Operating System - HW4

Angelo Soldi

gsoldi@proton.me

1 Chef's Problem

1.1 Explanation

We are tasked with solving the Chef's problem, where resource management involves files. Threads (`pthread`) are utilized in our concurrent simulation to address the synchronization issue between a *Master Chef* and three chefs. The *Master Chef* places ingredients on a table (represented by the `table.txt` file), while the chefs check if they can use them to prepare ramen. Each chef has a unique ingredient and takes the other two from the table if available. `Mutex` and condition variables are employed to ensure threads safely access the file and coordinate their execution, with a fixed round limit to terminate the simulation.

1.2 Code

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>

#define NUM_CHEFS 3
#define ROUNDS 8   // (limit on the number of rounds to stop the simulation)

// Ingredients
typedef enum { NOODLES, WATER, SEEDS } Ingredient;

// Mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t ingredients_available = PTHREAD_COND_INITIALIZER;
pthread_mutex_t print_mutex = PTHREAD_MUTEX_INITIALIZER;

int remaining_rounds = ROUNDS;

/**
```

```c
 * Safely prints messages from threads.
 */
void safe_printf(const char *format, ...) {
    va_list args;
    va_start(args, format);
    pthread_mutex_lock(&print_mutex);
    vprintf(format, args);
    pthread_mutex_unlock(&print_mutex);
    va_end(args);
}

/**
 * Places ingredients on the table by writing them to the `table.txt` file.
 */
void put_on_table(Ingredient ing1, Ingredient ing2) {
    pthread_mutex_lock(&mutex);

    // Ensure the table is empty before placing new items
    while (access("table.txt", F_OK) != -1) {
        pthread_cond_wait(&ingredients_available, &mutex);
    }

    FILE *file = fopen("table.txt", "w");
    if (file == NULL) {
        perror("Error opening table.txt");
        pthread_mutex_unlock(&mutex);
        exit(1);
    }

    fprintf(file, "%d %d\n", ing1, ing2);
    fclose(file);

    safe_printf("Master Chef: Places %d and %d on the table (file).\n", ing1, ing2);

    pthread_cond_broadcast(&ingredients_available); // Notify chefs
    pthread_mutex_unlock(&mutex);
}

/**
 * Reads the current ingredients on the table from the `table.txt` file.
 */
```

```c
int read_table(Ingredient *ing1, Ingredient *ing2) {
    pthread_mutex_lock(&mutex);

    // Check if the file exists
    if (access("table.txt", F_OK) == -1) {
        pthread_mutex_unlock(&mutex);
        return 0; // No ingredients
    }

    FILE *file = fopen("table.txt", "r");
    if (file == NULL) {
        perror("Error reading table.txt");
        pthread_mutex_unlock(&mutex);
        exit(1);
    }

    fscanf(file, "%d %d", (int *)ing1, (int *)ing2);
    fclose(file);

    pthread_mutex_unlock(&mutex);
    return 1; // Ingredients successfully read
}

/**
 * Deletes the `table.txt` file, indicating the table is empty.
 */
void clear_table() {
    pthread_mutex_lock(&mutex);

    if (remove("table.txt") != 0) {
        perror("Error deleting table.txt");
    }

    pthread_cond_signal(&ingredients_available); // Notify the master chef that the
    ↪  table is empty
    pthread_mutex_unlock(&mutex);
}

/**
 * Main function for the Master Chef. Places random ingredients on the table.
 */
```

```c
void *master_chef(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        if (remaining_rounds <= 0) { // Ensure not to exceed the set number of rounds
            pthread_cond_broadcast(&ingredients_available);
            pthread_mutex_unlock(&mutex);
            break;
        }

        remaining_rounds--;
        pthread_mutex_unlock(&mutex);

        // Generate two random ingredients
        Ingredient ing1 = rand() % NUM_CHEFS;
        Ingredient ing2;
        do {
            ing2 = rand() % NUM_CHEFS;
        } while (ing1 == ing2);

        // Place the ingredients on the table
        put_on_table(ing1, ing2);

        sleep(1); // Simulate master chef's work time
    }
    return NULL;
}

/**
 * Main function for each chef. Takes ingredients from the table and prepares ramen.
 */
void *chef(void *arg) {
    Ingredient my_ingredient = *(Ingredient *)arg;

    while (1) {
        Ingredient ing1, ing2;

        pthread_mutex_lock(&mutex);

        // Wait for available ingredients or end of rounds
        while (access("table.txt", F_OK) == -1 && remaining_rounds > 0) {
```

```
143             pthread_cond_wait(&ingredients_available, &mutex);
144         }
145 
146         if (remaining_rounds <= 0 && access("table.txt", F_OK) == -1) {
147             pthread_mutex_unlock(&mutex);
148             break;
149         }
150 
151         pthread_mutex_unlock(&mutex);
152 
153         // Read ingredients
154         if (!read_table(&ing1, &ing2)) {
155             continue; // Skip if the table was empty
156         }
157 
158         // Check if the ingredients are useful for this chef
159         if (ing1 != my_ingredient && ing2 != my_ingredient) {
160             safe_printf("Chef %d: Takes %d and %d from the table (file).\n",
                ↪  my_ingredient, ing1, ing2);
161 
162             // Ramen preparation (simulation)
163             safe_printf("Chef %d: Prepares ramen.\n", my_ingredient);
164             sleep(1); // Simulation
165             safe_printf("Chef %d: Finishes and notifies the Master Chef.\n",
                ↪  my_ingredient);
166 
167             // Clear the table (delete the file to refill it with new ingredients)
168             clear_table();
169         }
170     }
171 
172     return NULL;
173 }
174 
175 int main() {
176     srand(time(NULL));
177 
178     pthread_t master_thread; // Thread for the Master Chef
179     pthread_t chefs[NUM_CHEFS]; // Threads for the chefs
180     Ingredient ingredients[NUM_CHEFS] = {NOODLES, WATER, SEEDS}; // Ingredients
181 
```

```
    pthread_create(&master_thread, NULL, master_chef, NULL);

    for (int i = 0; i < NUM_CHEFS; i++) {
        pthread_create(&chefs[i], NULL, chef, &ingredients[i]);
    }

    // Wait for the Master Chef to finish
    pthread_join(master_thread, NULL);

    // Wait for the chefs to finish
    for (int i = 0; i < NUM_CHEFS; i++) {
        pthread_join(chefs[i], NULL);
    }

    safe_printf("All chefs have finished.\n");
    return 0;
}
```

1.3 Console Output (Valgrind Tests)

1.3.1 Summary of Memory Leaks (valgrind –leak-check=summary)

```
Master Chef: Coloca 0 y 2 en la mesa (archivo).
Chef 1: Toma 0 y 2 de la mesa (archivo).
Chef 1: Prepara el ramen.
Chef 1: Termina y avisa al Master Chef.
Todos los chefs terminaron.
==47373==
==47373== HEAP SUMMARY:
==47373==     in use at exit: 0 bytes in 0 blocks
==47373==   total heap usage: 866,707 allocs, 866,707 frees, 1,979,549,480 bytes
==47373==
==47373== All heap blocks were freed -- no leaks are possible
==47373==
==47373== For lists of detected and suppressed errors, rerun with: -s
==47373== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
darkblackv32@pop-os:~/Documents/CS/CS3015/e4$
```

Figure 1: Leak Summary.

1.3.2 Memory and Leak Error Detection (valgrind –track-origins=yes –leak-check=full)

```
==49045== Memcheck, a memory error detector
==49045== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==49045== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==49045== Command: ./ramen
==49045==
Master Chef: Coloca 1 y 0 en la mesa (archivo).
Chef 2: Toma 1 y 0 de la mesa (archivo).
Chef 2: Prepara el ramen.
Chef 2: Termina y avisa al Master Chef.
Master Chef: Coloca 1 y 0 en la mesa (archivo).
Chef 2: Toma 1 y 0 de la mesa (archivo).
Chef 2: Prepara el ramen.
Chef 2: Termina y avisa al Master Chef.
Todos los chefs terminaron.
==49045==
==49045== HEAP SUMMARY:
==49045==     in use at exit: 0 bytes in 0 blocks
==49045==   total heap usage: 112,093 allocs, 112,093 frees, 256,011,104 bytes allocated
==49045==
==49045== All heap blocks were freed -- no leaks are possible
==49045==
==49045== For lists of detected and suppressed errors, rerun with: -s
==49045== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 2: No memory leak.

1.3.3 Memory Check (valgrind –leak-check=full –show-leak-kinds=all)

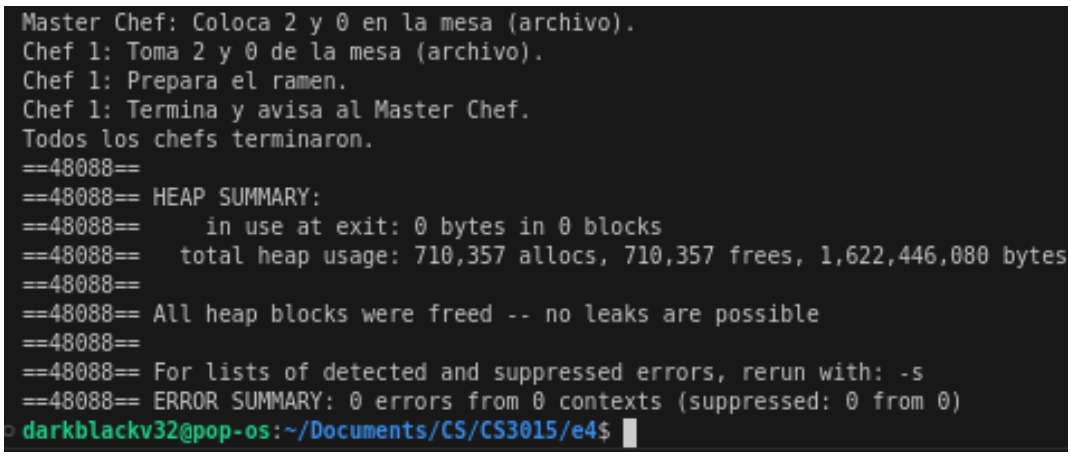

Figure 3: No memory errors.

1.3.4 Thread Concurrency Problems (valgrind –tool=helgrind)

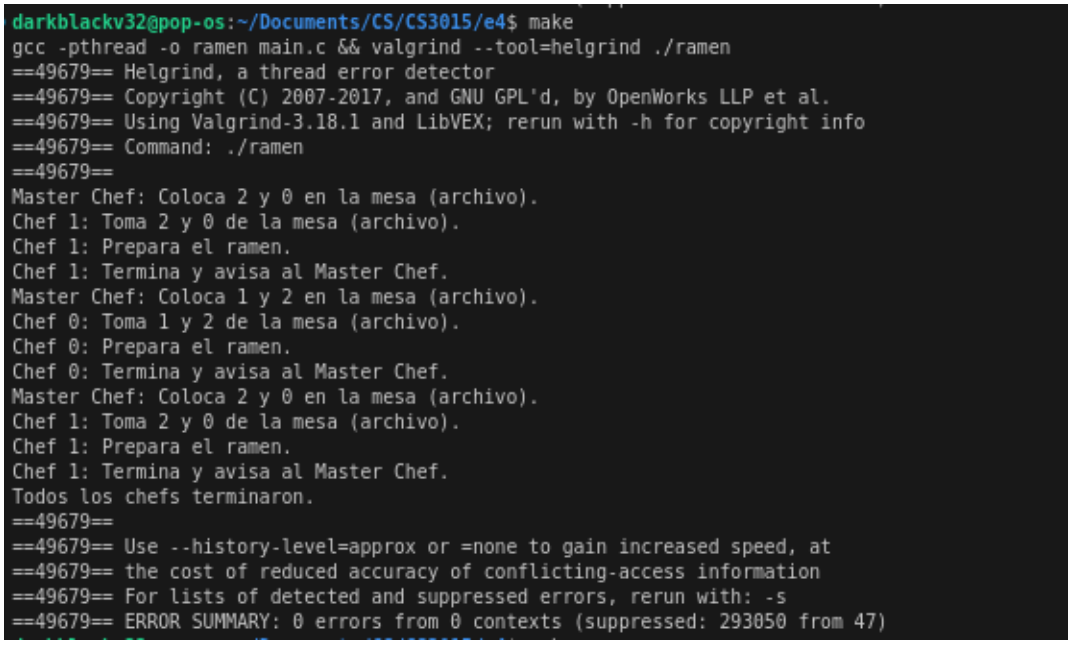

Figure 4: No thread concurrency errors.

2 Cinema

2.1 Explanation

This program models a *multiplex* cinema using threads and monitors in C. The cinema rooms have limited capacities (4, 5, and 7 people), and customers attempt to enter a randomly selected room.

A mutex is used to protect concurrent access, and condition variables handle customer waiting when rooms are full or during a projection. Each room runs its projection in an independent thread, blocking new entries while the movie is playing. At the end of the projection, waiting customers are notified.

2.2 Code

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <string.h>

// Monitor structure for synchronization
typedef struct {
    pthread_mutex_t lock;           // Mutex for access control
    pthread_cond_t room_available; // Condition variable for available rooms
    int capacity[3];               // Capacity of each room (4, 5, and 7 people)
} CinemaMonitor;

// Global monitor for the cinema
CinemaMonitor cinema = {
    .lock = PTHREAD_MUTEX_INITIALIZER,
    .room_available = PTHREAD_COND_INITIALIZER,
    .capacity = {4, 5, 7} // Room capacities
};

int active_projections = 3; // Number of active projections

/**
 * Safely prints messages from threads.
 */
void safe_printf(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
}

/**
```

```c
 * Initializes room files with their initial data.
 */
void initialize_rooms() {
    FILE *file;
    for (int i = 0; i < 3; i++) {
        char filename[20];
        sprintf(filename, "room%d.txt", i + 1);
        file = fopen(filename, "w");
        if (file == NULL) {
            perror("Error initializing room file");
            exit(1);
        }
        // Write initial occupancy (0), capacity, and projection state (0)
        fprintf(file, "0 %d 0\n", cinema.capacity[i]);
        fclose(file);
        printf("File %s successfully created.\n", filename);
    }
}

/**
 * Simulates a customer entering a room.
 */
void *enter_room(void *arg) {
    int customer_id = *(int *)arg; // Customer ID
    int room_id = rand() % 3 + 1;  // Random room selection
    free(arg);

    pthread_mutex_lock(&cinema.lock); // Lock monitor mutex

    char filename[20];
    sprintf(filename, "room%d.txt", room_id);

    while (1) {
        if (active_projections == 0) {
            // No active projections, customer cannot enter
            pthread_mutex_unlock(&cinema.lock);
            return NULL;
        }

        FILE *file = fopen(filename, "r+");
        if (file == NULL) {
```

```
76              perror("Error opening room file");
77              pthread_mutex_unlock(&cinema.lock);
78              return NULL;
79          }
80
81          // Read current room data (check for space and projection status)
82          int occupancy, capacity, in_projection;
83          fscanf(file, "%d %d %d", &occupancy, &capacity, &in_projection);
84
85          if (occupancy < capacity && in_projection == 0) {
86              // If there is space and no active projection, the customer enters
87              occupancy++;
88              rewind(file); // Move pointer to the start of the file
89              fprintf(file, "%d %d %d\n", occupancy, capacity, in_projection);
90              fclose(file);
91              safe_printf("Customer %d entered room %d. Occupancy: %d/%d.\n",
                ↪  customer_id, room_id, occupancy, capacity);
92              break;
93          }
94
95          fclose(file);
96          // Wait for room availability
97          pthread_cond_wait(&cinema.room_available, &cinema.lock);
98      }
99
100     pthread_mutex_unlock(&cinema.lock); // Unlock monitor mutex
101     return NULL;
102 }
103
104 /**
105  * Simulates starting a projection in a room.
106  */
107 void *start_projection(void *arg) {
108     int room_id = *(int *)arg; // Room ID
109     free(arg);
110
111     pthread_mutex_lock(&cinema.lock); // Lock monitor mutex
112
113     char filename[20];
114     sprintf(filename, "room%d.txt", room_id);
115
```

```c
    FILE *file = fopen(filename, "r+");
    if (file == NULL) {
        perror("Error opening room file");
        pthread_mutex_unlock(&cinema.lock);
        return NULL;
    }

    // Read current data
    int occupancy, capacity, in_projection;
    fscanf(file, "%d %d %d", &occupancy, &capacity, &in_projection);

    // Update the room state to "in projection"
    in_projection = 1;
    rewind(file); // Move pointer to the start of the file
    fprintf(file, "%d %d %d\n", occupancy, capacity, in_projection);
    fclose(file);

    safe_printf("Room %d started the projection. Occupancy: %d/%d.\n", room_id,
    ↪  occupancy, capacity);

    pthread_mutex_unlock(&cinema.lock); // Unlock mutex to allow other threads to
    ↪  access

    sleep(2); // Simulate projection time

    pthread_mutex_lock(&cinema.lock); // Lock mutex to finish projection

    file = fopen(filename, "r+");
    if (file == NULL) {
        perror("Error opening room file");
        pthread_mutex_unlock(&cinema.lock);
        return NULL;
    }

    // Read room data
    fscanf(file, "%d %d %d", &occupancy, &capacity, &in_projection);
    in_projection = 0; // Mark projection as finished
    rewind(file);
    fprintf(file, "%d %d %d\n", occupancy, capacity, in_projection);
    fclose(file);

```

```c
    safe_printf("Room %d finished the projection. Occupancy: %d/%d.\n", room_id,
    ↪  occupancy, capacity);

    active_projections--; // Reduce active projections
    pthread_cond_broadcast(&cinema.room_available); // Notify all waiting threads

    pthread_mutex_unlock(&cinema.lock); // Unlock mutex
    return NULL;
}

int main() {
    initialize_rooms(); // Initialize room files

    pthread_t customers[100]; // Threads for 100 customers
    pthread_t projections[3]; // Threads for 3 rooms

    // Create customer threads
    for (int i = 0; i < 100; i++) {
        int *customer_id = malloc(sizeof(int));
        *customer_id = i + 1;
        pthread_create(&customers[i], NULL, enter_room, customer_id);
    }

    // Create projection threads
    for (int i = 0; i < 3; i++) {
        int *room_id = malloc(sizeof(int));
        *room_id = i + 1;
        pthread_create(&projections[i], NULL, start_projection, room_id);
    }

    for (int i = 0; i < 100; i++) {
        pthread_join(customers[i], NULL);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(projections[i], NULL);
    }

    safe_printf("Simulation completed.\n");
    return 0;
}
```

195

2.3 Console Output (Valgrind Tests)

2.3.1 Summary of Memory Leaks (valgrind –leak-check=summary)

Figure 5: Leak Summary.

2.3.2 Memory and Leak Error Detection (valgrind –track-origins=yes –leak-check=full)

Figure 6: No memory leaks.

2.3.3 Memory Check (valgrind –leak-check=full –show-leak-kinds=all)

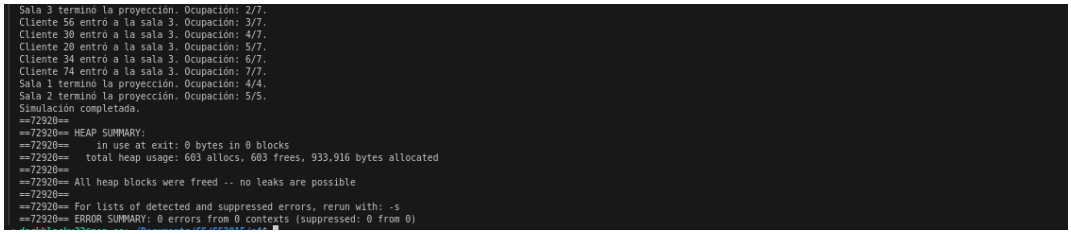

Figure 7: No memory errors.

2.3.4 Thread Concurrency Issues (valgrind –tool=helgrind)

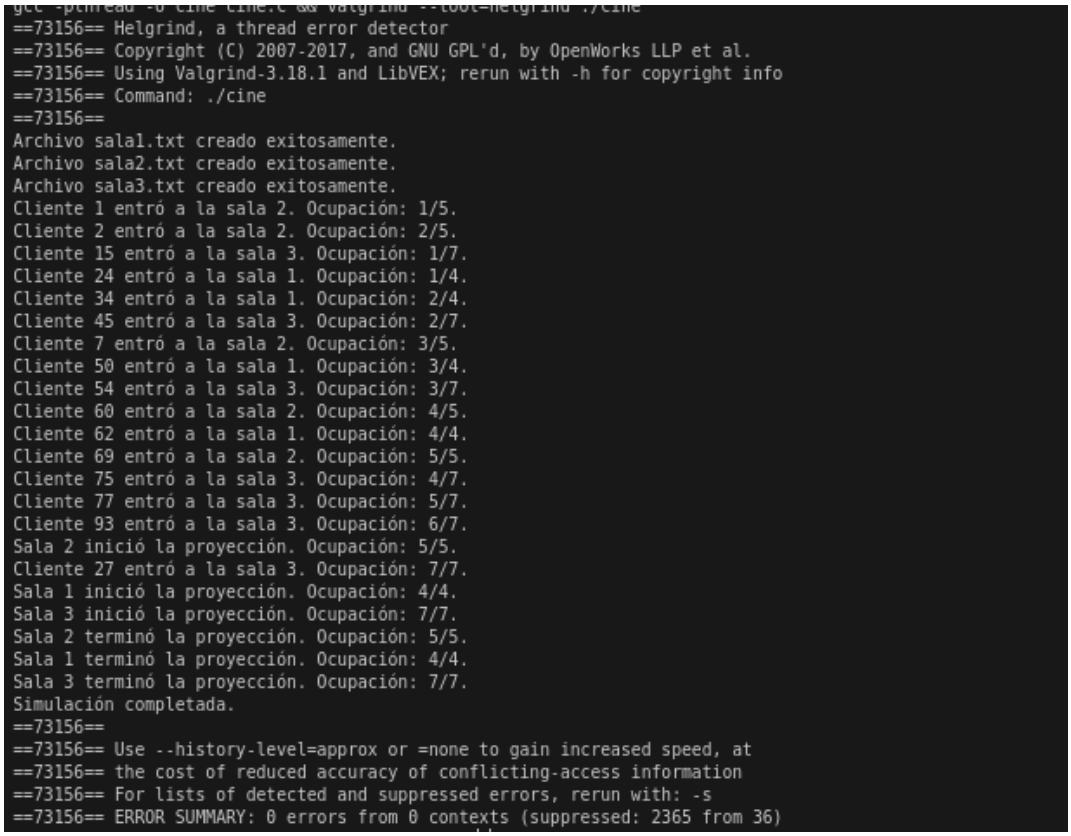

Figure 8: No thread concurrency errors.