the value of threads made till now .
Now we board() the passengers using the board function , and make it between the sem_wait(&loading_check) and sem_post(&loading_check)
Similarly , when the passenger has to unboard we call the unboard() function in between `sem_wait(&unloading_check) ;` and `sem_post(&unloading_check) ;`

The whole code for this then goes like :

```
 void* passengers(void* args){
    while(1){

        current_threads += 1 ;
        sem_wait(&car_semaphore) ;

        sem_wait(&loading_check) ;

        board(*(int*)args) ;


        sem_post(&loading_check) ;

        sem_wait(&unloading_check) ;
```

```
      offboard(*(int*) args) ;

      sem_post(&unloading_check) ;

   }

   pthread_exit(NULL) ;

}
```

3) void load()   // loading car with passengers
This just checks the condition if current threads are less than total passengers then the code do nothing , else it just prints the car is loading . .and sleeps

```
void load(){
   while(current_threads < total_passengers){

   }
   printf("car is loading ..\n") ;
   sleep(1) ;
}
```

4) coid unload()
This just prints that car is unloading

```
void unload(){

   printf("car is unloading ..\n" ) ;
   sleep(1) ;
}
```

5) board()
This increases the current_capacity and then prints which passenger is boarding with its id .

```

void board(int id){
   sem_wait(&lock) ;

```

```
    printf("passenger %d is curently boarding the car..\n",id) ;
    sleep(1) ;
    current_capacity += 1 ;

    sem_post(&lock) ;

}
```

6) offboard()
This just decreases the current capacity and prints which passenger is boarding with its id .

```

void offboard(int id){
    sem_wait(&lock) ;

    printf("passenger %d is curently unboarding the car..\n",id) ;
    sleep(1) ;
    current_capacity -= 1 ;

    sem_post(&lock) ;



}
```

Initializing the code with taking the input and initializing the semaphores and threads we get the whole code as :

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

int total_passengers ;
int capacity ;



sem_t passenger_sem ;

sem_t boarding ;
sem_t offboarding ;

sem_t waiting  ;

sem_t car_semaphore ;

sem_t lock  ;
sem_t loading_check  ;

sem_t unloading_check ;

sem_t capacity_check ;


int current_capacity = 0;

int current_threads = 0 ;


void load(){
   while(current_threads < total_passengers){

   }
   printf("car is loading ..\n") ;
   sleep(1) ;
}

void unload(){

   printf("car is unloading ..\n" ) ;
   sleep(1) ;
}
void* car(void* args){
   while(1){
       load() ;
       sem_post(&loading_check) ;
       while(current_capacity != capacity) {
          // printf("passengers are boarding ..") ;
       }
```

```c
        sem_wait(&loading_check) ;
        sem_wait(&capacity_check) ;
        printf("Car is running ..\n") ;
        sleep(1) ;
        unload() ;
        sem_post(&unloading_check) ;

        while(current_capacity != 0 ){

            //printf("passengers are unboarding ..") ;

        }

        sem_post(&lock) ;

        for(int i = 0 ;i< capacity ;i++) {

            sem_post(&car_semaphore) ;



        }

        sem_wait(&lock) ;

        sem_wait(&unloading_check) ;

        sem_post(&capacity_check) ;




    }


    pthread_exit(NULL) ;

}

void board(int id){
   sem_wait(&lock) ;

   printf("passenger %d is curently boarding the car..\n",id) ;
   sleep(1) ;
   current_capacity += 1 ;

   sem_post(&lock) ;

}

void offboard(int id){
   sem_wait(&lock) ;

   printf("passenger %d is curently unboarding the car..\n",id) ;
   sleep(1) ;
   current_capacity -= 1 ;

   sem_post(&lock) ;


}

void* passengers(void* args){
   while(1){

       current_threads += 1 ;
       sem_wait(&car_semaphore) ;

       sem_wait(&loading_check) ;

       board(*(int*)args) ;


       sem_post(&loading_check) ;

       sem_wait(&unloading_check) ;

       offboard(*(int*) args) ;

       sem_post(&unloading_check) ;
```

```c
    }

    pthread_exit(NULL) ;

}



int main (){

    printf("Give input for total number of passengers: ") ;
    scanf("%d",&total_passengers) ;
    printf("Give input for capacity of car: ") ;
    scanf("%d",&capacity) ;

    if(total_passengers<0 || capacity <0 || total_passengers <= capacity){
        printf("this is a wrong input\n") ;
    }
    else{

        //making semaphore for car ..
        sem_init(&car_semaphore , 0 , capacity) ;
        //making semahore for implementing lock ..
        sem_init(&lock , 0 , 1) ;
        //making semaphore to check loading process ..
        sem_init( &loading_check,0 , 0) ;
        //making semaphore to check unloading process ..
        sem_init(&unloading_check , 0 , 0) ;
        //making semaphore to check capacity ..
        sem_init( &capacity_check , 0  , 1);

        sem_init(&passenger_sem,0,0) ;

        sem_init(&waiting,0,1) ;

        sem_init(&boarding,0,0 ) ;

        sem_init(&offboarding,0, 1) ;
        pthread_t  my_passenger[total_passengers] ;

        int id[total_passengers] ;

        for(int i = 0;i<total_passengers ;i++){

            id[i] = i ;
            pthread_create(&my_passenger[i] , NULL , passengers , &id[i]) ;

        }

        pthread_t my_car ;

        pthread_create(&my_car,NULL, car ,NULL) ;

        for(int i = 0 ;i<total_passengers ;i++){

            pthread_join(my_passenger[i],NULL) ;

        }

        pthread_join(my_car , NULL) ;


        sem_destroy(&car_semaphore) ;
        //destroying semaphore for car ..
        sem_destroy(&lock) ;
        //destroying semaphore for implementing lock ..
        sem_destroy(&loading_check) ;
        //destroying semaphore for implementing loding check ..
        sem_destroy(&unloading_check) ;
        //destroying semaphore for implementing unloading check ..
        sem_destroy(&capacity_check) ;
        //destroying semaphore for implementing capacity ..

    }

    return 0 ;
}
```

The code uses semaphores to avoid concurrency bugs such that only the code of that part acquires the critical section

Ans 3)

Explaining the functions we get :
We have made a linked which works on FIFO order as we are taking ids of those cars which are entered from any side to make a check of ordering that no threads overtakes another thread .

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

#define MAX_CARS 100000
#define MAX_ON_BRIDGE 5
// sem_t bin;
struct Node {
   int data;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
       printf("Memory allocation failed!\n");
       exit(1);
   }
   newNode->data = value;
   newNode->next = NULL;
   return newNode;
```

```c
}



void append(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
void pop(struct Node** head) {
        if (*head == NULL || (*head)->next == NULL) {
        printf("Linked list doesn't have enough elements to delete the second node.\n");
        return;
    }
    struct Node* temp = (*head)->next;
    (*head)->next = temp->next;
    free(temp);
}
void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

passing() function :
This function takes two parameters such a direction and car number , now if the direction is one then it going to the left direction , here it uses mutex semaphore binary lock which increaments the waiting left numbers , it then uses a semaphore of bridge function which blocks the from the other side , here we have checked the other

condition that any thread scheduled form other side or max limit of 5 has been reached then we stop the incoming threads , the gone threads will start their crossing and print hte direction they are crossing .
The code for the same goes like :

```
int tmpl=5;
void passing(int dir, int car_num) {
   if (dir == 1) {
       sem_wait(&mutex);
       waiting_left++;
       sem_post(&mutex);
       sem_wait(&bridge);
       int flag = 0 ;
       while(tmpr<5 || tmpl<=0){
           if(flag++ == 0){
               printf("Car with id %d is waiting on left side as no space
left on bridge .. \n" , car_num) ;
           // sleep(
           //     1);
           }
       }
       sem_wait(&leftlock);
       // pthread_mutex_lock(&leftlock);
       printf("Car from left side with ID %d is crossing the bridge.\n",
car_num);
       sem_wait(&mutext2);
       struct Node *node = headl;
       append(&headl, car_num);
       // buff[car_num-1] =1;
       sem_post(&mutext2);
       // pr();
       tmpl--;
       // sem_post(&mutext2);
       sem_post(&leftlock);
       // pthread_mutex_unlock(&leftlock);
       sleep(1);
       // sem_wait(&mutext2);
       sem_wait(&leftlock);
       // pthread_mutex_lock(&leftlock);
       if (headl->next->data==car_num){
           sem_wait(&mutext2);
```

```
            pop(&headl);
            // displayList(headl);
            sem_post(&mutext2);
        }
        else{
            // pthread_mutex_lock(&mutext2);
            sem_wait(&mutext2);
            struct Node* n = headl;
            // append(&headl, car_num);
            car_num = headl->next->data;
            pop(&headl);
            // displayList(headl);
            sem_post(&mutext2);
        }
        // pthread_mutex_unlock(&leftlock);
        sem_post(&leftlock);
        // sem_post(&mutext2);
        // pr();
        sem_wait(&mutext2);
        sem_wait(&lock2);
        // pthread_mutex_lock(&lock2);
        printf("Car from left side with ID %d crossed the bridge.\n",
car_num);
        sem_post(&lock2);
        // pthread_mutex_unlock(&lock2);
        // sem_post(&lock2);
        sem_post(&mutext2);
        tmpl++;
        sem_post(&leftlock);
        // pthread_mutex_unlock(&leftlock);
        sem_wait(&mutex);
        waiting_left--;
        sem_post(&mutex);

        sem_post(&bridge);
    } else {
        sem_wait(&mutex);
        waiting_right++;
        sem_post(&mutex);
```

```c
        sem_wait(&bridge);
        int flag = 0 ;
        while(tmpl<5 || tmpr<=0){
            if(flag++ == 0){
                printf("Car with id %d is waiting on right side as no space
left on bridge .. \n" , car_num) ;
            // sleep(1);
            }
        }
        sem_wait(&rightlock);
        // pthread_mutex_lock(&rightlock);
        printf("Car from right side with ID %d is crossing the bridge.\n",
car_num);
        sem_wait(&mutext2);
        // buff2[car_num-1] =1;
        struct Node *node = headr;
        append(&headr, car_num);
        sem_post(&mutext2);
        tmpr--;
        sem_post(&rightlock);
        // pthread_mutex_unlock(&rightlock);
        sleep(1);
        // pthread_mutex_lock(&rightlock);
        sem_wait(&rightlock);
        // pthread_mutex_lock(&rightlock);
        if (headr->next->data==car_num){
            sem_wait(&mutext2);
            pop(&headr);
            sem_post(&mutext2)  ;
        }
        else{
            sem_wait(&mutext2);
            struct Node* n = headr;
            // append(&headr, car_num);
            car_num = headr->next->data;
            pop(&headr);
            sem_post(&mutext2);
        }
        sem_post(&rightlock);
        // pthread_mutex_unlock(&rightlock);
        // if (flag==0)
        sem_wait(&mutext2);
        printf("Car from right side with ID %d crossed the bridge.\n",
car_num);
        sem_post(&mutext2);
        tmpr++;
        // pthread_mutex_unlock(&rightlock);

        sem_wait(&mutex);
        waiting_right--;
        sem_post(&mutex);

        sem_post(&bridge);
    }
}
```

The left adnd right functions are used for ids and calling the passing functions with appropriate direction , and then pthread exit has been called . void *left(void *args) {

```
    // sleep(10);
    int car_num = *((int *)args);
    passing(1, car_num);
    pthread_exit(NULL);
}

void *right(void *args) {
    int car_num = *((int *)args);
    passing(2, car_num);
    pthread_exit(NULL);
}
```

The whole code goes like this :

```

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

```

```c
#define MAX_CARS 100000
#define MAX_ON_BRIDGE 5
// sem_t bin;
struct Node {
   int data;
   struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
       printf("Memory allocation failed!\n");
       exit(1);
   }
   newNode->data = value;
   newNode->next = NULL;
   return newNode;
}


void append(struct Node** head, int value) {
   struct Node* newNode = createNode(value);
   if (*head == NULL) {
       *head = newNode;
       return;
   }
   struct Node* temp = *head;
   while (temp->next != NULL) {
       temp = temp->next;
   }
   temp->next = newNode;
}
void pop(struct Node** head) {
       if (*head == NULL || (*head)->next == NULL) {
       printf("Linked list doesn't have enough elements to delete the
second node.\n");
       return;
   }
   struct Node* temp = (*head)->next;
   (*head)->next = temp->next;
   free(temp);
}
void displayList(struct Node* head) {
   struct Node* temp = head;
   while (temp != NULL) {
       printf("%d -> ", temp->data);
       temp = temp->next;
   }
   printf("NULL\n");
}


sem_t bridge;
sem_t mutex;
sem_t mutext2;
int cars_on_bridge = 0;
int waiting_left = 0;
struct Node * headl;
struct Node * headr;
sem_t leftlock;
sem_t rightlock;
sem_t lock2;
int waiting_right = 0;
int tmpr=5;
// void pr(){
//     int k =0;
//     while (k<MAX_CARS) {
//         printf("%d ",buff[k
//         ]);
//         k++;
//     }
// }
int tmpl=5;
void passing(int dir, int car_num) {
   if (dir == 1) {
       sem_wait(&mutex);
       waiting_left++;
       sem_post(&mutex);
       sem_wait(&bridge);
```

```
        int flag = 0 ;
        while(tmpr<5 || tmpl<=0){
            if(flag++ == 0){
                printf("Car with id %d is waiting on left side as no space
left on bridge .. \n" , car_num) ;
            // sleep(
            //     1);
            }
        }
        sem_wait(&leftlock);
        // pthread_mutex_lock(&leftlock);
        printf("Car from left side with ID %d is crossing the bridge.\n",
car_num);
        sem_wait(&mutext2);
        struct Node *node = headl;
        append(&headl, car_num);
        // buff[car_num-1] =1;
        sem_post(&mutext2);
        // pr();
        tmpl--;
        // sem_post(&mutext2);
        sem_post(&leftlock);
        // pthread_mutex_unlock(&leftlock);
        sleep(1);
        // sem_wait(&mutext2);
        sem_wait(&leftlock);
        // pthread_mutex_lock(&leftlock);
        if (headl->next->data==car_num){
            sem_wait(&mutext2);
            pop(&headl);
            // displayList(headl);
            sem_post(&mutext2);
        }
        else{
            // pthread_mutex_lock(&mutext2);
            sem_wait(&mutext2);
            struct Node* n = headl;
            // append(&headl, car_num);
            car_num = headl->next->data;
            pop(&headl);
            // displayList(headl);
            sem_post(&mutext2);
        }
        // pthread_mutex_unlock(&leftlock);
        sem_post(&leftlock);
        // sem_post(&mutext2);
        // pr();
        sem_wait(&mutext2);
        sem_wait(&lock2);
        // pthread_mutex_lock(&lock2);
        printf("Car from left side with ID %d crossed the bridge.\n",
car_num);
        sem_post(&lock2);
        // pthread_mutex_unlock(&lock2);
        // sem_post(&lock2);
        sem_post(&mutext2);
        tmpl++;
        sem_post(&leftlock);
        // pthread_mutex_unlock(&leftlock);
        sem_wait(&mutex);
        waiting_left--;
        sem_post(&mutex);

        sem_post(&bridge);
    } else {
        sem_wait(&mutex);
        waiting_right++;
        sem_post(&mutex);

        sem_wait(&bridge);
        int flag = 0 ;
        while(tmpl<5 || tmpr<=0){
            if(flag++ == 0){
                printf("Car with id %d is waiting on right side as no space
left on bridge .. \n" , car_num) ;
            // sleep(1);
            }
        }
        sem_wait(&rightlock);
        // pthread_mutex_lock(&rightlock);
        printf("Car from right side with ID %d is crossing the bridge.\n",
car_num);
        sem_wait(&mutext2);
        // buff2[car_num-1] =1;
        struct Node *node = headr;
        append(&headr, car_num);
        sem_post(&mutext2);
        tmpr--;
        sem_post(&rightlock);
        // pthread_mutex_unlock(&rightlock);
        sleep(1);
        // pthread_mutex_lock(&rightlock);
        sem_wait(&rightlock);
        // pthread_mutex_lock(&rightlock);
        if (headr->next->data==car_num){
            sem_wait(&mutext2);
            pop(&headr);
            sem_post(&mutext2)  ;
        }
        else{
            sem_wait(&mutext2);
            struct Node* n = headr;
            // append(&headr, car_num);
            car_num = headr->next->data;
            pop(&headr);
            sem_post(&mutext2);
        }
        sem_post(&rightlock);
        // pthread_mutex_unlock(&rightlock);
        // if (flag==0)
        sem_wait(&mutext2);
        printf("Car from right side with ID %d crossed the bridge.\n",
car_num);
        sem_post(&mutext2);
        tmpr++;
        // pthread_mutex_unlock(&rightlock);

        sem_wait(&mutex);
        waiting_right--;
        sem_post(&mutex);
```

```c
        sem_post(&bridge);
    }
}

void *left(void *args) {
    // sleep(10);
    int car_num = *((int *)args);
    passing(1, car_num);
    pthread_exit(NULL);
}

void *right(void *args) {
    int car_num = *((int *)args);
    passing(2, car_num);
    pthread_exit(NULL);
}

int main() {
    int num_left, num_right;
    printf("Enter the number of cars on the left side: ");
    scanf("%d", &num_left);
    printf("Enter the number of cars on the right side: ");
    scanf("%d", &num_right);
    headl = createNode(0);
    headr = createNode(0);
    pthread_t left_threads[MAX_CARS], right_threads[MAX_CARS];
    sem_init(&bridge, 0, MAX_ON_BRIDGE);
    sem_init(&mutex, 0, 1);
    sem_init(&mutext2, 0, 1);
    sem_init(&leftlock,0,1);
    sem_init(&rightlock,0,1);
    sem_init(&lock2,0,1);
    for (int i = 0; i < num_left; ++i) {
        int *arg = (int *)malloc(sizeof(*arg));
        *arg = i + 1;
        if (pthread_create(&left_threads[i], NULL, left, arg)==-1){
            perror("Threading failed");
            exit(-1);
        };
    }

    for (int i = 0; i < num_right; ++i) {
        int *arg = (int *)malloc(sizeof(*arg));
        *arg = i + 1;
        if (pthread_create(&right_threads[i], NULL, right, arg)==-1){
            perror("Threading failed");
            exit(-1);
        };
    }

    for (int i = 0; i < num_left; ++i) {
        if (pthread_join(left_threads[i], NULL)==-1){
            perror("Thread join unsuccessfull");
            exit(-1);
        };
    }

    for (int i = 0; i < num_right; ++i) {
        if (pthread_join(right_threads[i], NULL)==-1){
            perror("Thread join unsuccessfull");
            exit(-1);
        };
    }

    if (sem_destroy(&bridge)==-1){
        perror("Semaphore cant be destroyed");
        exit(-1);
    };
    if (sem_destroy(&mutex)==-1){
        perror("Semaphore cant be destroyed");
        exit(-1);
    };

    return 0;
}
```

END OF DOCUMENT