OPERATING SYSTEMS
SEMAPHORES SYNCHRONISATION

ANUDEEP RAO PERALA

Contents

1 Goal :

- Using semaphores to synchronize the P passenger processes and the C car processes and analyzing time of execution based on changing values of P, C.

2 Input :

- Input is the set of numbers:
  1. Number of Passenger threads (P)
  2. Number of Cars (C)
  3. Mean value of exponential wait time between 2 successive ride requests made by the passenger (λ_P)
  4. Mean value of exponential wait time between 2 successive ride request accepted by a car (λ_C)

3 Output :

- The output data is written to output file *output.txt* which contains all log for all the rides.

4 Procedure :

- Input is read from the file **input.txt** and stored in variables:
  1. P - stores Passengers value
  2. C - stores Consumers value
  3. $lamb_1$ - stores value of λ_1
  4. $lamb_2$ - stores value of λ_2
  5. K - stores value of number of ride requests by each passenger

```
int P = 0;
int C = 0;
int lamb1 = 0;
int lamb2 = 0;
int K = 0;
```

- The mutex variables:
  1. **ret** - This vector array stores all the strings outputted from the program which are needed to be written to the **output.txt** file.
  2. **cars_availability** - The array which keeps track of which car is free, and which car is already in ride.
  3. **mtx_writing** - is used for locking when there is a new string addition to the vector array **ret** is done.
  4. **sem_array** - This is a counting semaphore, which works based on the number of cars free for ride.
  5. **total_time_passengers** - This adds up the total time for the whole process execution for each passenger, then we use it for calculating the average time for each set of $(P, C, \lambda_1, \lambda_2, K)$

6. **total_time_cars** - This adds up the total time for the whole process execution for each car, then we use it for calculating the average time for each set of $(P, C, \lambda_1, \lambda_2, K)$

```
vector<string> ret;
int *cars_availability;
pthread_mutex_t mtx_writing;
sem_t *sem_array;
double total_time_passengers = 0;
double total_time_cars = 0;
```

- This structure is used to send arugument to the *passenger*() whose parameters are :

  1. **passenger_number** - This varable stores the serial number of passenger
  2. **cars** - This is the array which has the thread ID's of the cars threads.
  3. **attr** - The default paramter for thread creation

```
struct data_to_passenger_func
{
  int passenger_number;
  pthread_t *cars;
  pthread_attr_t attr;
};
typedef struct data_to_passenger_func parampass;
```

- This structure is used to send arugument to the *cars*() whose parameters are :

  1. **car_number** - This varable stores the serial number of car in use by the current passenger.
  2. **passenger_number** - This varable stores the serial number of passenger
  3. **time_sleep** - This is the time for which the car thread sleeps so that the given constraint of not accepting new request is satisfied

```
struct data_to_car_func
{
  int car_number;
  int passenger_number;
  double time_sleep;
};
typedef struct data_to_car_func paramcar;
```

- The following code block corresponds to all the intialization of mutex, semaphore variable and the availability array for the cars.

```
// intializing the mutex variable
  pthread_mutex_init(&mtx_writing, nullptr);

// initializing semaphore variable to cars count
  sem_array = sem_open("/arraysem", O_CREAT | O_EXCL, 0644, C);
```

```
// initializing the cars availability array
  cars_availability = (int *)calloc(C, sizeof(int));
  for (int i = 0; i < C; i++)
    cars_availability[i] = 1;
```

- Creating passenger threads, with the pointer variable of **parampass struct** being passed as argument with suitable values loaded in.
- Control of program is now shifted from *main*() to *passengers*().

```
  pthread_t passengers[P]; // This is the thread pool of passengers
  pthread_t cars[C];       // This is the thread pool of cars
  pthread_attr_t attr;
  pthread_attr_init(&attr);

  for (int i = 0; i < P; i++)
    {
        auto *dat = (parampass *)calloc(1, sizeof(parampass));
        dat->passenger_number = i + 1;
        dat->cars = cars;
        dat->attr = attr;
        pthread_create(&passengers[i], &attr, passenger, dat);
      }
```

- The following code block is used for generation of the random times used in the program in the critical section of *passenger*() and *cars*() threads.

```
  int seed = chrono::system_clock::now().time_since_epoch().count();
    default_random_engine generator(seed);
    exponential_distribution<double> t_1(1.0 / lamb1);
    exponential_distribution<double> t_2(1.0 / lamb2);

    double t1 = t_1(generator);
    double t2 = t_2(generator);
```

- We use the mutex lock **mtx_writing**, whenever we want to add a string value to the vector array **ret**.

```
  pthread_mutex_lock(&mtx_writing);
    /*
      Adding a new string to ret vector array
    */
  pthread_mutex_unlock(&mtx_writing);
```

- Whenever we want to find the local time the above code block is used. From the variable **timeinfo** we can get the required info about local time by accessing the struct data.

```
  time_t raw_time;
  struct tm *timeinfo;
    time(&raw_time);
  timeinfo = localtime(&raw_time);
```

- The following code block is the entry section for the passenger thread.
- The semaphore variable **sem_array** ensures that a passenger thread can only access the **car_availability** array when there is a car available for riding i.e., **sem_array**> 0.
- After crossing the *sem_wait*() function search for a free car is done, after finding it the index of car it is stored in variable **index_car** variable which is then passed on to thread function *cars*().
- After loading required values to argument for passing we create a car thread with the **pthread_t** variable **dat− >cars[index_car]**.

```
// Entry section of the passenger thread
  for (int j = 0; j < K; j++)
  {
    int index_car = 0;
// Blocked this section inorder to remove race conditions
// for array value choosing
    sem_wait(sem_array);
    for (int i = 0; i < C; i++)
    {
      if (cars_availability[i] == 1)
      {
        index_car = i;
        cars_availability[i] = 0;
        break;
      }
    }

  paramcar *a = (paramcar *)calloc(1, sizeof(paramcar));
  a->passenger_number = dat->passenger_number;
  a->car_number = index_car + 1;
  a->time_sleep = t2;
  pthread_create(&dat->cars[index_car], &dat->attr, cars, a);
  pthread_join(dat->cars[index_car], nullptr]);
  cars_availability[index_car] = 1;
  sem_post(sem_array); // Incrementing the sem_array value
```

- Now here the program control shifts from *passengers*() to *cars*().
- The following code block is the Entry Section and CS of the car thread.
- The sleep time is in μ sec.

```
// Entry section for car thread
  pthread_mutex_unlock(&mtx_writing);

// CS for car thread
  usleep(d->time_sleep*1000);
```

- We use the following code block to calculate time of execution of an action.
- The output time is in μ seconds.

```
  auto start = high_resolution_clock::now();
/*
  The required action is performed
*/
    auto stop = high_resolution_clock::now();
  auto timees = duration_cast<microseconds>(stop - start);
  cout << (double)timees.count() ;
```

- After execution of Remainder Section control of the program shifts from *cars*() to *passengers*().
- The function *pthread_join*() ensures that the *car* thread is executed fully.
- Then the **cars_availibity** arrays value is resette to 1. Then semaphore value is incremented using *sem_post*().

```
  pthread_join(dat->cars[index_car], nullptr);
    cars_availability[index_car] = 1;
    sem_post(sem_array);
```

- The critical section of passenger depends on t_1, t_2 if $t_1 > t_2$ then we need to still wait for a time of $t_1 - t_2$, for the passenger to pass new request to a car. But when $t_1 < t_2$, in this case as already the passenger has passed the time t_1 passenger can putforth a new request as soon as the passeger exits the car.

```
// CS of passenger thread
  if (t1 > t2)
  usleep((int)(t1-t2)*1000);
```

- The average Time of execution of Passengers and Cars is calculated like as shown below.

```
cout << "Total avg time of execution of passengers is in secs : " <<
                total_time_passengers / P << endl;
cout << "Total avg time of execution of cars is in secs : " <<
                 total_time_cars / C << endl;
```

- File writing is done with the global vector array **ret**.
- The output to file is not in the order that was mentioned in the problem statement, this was done in purpose inorder to prove the concurrency of the program.
- In the output file the system time of the output can be seen it will not change this is because as we considered the sleep times in milli seconds.

```
  ofstream op;
  op.open("output.txt", ios_base::out);
   for (const auto & i : ret)
  {
    op << i;
  }
```

5 Analyzing Program Output :

- We analyze the program in two modes:

  1. T_{avg} for passengers to complete their tour with Total number of Cars as constant (C is constant).
  2. T_{avg} for Cars to complete their tour with Total number of Passengers as constant (P is constant).

5.1 C is constant :

- The value of C used by the program is fixed and is equal to 25.
- The value of K used by the program is equal to 5
- λ_1 is equal to 10
- λ_2 is equal to 10

5.1.1 Graph :

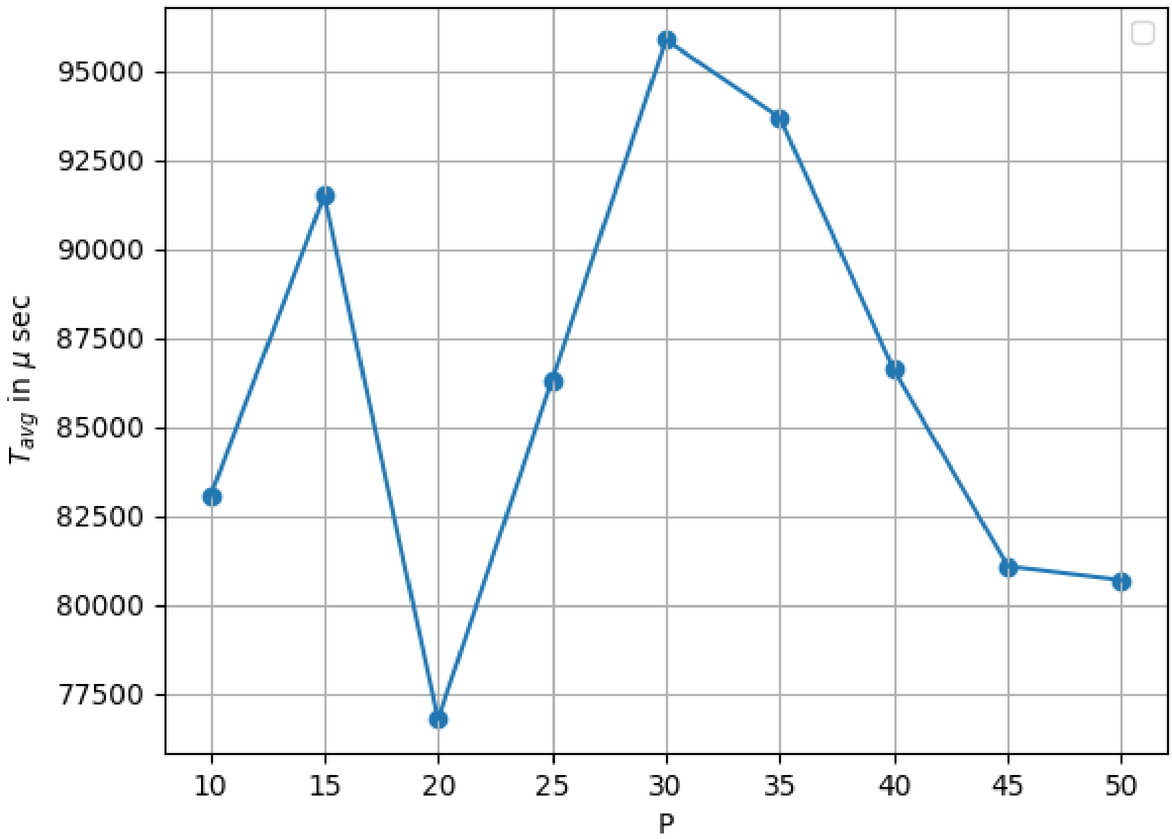

Figure 1: T_{avg} vs P (C is constant)

5.1.2 Observations :

- **Expected** :
  - As the value of P increases T_{avg} should increase as C is constant waiting time of each passenger increases.
- **Observed** :
  - We observe some unusual trend in the T_{avg} of this case.
  - We see that there isn't much difference between the average times between the values of $P = 10$ to $P = 25$ which is because of the fact that the value of $C = 25$ we have more cars than passengers, because of this the time differences are not much high in this range of P, i.e., the waiting time isn't that high.
  - The value of T_{avg} increases as expected in $P = 20$ to $P = 30$.
  - Except for the values at $P = 20$ and $P = 30$ all other T_{avg} are nearly at the same times, as we are measuring the time in μ seconds.
  - This irregular time variations can be attributed to hardware implementation of the PC.

5.2 P is constant :

- The value of P used by the program is fixed and is equal to 50.
- The value of K used by the program is equal to 3
- λ_1 is equal to 10
- λ_2 is equal to 10

5.2.1 Graph :

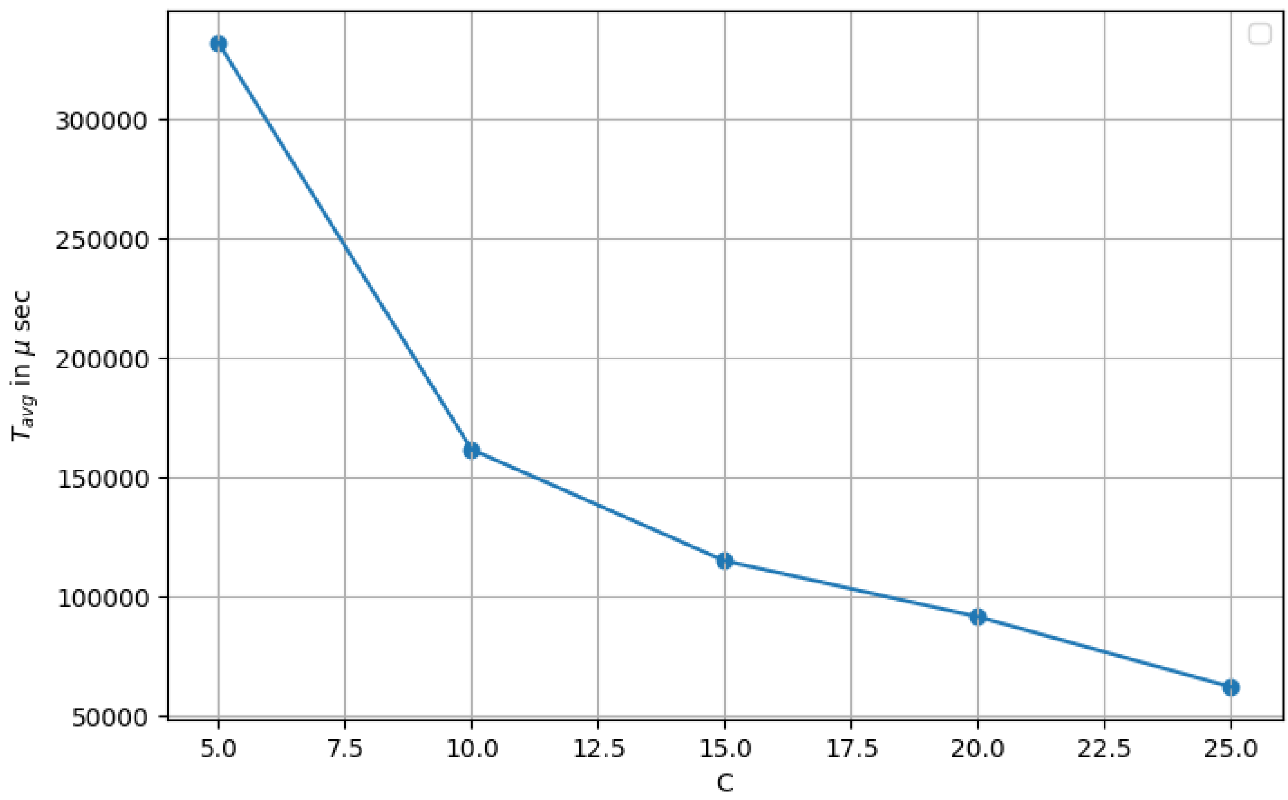

Figure 2: T_{avg} vs C (P is constant)

5.2.2 Observations :

- **Expected** :
  - As the value of C increases the average working time decreases.
- **Observed** :
  - As the value of C increases T_{avg} decreases, which follows the expected pattern.
  - This is because as the number of cars increases work load on each car decreases which inturn decreases the time of ride for each car which results in decrease of T_{avg}.