EAST WEST UNIVERSITY

Department of Computer Science and Engineering

Project Report

Project Title: Burger Buddies Problem

Course Title: Operating Systems

Course: CSE325

Section: 1

Submitted by –

Roll No.	Name	ID
22	Rakibul Islam	2021-3-60-030
23	Sagor Ahmed	2021-3-60-117
24	Syeda Tasmiah Chowdhury Orpa	2021-3-60-185
25	Nushrat Zahan	2021-3-60-213

Submitted to –

Dr. Md. Nawab Yousuf Ali
Professor, Department of Computer Science and Engineering
East West University

Submission Date: 26/12/2023

Project Title: Burger Buddies Problem

Project Description:

Design, implement, and test a solution for the IPC problem specified below. 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. Implement a (concurrent multi‑threaded) solution to solve the problem and test it thoroughly.

Introduction:

In the simulated scenario, a restaurant-like setting is organized, with a particular emphasis on the interactions that take place between customers, cashiers, and cooks in a concurrent multithreaded system. The shared resource in this case is a burger rack; the task at hand is to devise and put into action a plan that would ensure smooth cooperation between every participant. The goal is to simulate a realistic setting in which cooks continuously churn out hamburgers and consumers approach cashiers to make orders, all while keeping everything in sync and avoiding possible issues like overloading or deadlocks. The approach uses

synchronization techniques like mutexes, condition variables, and threads to simulate a smooth and effective restaurant system. In this system, customers can place orders only when cashiers are ready, cooks make the burgers, and everything is done without compromising the integrity of the shared resources. This challenge offers an opportunity to explore the complexities of thread synchronization and the creation of a reliable, effective method for imitating a dynamic restaurant environment through the use of inter-process communication (IPC) techniques.

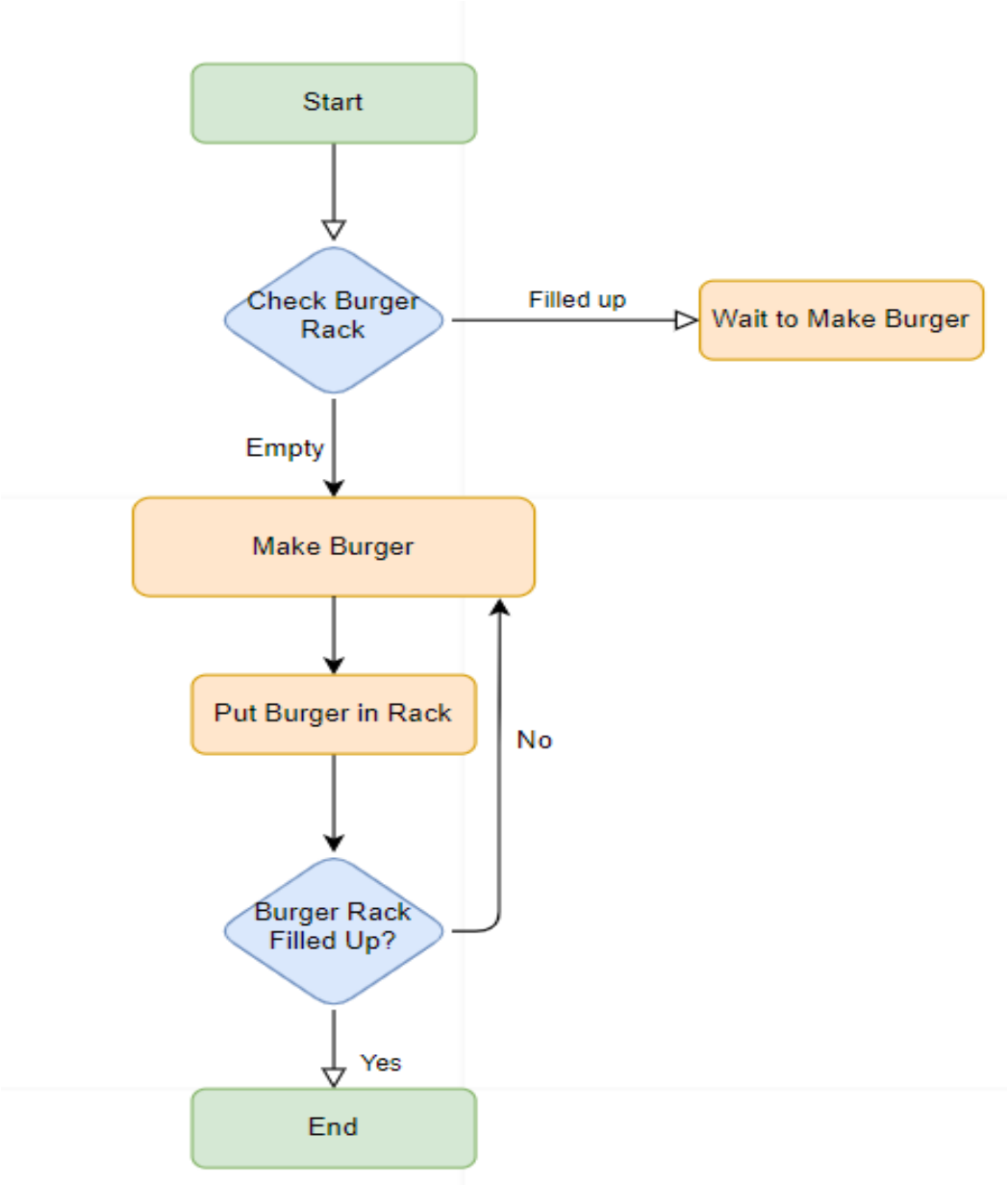
Abstract:

The purpose of this project is to use the Posix Pthread library to solve inter-process communication problems in concurrent programming. Understanding IPC concepts, researching Pthread functions, putting communication mechanisms into practice, resolving mutexes and condition variable synchronization issues, guaranteeing thread safety, managing concurrent execution, incorporating error handling, carrying out comprehensive testing, and generating thorough documentation are all important tasks. Examining Pthread's advanced features might also be taken into account. Reaching these goals will lead us to a good understanding of inter-process communication in POSIX Pthreads concurrent execution.

Flow Chart:

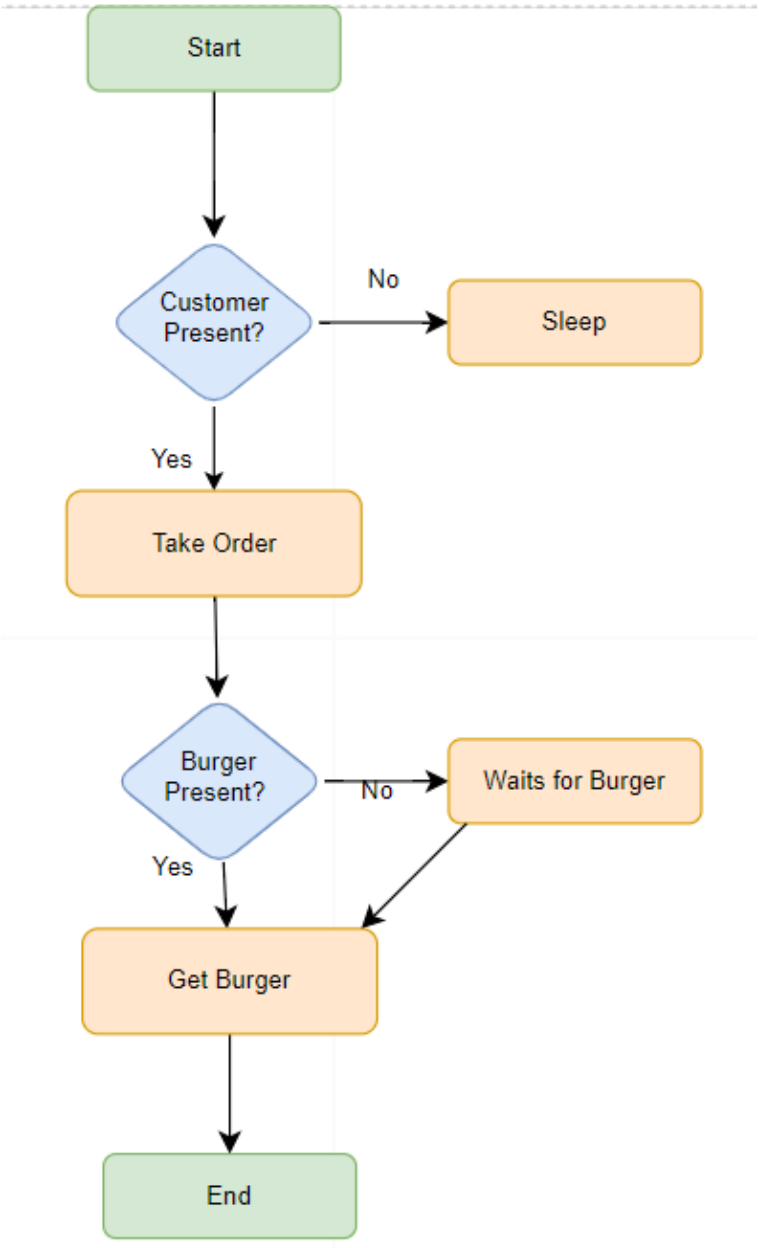
For Cook:

- A Customer cannot order until the cashier is ready



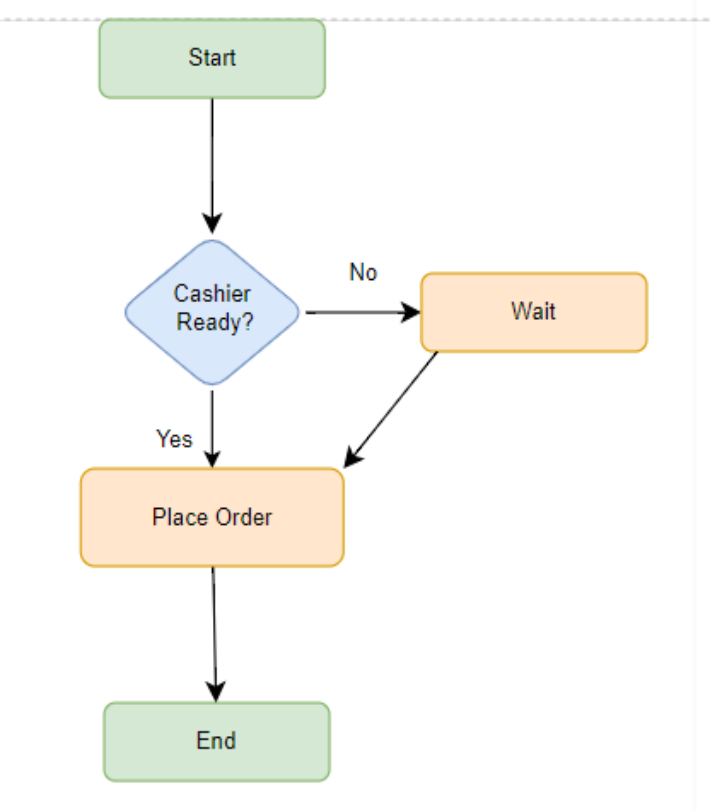
For Cashier:

- 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.
- There are NO synchronization constraints for a cashier presenting food to the customer.



For Customer:

- A Customer cannot order until the cashier is ready



Source Code:

```
// Section: 1
// Group: 2
// Tittle: Burger Buddies Problem

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>
#include <semaphore.h>
#include <inttypes.h>
#include <stdbool.h>
#include <time.h>
```

```
int COOK_COUNT;
int CASHIER_COUNT;
int CUSTOMER_COUNT;
int RACK_CAPACITY;
int WAITING_TIME;

bool interrupt = false;

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

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

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 burger_count = 0;

void check_state() {
   if(burger_count < 0) {
      printf("ERROR --> Negative burger count!\n");
      exit(40);
   }

   if(burger_count > RACK_CAPACITY) {
```

```
        printf("ERROR --> Rack overfull!\n");
        exit(41);
    }

    printf("\t\t|              State is consistent.              |\n");
}

void *cook_func(void *args) {
    simple_arg_t *args_ptr = (simple_arg_t*) args;

    int cook_id = args_ptr->id;

    printf("COOK %d has been created.\n", cook_id + 1);
    sem_post(args_ptr->init_done);

    while(1) {
        sem_wait(&cook);
        if(interrupt) {
            break;
        }
        sleep(rand() % WAITING_TIME);

        sem_wait(&rack);
        check_state();
        burger_count++;
        check_state();
        sem_post(&rack);

        printf("COOK %d has placed a new burger in the rack.\n", cook_id + 1);

        sem_post(&cashier);
    }

    printf("\t\t|            COOK %d is done cooking.            |\n", cook_id + 1);
    return NULL;
}

void *cashier_func(void *args) {
```

```
simple_arg_t *args_ptr = (simple_arg_t*) args;

int cashier_id = args_ptr->id;

sem_t order;
sem_t burger;
sem_init(&order, 0, 0);
sem_init(&burger, 0, 0);

printf("CASHIER %d has been created.\n", cashier_id + 1);
sem_post(args_ptr->init_done);

while(1) {
   sem_wait(&customer);

   if(interrupt) {
      break;
   }
   printf("CASHIER %d is Serving the customer.\n", cashier_id + 1);

   cashier_exchange.order = &order;
   cashier_exchange.burger = &burger;
   cashier_exchange.id = cashier_id;

   sem_post(&cashier_awake);
   sem_wait(&order);
   printf("CASHIER %d has got an order.\n", cashier_id + 1);

   printf("CASHIER %d is going to rack to get burger...\n", cashier_id + 1);

   sleep(rand() % WAITING_TIME);

   sem_wait(&cashier);
   sem_wait(&rack);
   check_state();
   burger_count--;
   check_state();
   sem_post(&rack);
```

```
        sem_post(&cook);

        printf("CASHIER %d got burger from rack, going back\n", cashier_id + 1);

        sleep(rand() % WAITING_TIME);

        sem_post(&burger);
        printf("CASHIER %d gave burger to customer.\n", cashier_id + 1);
    }

    sem_destroy(&order);
    sem_destroy(&burger);
    printf("\t\t|          CASHIER %d is done working.          |\n", cashier_id + 1);

    return NULL;
}

void *customer_func(void *args) {
    simple_arg_t *args_ptr = (simple_arg_t*) args;

    int customer_id = args_ptr->id;

    printf("CUSTOMER %d has been created.\n", customer_id + 1);
    sem_post(args_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 approached to cashier no. %d.\n", customer_id + 1, cashier_id + 1);

    printf("CUSTOMER %d is placing an order to cashier no. %d.\n", customer_id + 1, cashier_id + 1);
    sleep(rand() % WAITING_TIME);

    sem_post(order);

    sem_wait(burger);

    printf("CUSTOMER %d got burger from cashier no. %d. Thank you!\n", customer_id + 1, cashier_id + 1);
    printf("\t\t|   CUSTOMER %d is done with his/her order.   |\n", customer_id + 1);

    return NULL;

}

int main(int argc, char **argv) {
    printf("\t\t\x1B[32m _____\n");
    printf("\t\t|                          |\n");
    printf("\t\t|   Welcome to Burger Buddies  |\n");
    printf("\t\t|_____|\n\n");

    printf("\t\t Today's cook count: ");
    scanf("%d", &COOK_COUNT);
    printf("\t\t Today's cashier count: ");
    scanf("%d", &CASHIER_COUNT);
    printf("\t\t Today's customer count: ");
    scanf("%d", &CUSTOMER_COUNT);
    printf("\t\t Today's rack capacity: ");
    scanf("%d", &RACK_CAPACITY);
    printf("\t\t Today's waiting time: ");
    scanf("%d", &WAITING_TIME);
    printf("\n");

    srand(time(NULL));
    sem_init(&rack, 0, 1);
    sem_init(&cashier, 0, 1);
```

```
sem_init(&cashier_awake, 0, 0);
sem_init(&cook, 0, RACK_CAPACITY);
sem_init(&customer, 0, 0);
sem_init(&customer_private_mutex, 0, 1);

simple_arg_t args;
sem_t init_done;
sem_init(&init_done, 0, 0);
args.init_done = &init_done;

pthread_t cooks[COOK_COUNT];
for(int  i=0; i<COOK_COUNT; i++) {
   args.id = i;

   if(pthread_create(cooks+i, NULL, cook_func, (void*) &args)) {
      printf("ERROR: Unable to create cook thread.\n");
      exit(1);
   }

   sem_wait(&init_done);
}

pthread_t cashiers[CASHIER_COUNT];
for(uint8_t i=0; i<CASHIER_COUNT; i++) {
   args.id = i;

   if(pthread_create(cashiers+i, NULL, cashier_func, (void*) &args)) {
      printf("ERROR: Unable to create cashier thread.\n");
      exit(2);
   }

   sem_wait(&init_done);
}

pthread_t customers[CUSTOMER_COUNT];
for(int  i=0; i<CUSTOMER_COUNT; i++) {
   args.id = i;
```

```
    if(pthread_create(customers+i, NULL, customer_func, (void*) &args)) {
        printf("ERROR: Unable to create customer thread.\n");
        exit(3);
    }

    sem_wait(&init_done);
}

sem_destroy(&init_done);

for(int  i=0; i<CUSTOMER_COUNT; i++) {
    if(pthread_join(customers[i], NULL)) {
        printf("ERROR: Unable to join cutomers %d\n", i + 1);
        exit(4);
    }
}

printf("\n\n\n");
printf("\t\t\x1B[36m _____\n");
printf("\t\t|                                        |\n");
printf("\t\t|   All customers has been served successfully!   |\n");

interrupt = true;

for(int i=0; i<COOK_COUNT; i++) {
    sem_post(&cook);
}

for(int i=0; i<CASHIER_COUNT; i++) {
    sem_post(&customer);
}
printf("\t\t|   All threads has been terminated successfully! |\n");

for(int i=0; i<COOK_COUNT; i++) {
    if(pthread_join(cooks[i], NULL)){
        printf("ERROR: Unable to join cooks %d\n", i + 1);
        exit(5);
    }
```

```
    }

    for(int  i=0; i<CASHIER_COUNT; i++) {
        if(pthread_join(cashiers[i], NULL)){
            printf("ERROR: Unable to join cashiers %d\n", i + 1);
            exit(6);
        }
    }
    check_state();
    printf("\t\t|   All threads has been terminated successfully! |\n");
    printf("\t\t|              in a consistent state.             |\n");
    printf("\t\t|_________________________________________________|\n\n");
}
```

Conclusion:

In conclusion, effectively utilizing Posix Pthreads for inter-process communication is essential for successful concurrent programming. Understanding IPC ideas, utilizing Pthread functions for thread management, putting communication mechanisms in place, and dealing with mutexes and condition variables synchronization difficulties are all necessary to achieve the goals listed. It also requires controlling concurrent execution, adding error handling, testing thoroughly, and ensuring thread safety. Examining more complex Pthread features provides further insights, and thorough documentation is essential for communicating design decisions and promoting teamwork. To sum up, being skilled in POSIX Pthreads inter-process communication improves the creation of reliable and effective concurrent programs.

Thank You!