Bahria University,

Karachi Campus



COURSE: Operating System

TERM: FALL 2022, CLASS: BSE- 4 (B)

PROJECT NAME: Ice Cream Factory

Group Members:

Abdul QUDDOS 02-131202-033

Mustufa 02-131202-012

Syed Abdullah 02-131202-066

Farhan Ali 02-131192-063

Submitted to

Engr. Rahemeen Khan

Signed Remarks: Score:

TABLE OF CONTENT

1. INTRODUCTION & PROBLEM
2. TECHNOLOGY
3. METHODOLOGY
4. MODULE DISTRIBUTION
5. RESULT
6. CONCLUSION

REFERENCES

**Introduction & Problem:**

Before start demonstrating the project first we talk about why we use to select this project and why we were worked on it? The problem before that how we do tackle deadlock when no. of threads increasing, multiprocessing, fast access to the system resources, how to utilize resources in the minimum time and get maximum throughput, avoid extra wait acquired by the processes and avoiding race condition.

**Technology:**

* + - 1. Linux Ubuntu operating System.
      2. C programming language.
      3. Procedural programming paradigm.
      4. Console based application.

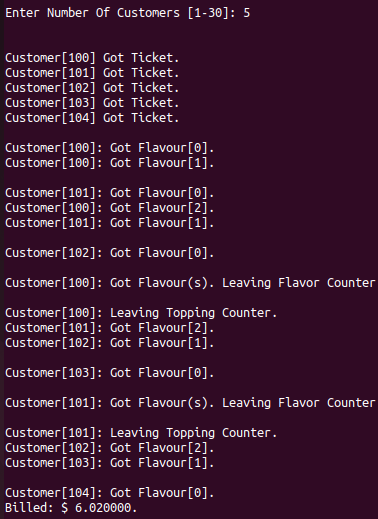
**Methodology:**

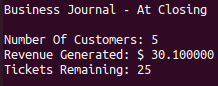
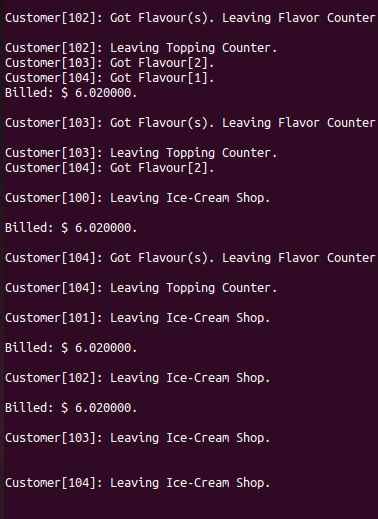
For the above described problem solving we designed an ice-cream factory this program creates a simulation of an ice cream shop where multiple customers are entertained with preventing race condition. It is written in pure C only. An ice-cream factory problem in which we call system/kernel libraries for fast access to the resources and we implementing threads for concurrent processing for fast c.p.u utilization, firstly we allocate resources like Customers, toppings, flavours, payment and tickets set random values to it. we initializes semaphores in the project to monitor the resource allocation and de-allocation i.e. In ice-cream problem The whole Process is ice-cream factory we divided into threads that no. of customers entered for acquiring ice-cream, no of ice creams, toppings. By using semaphores Each type of semaphore i.e. customers, toppings, ice-cream, payment and ticket to get ice-cream we manage the activities of critical section to give resources to the ticketed customer. Basically we have three flavours in the ice-cream if any customer didn’t get any flavour then race condition is equal to 0. Otherwise check priority of the threads.

Here is the code of c programming which we were implemented for this problem:

**C Code:**

|  |
| --- |
| #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <unistd.h> |
|  | #include <pthread.h> |
|  | #include <semaphore.h> |
|  | #include <string.h> |
|  | #include <sys/syscall.h> |
|  | #include <linux/kernel.h> |
|  | // prices of flavours and toppings |
|  |  |
|  | #define priceFlav\_1 1.05 |
|  | #define priceFlav\_2 2.00 |
|  | #define priceFlav\_3 1.67 |
|  | #define priceTopp\_1 0.8 |
|  | #define priceTopp\_2 0.5 |
|  |  |
|  | // resources |
|  | int ticket = 30, \_flavors[3] = {29, 34, 18}, \_toppings[2] = {20, 34}; |
|  | double revenue = 0.0; |
|  |  |
|  | // semaphores declaration |
|  | sem\_t ticketC, flavorC, f1, f2, f3, toppingC, t1, t2, paymentC; |
|  |  |
|  | // function prototypes |
|  | void \*iceCream(void \*\_id); |
|  |  |
|  | // main |
|  | int main() |
|  | { |
|  | int noC; |
|  |  |
|  | printf("\n\nEnter Number Of Customers [1-%d]: ", ticket); |
|  | scanf("%d", &noC); |
|  | printf("\n\n"); |
|  |  |
|  | if(noC > ticket || noC <= 0) |
|  | { |
|  | printf("\n\nInvalid Input!\n\n"); |
|  | return 0; |
|  | } |
|  |  |
|  | int \_id[noC]; |
|  |  |
|  | for(int i=0; i<noC; i++) |
|  | { |
|  | \_id[i] = i+100; |
|  | } |
|  |  |
|  | // semaphore initialization - START |
|  | sem\_init(&ticketC, 0, 1); |
|  |  |
|  | sem\_init(&flavorC, 0, 3); |
|  | sem\_init(&f1, 0, 1); |
|  | sem\_init(&f2, 0, 1); |
|  | sem\_init(&f3, 0, 1); |
|  |  |
|  | sem\_init(&toppingC, 0, 2); |
|  | sem\_init(&t1, 0, 1); |
|  | sem\_init(&t2, 0, 1); |
|  |  |
|  | sem\_init(&paymentC, 0, 1); |
|  |  |
|  | // semaphore initialization - END |
|  |  |
|  |  |
|  | // multithreading region - START |
|  | pthread\_t \_customer[noC]; |
|  |  |
|  | for(int i=0; i<noC; i++) |
|  | { |
|  | pthread\_create(&\_customer[i], 0, &iceCream, (void\*) &\_id[i]); |
|  | } |
|  |  |
|  | for(int i=0; i<noC; i++) |
|  | { |
|  | pthread\_join(\_customer[i], NULL); |
|  | } |
|  | // multithreading region - END |
|  |  |
|  |  |
|  | printf("\n\nBusiness Journal - At Closing\n\n"); |
|  | printf("Number Of Customers: %d", noC); |
|  | printf("\nRevenue Generated: $ %f", revenue); |
|  | printf("\nTickets Remaining: %d\n\n", ticket); |
|  |  |
|  |  |
|  | // semaphore destroying - START |
|  |  |
|  | sem\_destroy(&ticketC); |
|  |  |
|  | sem\_destroy(&flavorC); |
|  | sem\_destroy(&f1); |
|  | sem\_destroy(&f2); |
|  | sem\_destroy(&f3); |
|  |  |
|  | sem\_destroy(&toppingC); |
|  | sem\_destroy(&t1); |
|  | sem\_destroy(&t2); |
|  |  |
|  | sem\_destroy(&paymentC); |
|  |  |
|  | // semaphore destroying - END |
|  |  |
|  | return 0; |
|  | } |
|  | void \*iceCream(void \*\_id) |
|  | { |
|  | int \_ID = \*(int\*)\_id, checkRaceCond\_1 = 0; |
|  | double bill = 0.0; |
|  |  |
|  | // ticket counter - ENTER |
|  | sem\_wait(&ticketC); |
|  |  |
|  | if(ticket <= 0) |
|  | { |
|  | //printf("\nCustomer[%d]: Leaving Shop. [REASON]: Tickets Finished\n", \_ID); |
|  | syscall(333,"Leaving Shop. [REASON]: Tickets Finished\n",\_ID); |
|  | sleep(1); |
|  | pthread\_exit(NULL); |
|  | } |
|  |  |
|  | ticket--; |
|  | //printf("Customer[%d] Got Ticket.\n", \_ID); |
|  | syscall(333,"Got Ticket.\n",\_ID); |
|  |  |
|  | sem\_post(&ticketC); |
|  | // ticket counter - EXIT |
|  |  |
|  | sleep(2); |
|  |  |
|  | // flavours counter - ENTER |
|  | sem\_wait(&flavorC); |
|  |  |
|  | // Race Condition will occur here but it will be handled through thread Local Variable |
|  | // [REASON]: when this condition is checked by 4th thread |
|  | // and then \_flavours[i] is decremented by any of first 2 threads |
|  | // a thread will proceed even though flavors have been finished. |
|  | // WHY THIS CONDITION IS USED WHEN ITS WORK IS DONE BY LOCAL THREAD VARIABLE? |
|  | // [REASON]: If this is not used then a thread will check each condition below |
|  | // which will be time consuming. |
|  | if(\_flavors[0] <= 0 && \_flavors[1] <= 0 && \_flavors[2] <= 0) |
|  | { |
|  | syscall(333,"Leaving Shop. [REASON]: Flavours Finished\n", \_ID); |
|  | sleep(1); |
|  | pthread\_exit(NULL); |
|  | } |
|  | else |
|  | { |
|  | // flavor1 |
|  | sem\_wait(&f1); |
|  |  |
|  | if(\_flavors[0] > 0) |
|  | { |
|  | \_flavors[0]--; |
|  | checkRaceCond\_1++; |
|  | bill = bill + priceFlav\_1; |
|  | //printf("\nCustomer[%d]: Got Flavour[0].\n", \_ID); |
|  | syscall(333,"Got Flavour[0].\n",\_ID); |
|  | sleep(1); |
|  | } |
|  |  |
|  | sem\_post(&f1); |
|  |  |
|  | // flavor2 |
|  | sem\_wait(&f2); |
|  |  |
|  | if(\_flavors[1] > 0) |
|  | { |
|  | \_flavors[1]--; |
|  | checkRaceCond\_1++; |
|  | bill = bill + priceFlav\_2; |
|  | //printf("Customer[%d]: Got Flavour[1].\n", \_ID); |
|  | syscall(333,"Got Flavour[1].\n",\_ID); |
|  | sleep(1); |
|  | } |
|  |  |
|  | sem\_post(&f2); |
|  |  |
|  | // flavor3 |
|  | sem\_wait(&f3); |
|  |  |
|  | if(\_flavors[2] > 0) |
|  | { |
|  | \_flavors[2]--; |
|  | checkRaceCond\_1++; |
|  | bill = bill + priceFlav\_3; |
|  | //printf("Customer[%d]: Got Flavour[2].\n", \_ID); |
|  | syscall(333,"Got Flavour[2].\n",\_ID); |
|  | sleep(1); |
|  | } |
|  |  |
|  | sem\_post(&f3); |
|  |  |
|  | // if any of the threads did not get any flavour, |
|  | // 'checkRaceCond\_1' will remain 0, |
|  | // threads will exit |
|  | if(checkRaceCond\_1 == 0) |
|  | { |
|  | //printf("\nCustomer[%d]: Leaving Shop. [REASON]: Flavours Finished\n", \_ID); |
|  | syscall(333,"Leaving Shop. [REASON]: Flavours Finished\n",\_ID); |
|  | sleep(1); |
|  | pthread\_exit(NULL); |
|  | } |
|  | } |
|  |  |
|  | //printf("\nCustomer[%d]: Got Flavour(s). Leaving Flavor Counter\n", \_ID); |
|  | syscall(333,"Got Flavour(s). Leaving Flavor Counter\n",\_ID); |
|  | //sleep(2); |
|  |  |
|  | sem\_post(&flavorC); |
|  | // flavors counter - EXIT |
|  |  |
|  |  |
|  | // toppings counter - ENTER |
|  | sem\_wait(&toppingC); |
|  |  |
|  | // topping1 |
|  | sem\_wait(&t1); |
|  |  |
|  | if(\_toppings[0] > 0) |
|  | { |
|  | \_toppings[0]--; |
|  | bill = bill + priceTopp\_1; |
|  | } |
|  |  |
|  | sem\_post(&t1); |
|  |  |
|  | // topping2 |
|  | sem\_wait(&t2); |
|  |  |
|  | if(\_toppings[1] > 0) |
|  | { |
|  | \_toppings[1]--; |
|  | bill = bill + priceTopp\_2; |
|  | } |
|  |  |
|  | sem\_post(&t2); |
|  |  |
|  | //printf("\nCustomer[%d]: Leaving Topping Counter.\n", \_ID); |
|  | syscall(333,"Leaving Topping Counter.\n",\_ID); |
|  |  |
|  | sem\_post(&toppingC); |
|  | // toppings counter - EXIT |
|  |  |
|  | sleep(2); |
|  |  |
|  | // payments counter - ENTER |
|  | sem\_wait(&paymentC); |
|  |  |
|  | revenue = revenue + bill; |
|  |  |
|  | //printf("\nCustomer[%d]: Billed: $ %f.\n", \_ID, bill); |
|  | char a[100]; |
|  | sprintf(a,"Billed: $ %f.\n",bill); |
|  | syscall(333,a,\_ID); |
|  |  |
|  | sem\_post(&paymentC); |
|  | // payments counter - EXIT |
|  |  |
|  | sleep(2); |
|  |  |
|  | //printf("\nCustomer[%d]: Leaving Ice-Cream Shop.\n\n", \_ID); |
|  | syscall(333,"Leaving Ice-Cream Shop.\n\n",\_ID); |
|  |  |
|  | return NULL; |
|  | }  **Possible Outputs:** |

****

****

**Module distribution:**

**Result:**

Through Above program we can achieve the concurrent processing approach at thread levels and get maximum throughput and avoid race condition and avoid deadlock.

**Conclusion:**

We can avoid deadlock and avoid extra wait acquired by the processes in the system by using semaphores and acquiring locks in the system that if one processes acquired that process then no one can acquire and if the process releases the lock then the other process can acquired the resource, we can maximize the throughput by multi-threading synchronization, we can acquire fast access to the resources by calling the system i.e. system calls. To avoid race condition, any operation on a shared resources must be executed automatically must be mutually exclusive.