EAST WEST UNIVERSITY

Department of Computer Science & Engineering

Semester: **Summer-2022**
Course Code: **CSE325**
Course Title: Operating System
Section: 01
Group: 02

Project Title(2): Burger Buddies Problem

Presented by
Name: Lotifa Akan Anannya
ID: 2019-3-60-110
Name: MD. Mottakin Rahat
ID: 2020-2-60-015
Name: Md. Jihad
ID: 2020-3-60-087
Name: Ab. Rahim Ahmed Sowrov
ID: 2020-3-60-070

Presented to
Nawab Yousuf Ali
Professor
Department of Computer Science & Engineering
East West University

Date of Submission: 04-09-2022

Abstract

In our following project, we will implement and test a solution for the IPC (Inter Process Communication) problem of burger buddies problem in which we will use semaphore and multi threads to execute our code in order to have a synchronization for a cashier presenting food to the customer. In this program, we will mainly focus on three threads: function [cook, customer, and cashier]. These three functions are related to each other with semaphore and created by threads. The following code will define some variables at the beginning of the program. But we can change it according to one's choice.

Introduction

An operating system (OS) is software that manages computer hardware and software resources while also providing common functions to computer programs. Time-sharing operating systems plan tasks to make the most of the system's resources, and they may also contain accounting software for cost allocation of processor time, storage, printing, and other resources. Although application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it, the operating system acts as an intermediary between programs and the computer hardware for hardware functions such as input and output and memory allocation. From cellular phones and video game consoles to web servers and supercomputers, operating systems are found on many devices that incorporate a computer.
In our "burger buddies' problem", we will focus on the three main topics. Threads, process and semaphore.

Threads: Within a process, a thread is a path of execution. Multiple threads can exist in a process. The lightweight process is also known as a thread. By dividing a process into numerous threads, parallelism can be achieved. Multiple tabs in a browser, for example, can represent different threads. MS Word makes use of numerous threads: one to format the text, another to receive inputs, and so on. Below are some more advantages of multithreading.

Process: A process is essential for running software. The execution of a process must be done in a specific order. To put it another way, we write our computer programs in a text file, and when we run them, they turn into a process that

completes all of the duties specified in the program. A program can be separated into four components when it is put into memory and becomes a process: stack, heap, text, and data. The diagram below depicts a simplified structure of a process in main memory.

Semaphore: Dijkstra proposed the semaphore in 1965, which is a very important technique for managing concurrent activities using a basic integer value called a semaphore. A semaphore is just an integer variable shared by many threads. In a multiprocessing context, this variable is utilized to solve the critical section problem and establish process synchronization.
There are two types of semaphores:

1. **Binary Semaphore –**
   This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
2. **Counting Semaphore –**
   Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Similar Work:

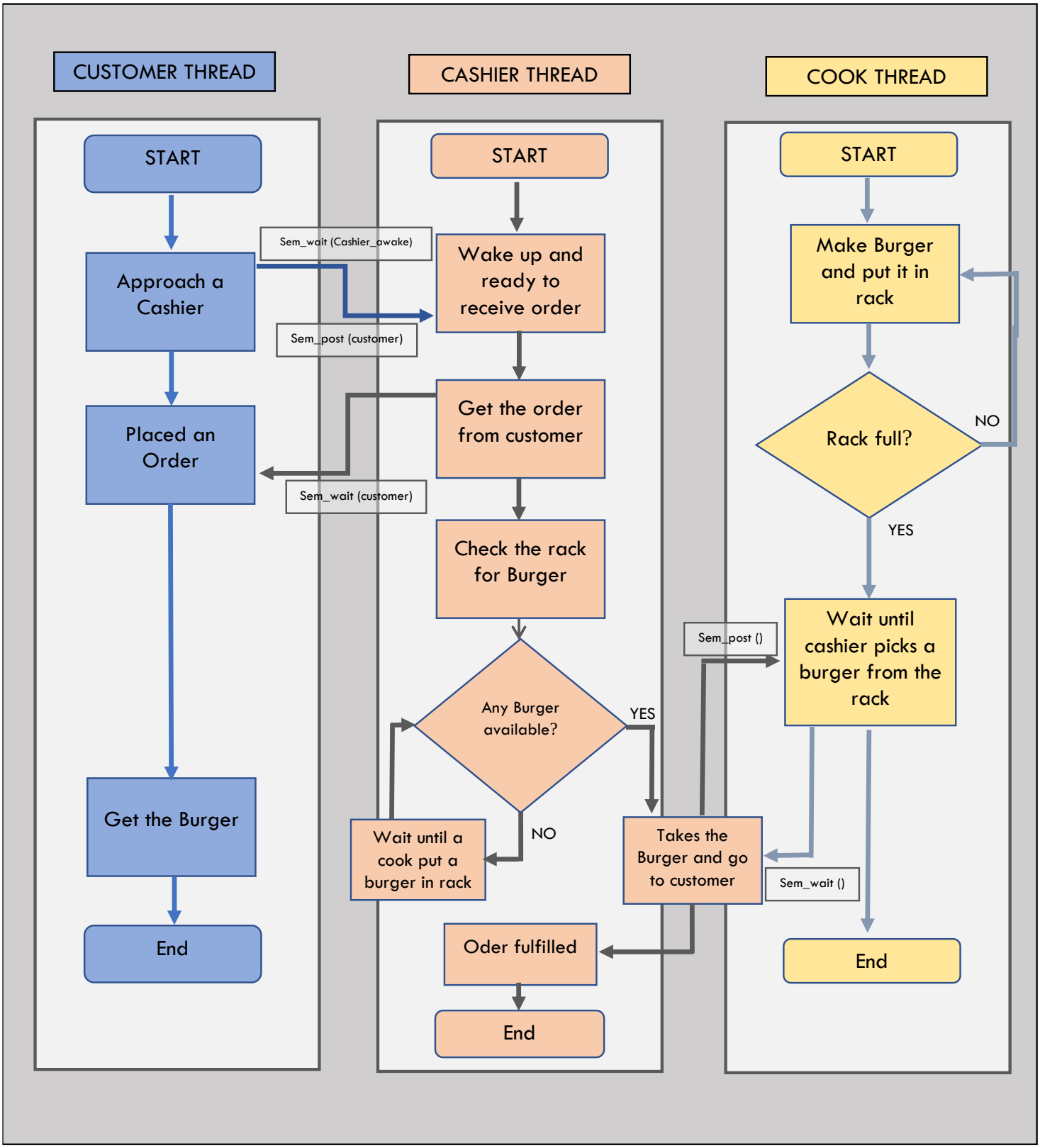
- **Dining Philosophers' Problems**
- **Producer Consumer Problem**
- **Barber Shop Problem**

Proposed Work

Suppose we have the following scenario: Cooks, Cashiers, and Customers are each modeled as a thread. Cashiers sleep until a customer is present. A Customer approaching a cashier can start the order process. A Customer cannot order until the cashier is ready. Once the order is placed, a cashier has to get a burger from the rack. If a burger is not available, a cashier must wait until one is made. The cook will always make burgers and place them on the rack. The cook will wait if the rack is full. There are NO synchronization constraints for a cashier presenting food to the customer. So, we will implement semaphore and multi threads to execute our code in order to have a synchronization.

Here, Cashiers thread wait until a customer is present when a customer thread arrive it will check whether the cashier thread is ready. Customer thread cannot order until the cashier is ready. When the cashier thread is ready customer thread will approach and place an order. Once the order is placed, a cashier thread has to get a burger from the rack. If a burger is not available, a cashier thread must wait until one is made. The cook thread will check whether the rack is full. If the rack is full the cook thread will wait until a burger is taken by cashier thread.

Flow Chart of the solution:


CUSTOMER THREAD
CASHIER THREAD
COOK THREAD
START
Approach a Cashier
Placed an Order
Get the Burger
End
Sem_wait (Cashier_awake)
Sem_post (customer)
Sem_wait (customer)
START
Wake up and ready to receive order
Get the order from customer
Check the rack for Burger
Any Burger available?
YES
NO
Wait until a cook put a burger in rack
Oder fulfilled
End
Sem_post ()
Takes the Burger and go to customer
Sem_wait ()
START
Make Burger and put it in rack
Rack full?
NO
YES
Wait until cashier picks a burger from the rack
End

C Program Code:

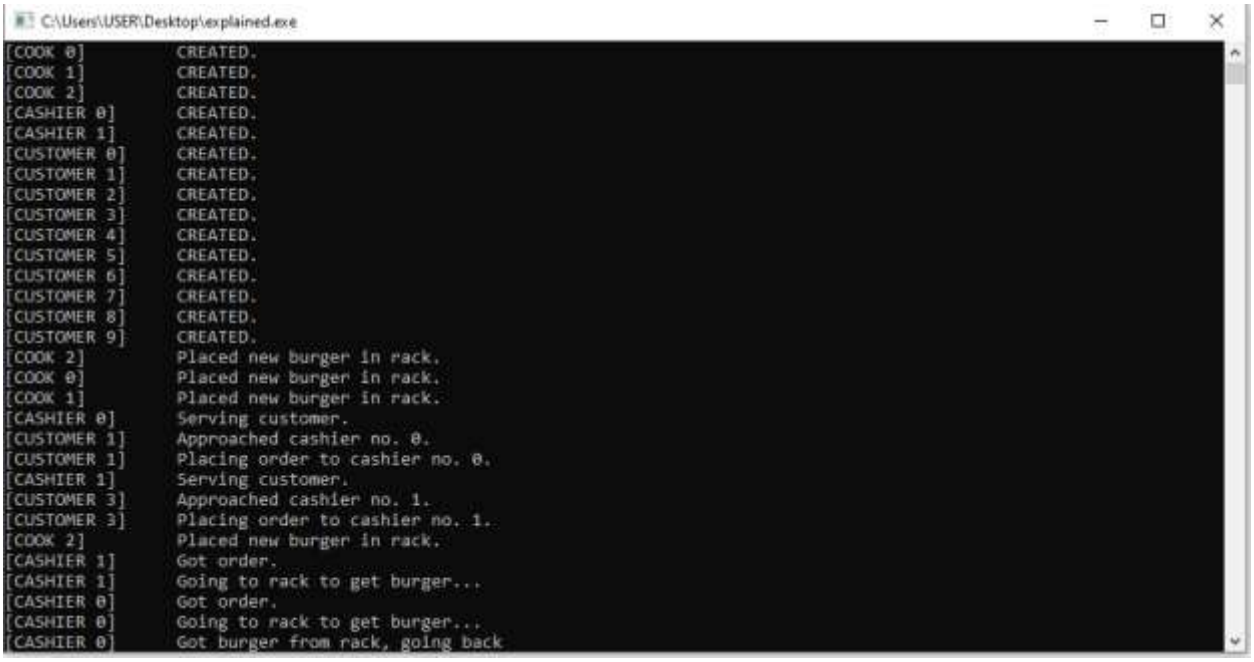
```c
///All this values are changeable
#define COOK_COUNT          3
#define CASHIER_COUNT       2
#define CUSTOMER_COUNT      10
#define RACK_HOLDER_SIZE    4
#define waiting_time        5

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <inttypes.h>
#include <stdbool.h>
#include <unistd.h>
#include <time.h>

bool interrupt = false;


typedef struct
{
    int id;
    sem_t *order;
    sem_t *burger;
} cashier_t;


typedef struct
{
    int id;
    sem_t *init_done;
} mos;

void *cook_run();
void *cashier_run();
void *customer_run();

void assure_state();

sem_t rack;
sem_t cook;
sem_t cashier;
sem_t cashier_awake;
sem_t customer;
sem_t customer_private_mutex;

cashier_t cashier_exchange;

int count = 0;

int main(int argc, char **argv)
{

    srand(time(NULL));
    sem_init(&rack, 0, 1);
    sem_init(&cashier, 0, 1);
    sem_init(&cashier_awake, 0, 0);
    sem_init(&cook, 0, RACK_HOLDER_SIZE);
    sem_init(&customer, 0, 0);
    sem_init(&customer_private_mutex, 0, 1);

```

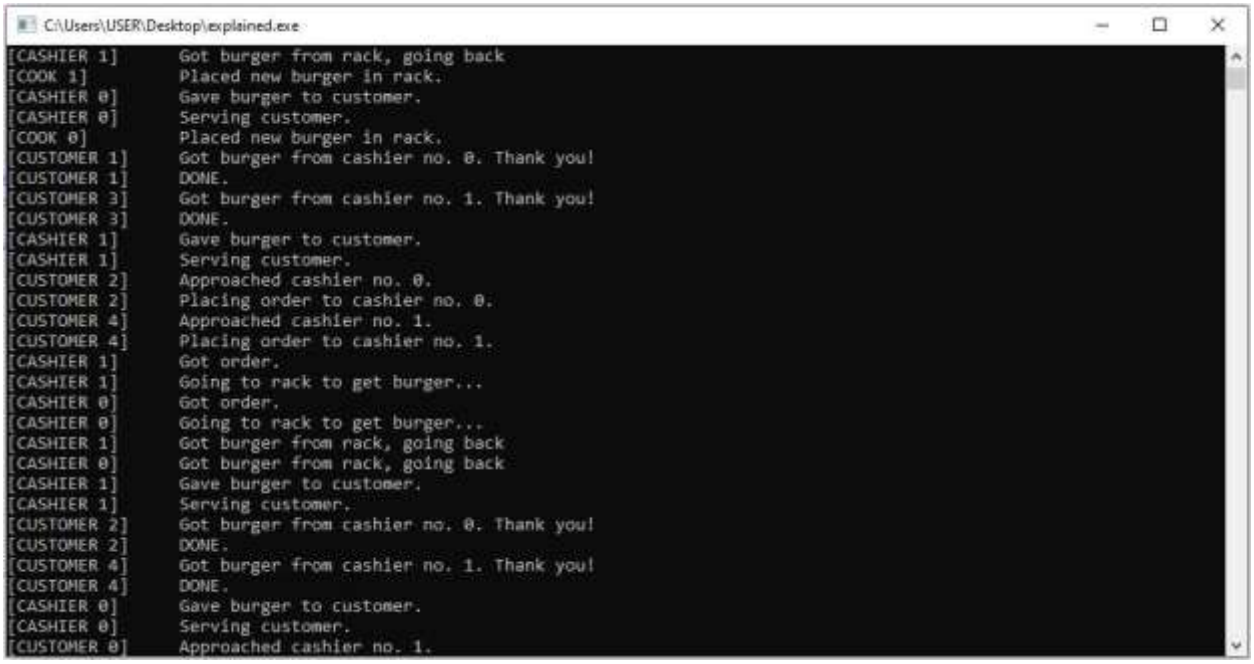
```
62
63      mos eas;
64      sem_t init_done;
65      sem_init(&init_done, 0, 0);
66      eas.init_done = &init_done;
67
68      pthread_t cooks[COOK_COUNT];
69      for(int i=0; i<COOK_COUNT; i++)
70      {
71          eas.id = i;
72          if(pthread_create(cooks+i, NULL, cook_run, (void*) &eas))
73          {
74              printf("[MAIN]\t\t ERROR: Unable to create cook thread.\n");
75              exit(1);
76          }
77          sem_wait(&init_done);
78      }
79
80      pthread_t cashiers[CASHIER_COUNT];
81      for(int i=0; i<CASHIER_COUNT; i++)
82      {
83          eas.id = i;
84          if(pthread_create(cashiers+i, NULL, cashier_run, (void*) &eas))
85          {
86              printf("[MAIN]\t\t ERROR: Unable to create cashier thread.\n");
87              exit(2);
88          }
89          sem_wait(&init_done);
90      }
91
92      pthread_t customers[CUSTOMER_COUNT];
93      for(int i=0; i<CUSTOMER_COUNT; i++)
94      {
95          eas.id = i;
96
97          if(pthread_create(customers+i, NULL, customer_run, (void*) &eas))
98          {
99              printf("[MAIN]\t\t ERROR: Unable to create customer thread.\n");
100             exit(3);
101         }
102         sem_wait(&init_done);
103     }
104
105     sem_destroy(&init_done);
106     for(int i=0; i<CUSTOMER_COUNT; i++)
107     {
108         if(pthread_join(customers[i], NULL))
109         {
110             printf("[MAIN]\t\t ERROR: Unable to join cutomers[%d]\n", i);
111             exit(4);
112         }
113     }
114
115     printf("[MAIN]\t\t SUCCESS: All threads terminated, state consistent.\n");
116 }
117
118 void *cook_run(void *eas)
119 {
120     mos *eas_ptr = (mos*) eas;
121     int cook_id = eas_ptr->id;
```

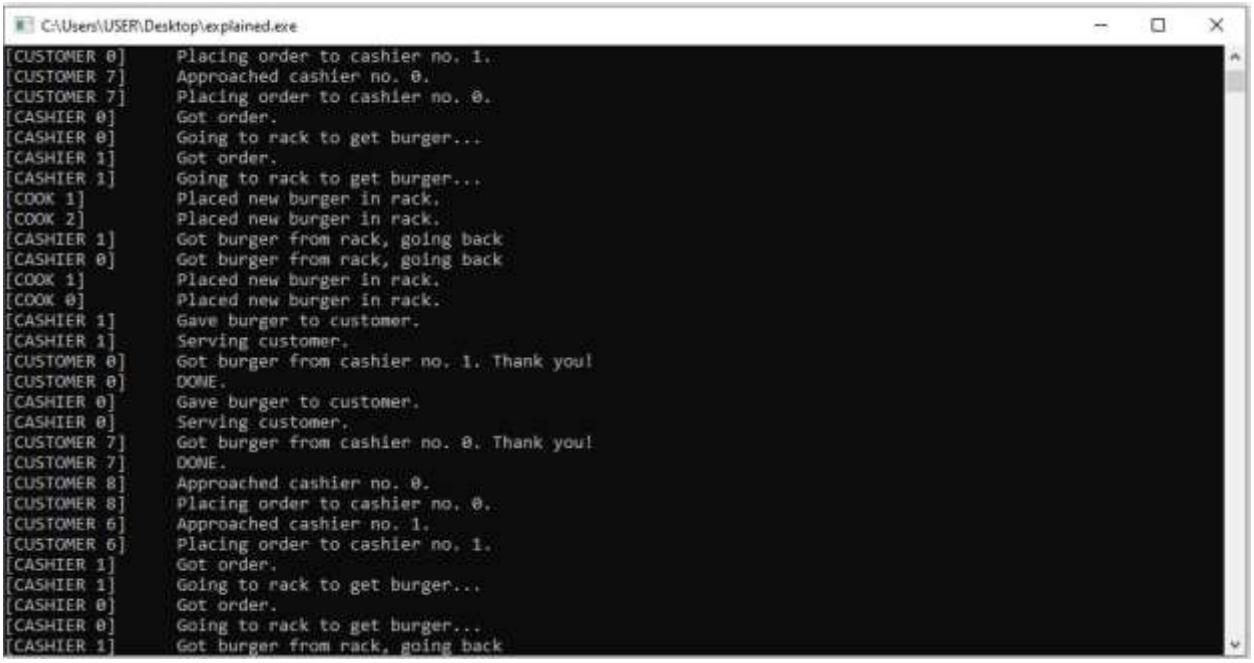
```
122     printf("[COOK %d]\t CREATED.\n", cook_id);
123     sem_post(eas_ptr->init_done);
124
125     while(1)
126     {
127         sem_wait(&cook);
128         if(interrupt) break;
129
130         sleep(rand() % waiting_time);
131
132         sem_wait(&rack);
133         count++;
134         sem_post(&rack);
135
136         printf("[COOK %d]\t Placed new burger in rack.\n", cook_id);
137
138         sem_post(&cashier);
139     }
140
141     printf("[COOK %d]\t DONE.\n", cook_id);
142     return NULL;
143 }
144
145 void *cashier_run(void *eas)
146 {
147     mos *eas_ptr = (mos*) eas;
148     int cashier_id = eas_ptr->id;
149
150     sem_t order;
151     sem_t burger;
152      sem_init(&order, 0, 0);
153      sem_init(&burger, 0, 0);
154
155      printf("[CASHIER %d]\t CREATED.\n", cashier_id);
156      sem_post(eas_ptr->init_done);
157
158
159      while(1)
160      {
161          sem_wait(&customer);
162          if(interrupt) break;
163          printf("[CASHIER %d]\t Serving customer.\n", cashier_id);
164
165          cashier_exchange.order = &order;
166          cashier_exchange.burger = &burger;
167          cashier_exchange.id = cashier_id;
168
169          sem_post(&cashier_awake);
170
171          sem_wait(&order);
172          printf("[CASHIER %d]\t Got order.\n", cashier_id);
173
174          printf("[CASHIER %d]\t Going to rack to get burger...\n", cashier_id);
175
176          sleep(rand() % waiting_time);
177
178
179          sem_wait(&cashier);
180
181          sem_wait(&rack);
```

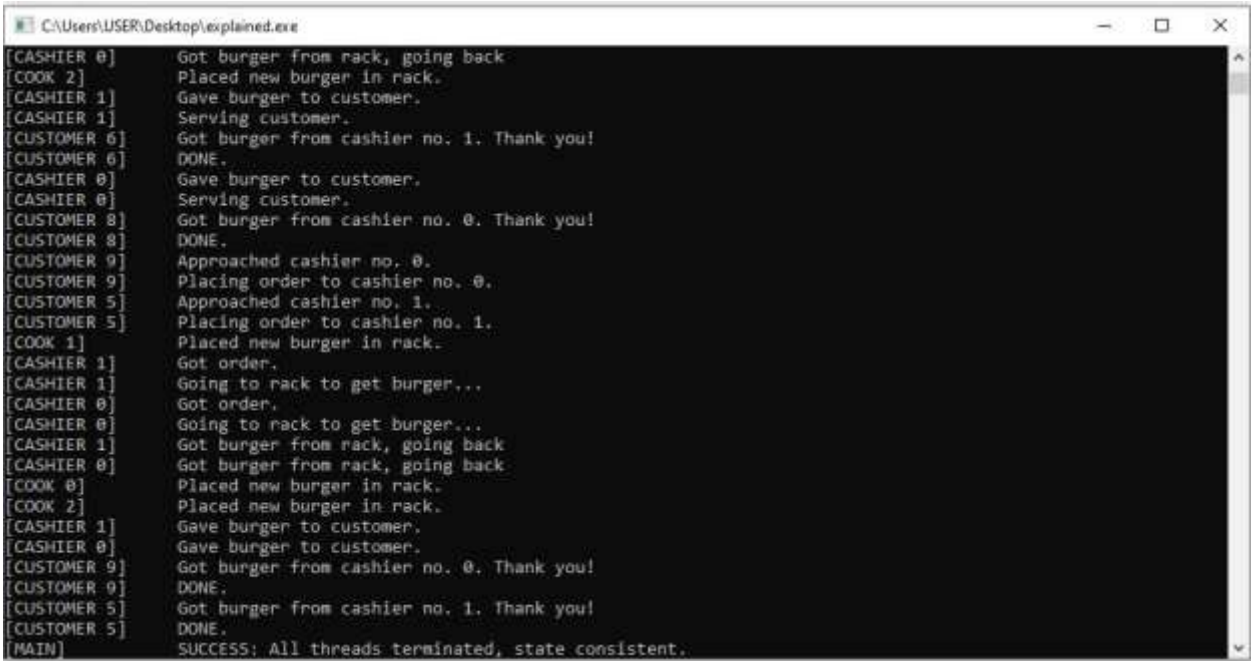
```
            count--;
            sem_post(&rack);
            sem_post(&cook);
            printf("[CASHIER %d]\t Got burger from rack, going back\n", cashier_id);
            sleep(rand() % waiting_time);
            sem_post(&burger);
            printf("[CASHIER %d]\t Gave burger to customer.\n", cashier_id);
        }
        sem_destroy(&order);
        sem_destroy(&burger);
        printf("[CASHIER %d]\t DONE.\n", cashier_id);

        return NULL;
    }

    void *customer_run(void *eas)
    {
        mos *eas_ptr = (mos*) eas;
        int customer_id = eas_ptr->id;
        printf("[CUSTOMER %d]\t CREATED.\n", customer_id);
        sem_post(eas_ptr->init_done);

        sleep(rand() % waiting_time + 1);

        sem_wait(&customer_private_mutex);

        sem_post(&customer);
        sem_wait(&cashier_awake);

        sem_t *order = cashier_exchange.order;
        sem_t *burger = cashier_exchange.burger;
        int cashier_id = cashier_exchange.id;

        sem_post(&customer_private_mutex);

        printf("[CUSTOMER %d]\t Approached cashier no. %d.\n", customer_id, cashier_id);
        printf("[CUSTOMER %d]\t Placing order to cashier no. %d.\n", customer_id, cashier_id);

        sleep(rand() % waiting_time);

        sem_post(order);
        sem_wait(burger);


        printf("[CUSTOMER %d]\t Got burger from cashier no. %d. Thank you!\n", customer_id, cashier_id);
        printf("[CUSTOMER %d]\t DONE.\n", customer_id);
        return NULL;

    }
```

Output:


C:\Users\USER\Desktop\explained.exe
[COOK 0] CREATED.
[COOK 1] CREATED.
[COOK 2] CREATED.
[CASHIER 0] CREATED.
[CASHIER 1] CREATED.
[CUSTOMER 0] CREATED.
[CUSTOMER 1] CREATED.
[CUSTOMER 2] CREATED.
[CUSTOMER 3] CREATED.
[CUSTOMER 4] CREATED.
[CUSTOMER 5] CREATED.
[CUSTOMER 6] CREATED.
[CUSTOMER 7] CREATED.
[CUSTOMER 8] CREATED.
[CUSTOMER 9] CREATED.
[COOK 2] Placed new burger in rack.
[COOK 0] Placed new burger in rack.
[COOK 1] Placed new burger in rack.
[CASHIER 0] Serving customer.
[CUSTOMER 1] Approached cashier no. 0.
[CUSTOMER 1] Placing order to cashier no. 0.
[CASHIER 1] Serving customer.
[CUSTOMER 3] Approached cashier no. 1.
[CUSTOMER 3] Placing order to cashier no. 1.
[COOK 2] Placed new burger in rack.
[CASHIER 1] Got order.
[CASHIER 1] Going to rack to get burger...
[CASHIER 0] Got order.
[CASHIER 0] Going to rack to get burger...
[CASHIER 0] Got burger from rack, going back


C:\Users\USER\Desktop\explained.exe
[CASHIER 1] Got burger from rack, going back
[COOK 1] Placed new burger in rack.
[CASHIER 0] Gave burger to customer.
[CASHIER 0] Serving customer.
[COOK 0] Placed new burger in rack.
[CUSTOMER 1] Got burger from cashier no. 0. Thank you!
[CUSTOMER 1] DONE.
[CUSTOMER 3] Got burger from cashier no. 1. Thank you!
[CUSTOMER 3] DONE.
[CASHIER 1] Gave burger to customer.
[CASHIER 1] Serving customer.
[CUSTOMER 2] Approached cashier no. 0.
[CUSTOMER 2] Placing order to cashier no. 0.
[CUSTOMER 4] Approached cashier no. 1.
[CUSTOMER 4] Placing order to cashier no. 1.
[CASHIER 1] Got order.
[CASHIER 1] Going to rack to get burger...
[CASHIER 0] Got order.
[CASHIER 0] Going to rack to get burger...
[CASHIER 1] Got burger from rack, going back
[CASHIER 0] Got burger from rack, going back
[CASHIER 1] Gave burger to customer.
[CASHIER 1] Serving customer.
[CUSTOMER 2] Got burger from cashier no. 0. Thank you!
[CUSTOMER 2] DONE.
[CUSTOMER 4] Got burger from cashier no. 1. Thank you!
[CUSTOMER 4] DONE.
[CASHIER 0] Gave burger to customer.
[CASHIER 0] Serving customer.
[CUSTOMER 0] Approached cashier no. 1.


C:\Users\USER\Desktop\explained.exe
[CUSTOMER 0] Placing order to cashier no. 1.
[CUSTOMER 7] Approached cashier no. 0.
[CUSTOMER 7] Placing order to cashier no. 0.
[CASHIER 0] Got order.
[CASHIER 0] Going to rack to get burger...
[CASHIER 1] Got order.
[CASHIER 1] Going to rack to get burger...
[COOK 1] Placed new burger in rack.
[COOK 2] Placed new burger in rack.
[CASHIER 1] Got burger from rack, going back
[CASHIER 0] Got burger from rack, going back
[COOK 1] Placed new burger in rack.
[COOK 0] Placed new burger in rack.
[CASHIER 1] Gave burger to customer.
[CASHIER 1] Serving customer.
[CUSTOMER 0] Got burger from cashier no. 1. Thank you!
[CUSTOMER 0] DONE.
[CASHIER 0] Gave burger to customer.
[CASHIER 0] Serving customer.
[CUSTOMER 7] Got burger from cashier no. 0. Thank you!
[CUSTOMER 7] DONE.
[CUSTOMER 8] Approached cashier no. 0.
[CUSTOMER 8] Placing order to cashier no. 0.
[CUSTOMER 6] Approached cashier no. 1.
[CUSTOMER 6] Placing order to cashier no. 1.
[CASHIER 1] Got order.
[CASHIER 1] Going to rack to get burger...
[CASHIER 0] Got order.
[CASHIER 0] Going to rack to get burger...
[CASHIER 1] Got burger from rack, going back


C:\Users\USER\Desktop\explained.exe
[CASHIER 0] Got burger from rack, going back
[COOK 2] Placed new burger in rack.
[CASHIER 1] Gave burger to customer.
[CASHIER 1] Serving customer.
[CUSTOMER 6] Got burger from cashier no. 1. Thank you!
[CUSTOMER 6] DONE.
[CASHIER 0] Gave burger to customer.
[CASHIER 0] Serving customer.
[CUSTOMER 8] Got burger from cashier no. 0. Thank you!
[CUSTOMER 8] DONE.
[CUSTOMER 9] Approached cashier no. 0.
[CUSTOMER 9] Placing order to cashier no. 0.
[CUSTOMER 5] Approached cashier no. 1.
[CUSTOMER 5] Placing order to cashier no. 1.
[COOK 1] Placed new burger in rack.
[CASHIER 1] Got order.
[CASHIER 1] Going to rack to get burger...
[CASHIER 0] Got order.
[CASHIER 0] Going to rack to get burger...
[CASHIER 1] Got burger from rack, going back
[CASHIER 0] Got burger from rack, going back
[COOK 0] Placed new burger in rack.
[COOK 2] Placed new burger in rack.
[CASHIER 1] Gave burger to customer.
[CASHIER 0] Gave burger to customer.
[CUSTOMER 9] Got burger from cashier no. 0. Thank you!
[CUSTOMER 9] DONE.
[CUSTOMER 5] Got burger from cashier no. 1. Thank you!
[CUSTOMER 5] DONE.
[MAIN] SUCCESS: All threads terminated, state consistent.

Conclusion:

Context switching is the process of storing a process's context or state such that it can be reloaded and execution continued from the same point as before. A "Context Switch" is the act of transitioning from one process to another. A computer system often has multiple duties to complete. So, if one activity requires some I/O, we want to start the I/O operation before moving on to the next process. We'll go through it again later. We should pick up where we left off when we return to a process. For all intents and purposes, this process should never be aware of the switch, and it should appear as if it were the only one in the system.

The End