

**Department of Computer Science & Engineering**

Semester: **Fall-2022**

Course Code: **CSE325**

Course Title: Operating System Section: 01

Group: 02

Project Title(2): Burger Buddies Problem

**Presented by**

Name: Oli Ahmed

ID: 2019-3-60-073

Name: Syed Akib Ahmed

ID: 2019-3-60-022

Name: Tanvir Hasan

ID: 2020-3-60-067

**Presented to Nawab Yousuf Ali** Professor

Department of Computer Science & Engineering East West University

**Date of Submission:** 02-01-2023

# Abstract

In our following project, we will implement and test a solution for the 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 a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is the most important type of system software in a computer system. Users interact with the operating system through a user interface such as a command line or a graphical user interface (GUI). Some examples of operating systems include Microsoft Windows, macOS, and Linux.. 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:

## 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.

## Counting Semaphore –

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

# 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

START

START

Sem\_wait (Cashier\_awake)

Approach a Cashier

Sem\_post (customer)

Wake up and ready to receive order

Make Burger and put it in rack

NO

Placed an Order

Get the order from customer

Rack full?

Check the rack for Burger

Sem\_wait (customer)

YES

Sem\_post ()

Any Burger available?

YES



NO

Takes the Burger and go to customer

Wait until a cook put a burger in rack

Get the Burger

Wait until cashier picks a burger from the rack

Sem\_wait ()

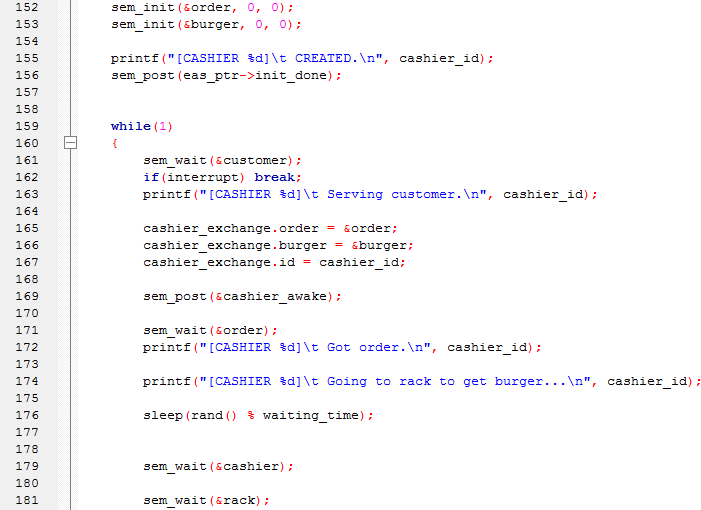
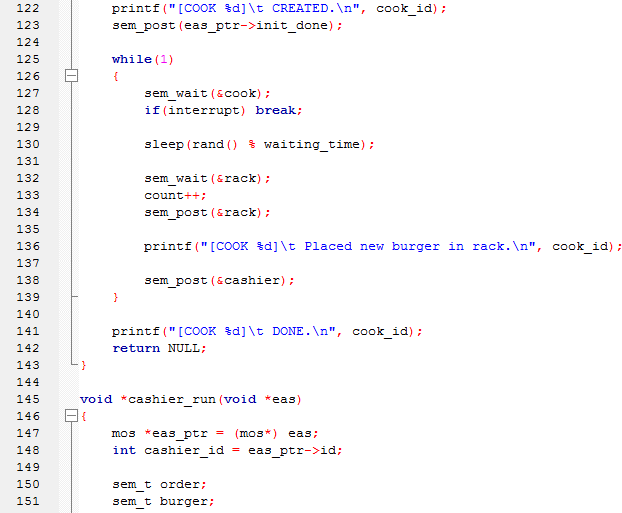
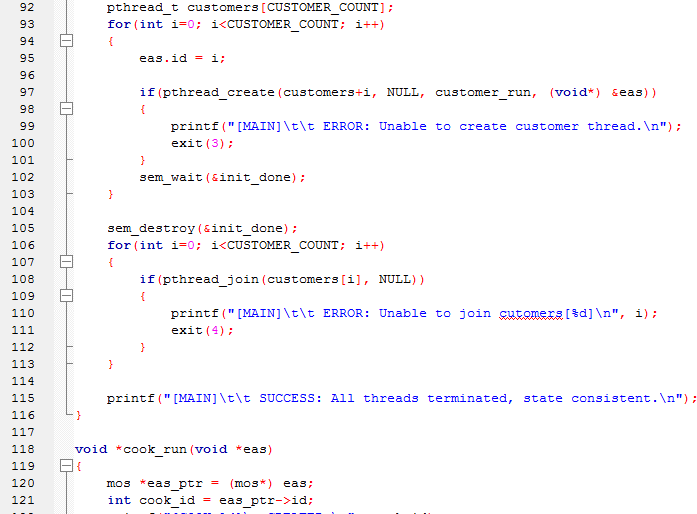
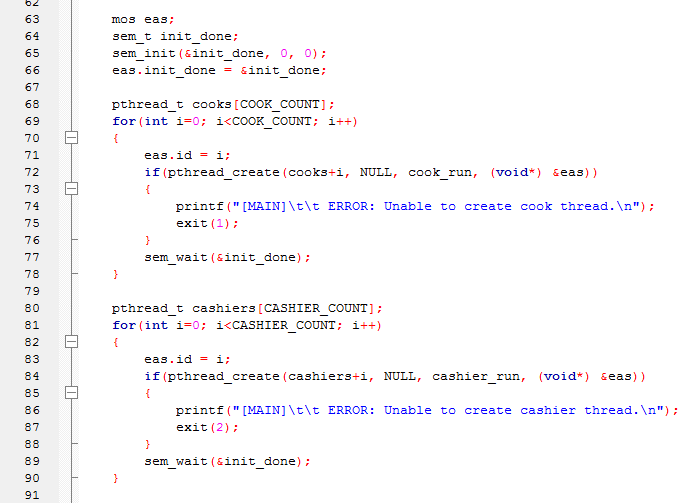
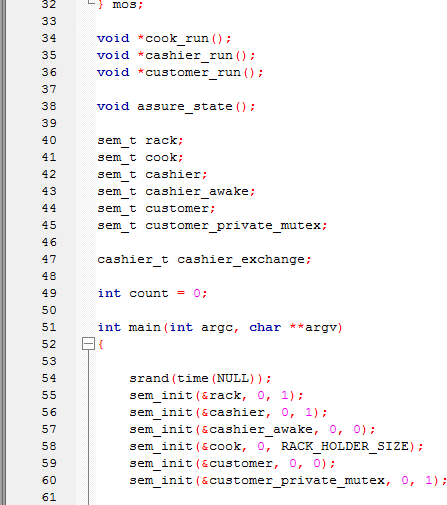
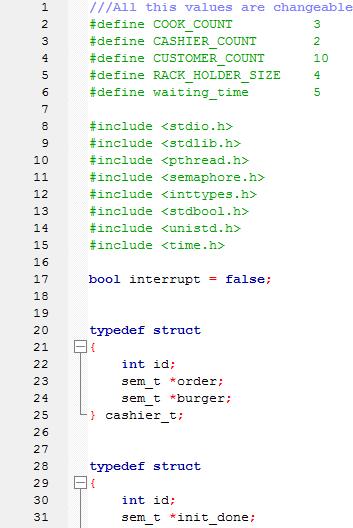
End

End

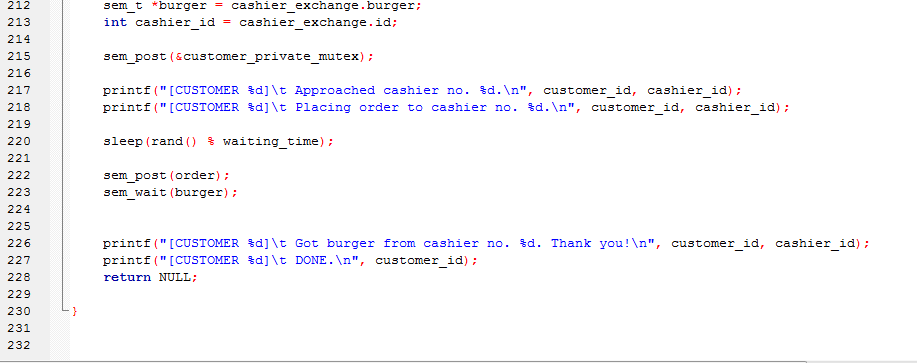
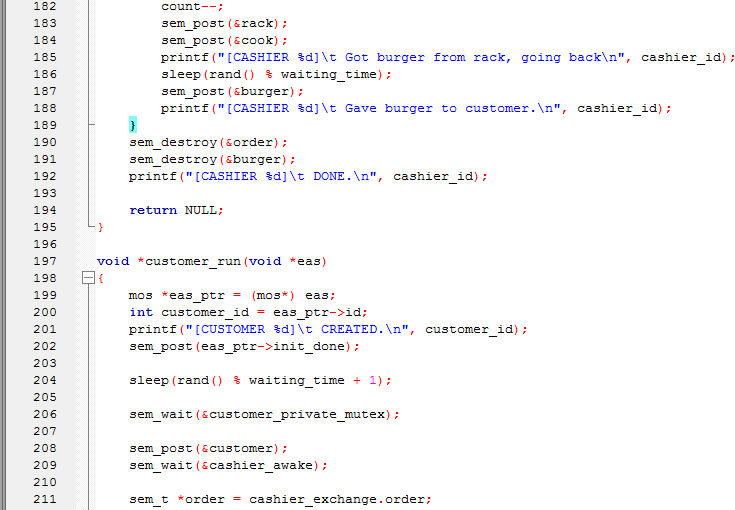
Oder fulfilled

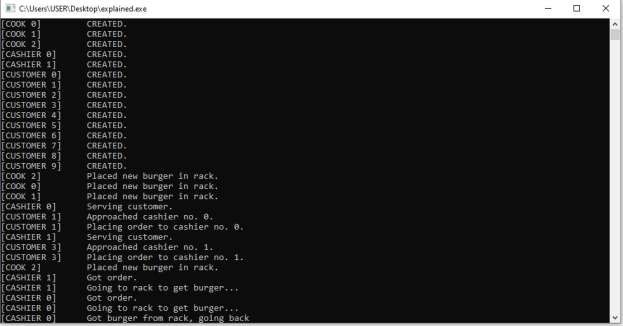
End

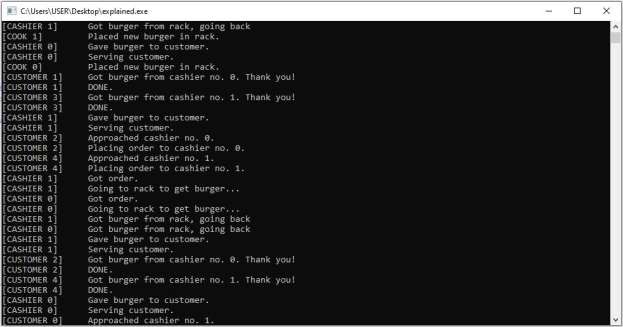
**C Program Code:**

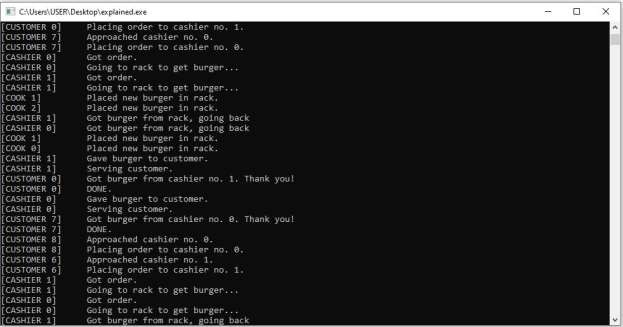


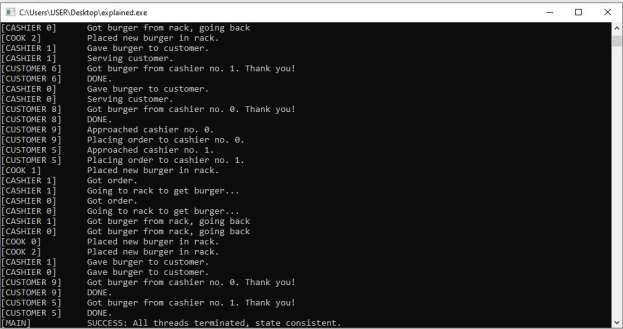
# Output:











**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**