

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

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <stdarg.h>
6
7  #define NUM_CHEFS 3
8  #define ROUNDS 8    // (limit on the number of rounds to stop the simulation)
9
10 // Ingredients
11 typedef enum { NOODLES, WATER, SEEDS } Ingredient;
12
13 // Mutex
14 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
15 pthread_cond_t ingredients_available = PTHREAD_COND_INITIALIZER;
16 pthread_mutex_t print_mutex = PTHREAD_MUTEX_INITIALIZER;
17
18 int remaining_rounds = ROUNDS;
19
20 /**
```

```

21  * Safely prints messages from threads.
22  */
23  void safe_printf(const char *format, ...) {
24      va_list args;
25      va_start(args, format);
26      pthread_mutex_lock(&print_mutex);
27      vprintf(format, args);
28      pthread_mutex_unlock(&print_mutex);
29      va_end(args);
30  }
31
32  /**
33   * Places ingredients on the table by writing them to the `table.txt` file.
34   */
35  void put_on_table(Ingredient ing1, Ingredient ing2) {
36      pthread_mutex_lock(&mutex);
37
38      // Ensure the table is empty before placing new items
39      while (access("table.txt", F_OK) != -1) {
40          pthread_cond_wait(&ingredients_available, &mutex);
41      }
42
43      FILE *file = fopen("table.txt", "w");
44      if (file == NULL) {
45          perror("Error opening table.txt");
46          pthread_mutex_unlock(&mutex);
47          exit(1);
48      }
49
50      fprintf(file, "%d %d\n", ing1, ing2);
51      fclose(file);
52
53      safe_printf("Master Chef: Places %d and %d on the table (file).\n", ing1, ing2);
54
55      pthread_cond_broadcast(&ingredients_available); // Notify chefs
56      pthread_mutex_unlock(&mutex);
57  }
58
59  /**
60   * Reads the current ingredients on the table from the `table.txt` file.
61   */

```

```

62 int read_table(Ingredient *ing1, Ingredient *ing2) {
63     pthread_mutex_lock(&mutex);
64
65     // Check if the file exists
66     if (access("table.txt", F_OK) == -1) {
67         pthread_mutex_unlock(&mutex);
68         return 0; // No ingredients
69     }
70
71     FILE *file = fopen("table.txt", "r");
72     if (file == NULL) {
73         perror("Error reading table.txt");
74         pthread_mutex_unlock(&mutex);
75         exit(1);
76     }
77
78     fscanf(file, "%d %d", (int *)ing1, (int *)ing2);
79     fclose(file);
80
81     pthread_mutex_unlock(&mutex);
82     return 1; // Ingredients successfully read
83 }
84
85 /**
86  * Deletes the `table.txt` file, indicating the table is empty.
87  */
88 void clear_table() {
89     pthread_mutex_lock(&mutex);
90
91     if (remove("table.txt") != 0) {
92         perror("Error deleting table.txt");
93     }
94
95     pthread_cond_signal(&ingredients_available); // Notify the master chef that the
96     ↪ table is empty
97     pthread_mutex_unlock(&mutex);
98 }
99
100 /**
101  * Main function for the Master Chef. Places random ingredients on the table.
102  */

```

```

102 void *master_chef(void *arg) {
103     while (1) {
104         pthread_mutex_lock(&mutex);
105
106         if (remaining_rounds <= 0) { // Ensure not to exceed the set number of rounds
107             pthread_cond_broadcast(&ingredients_available);
108             pthread_mutex_unlock(&mutex);
109             break;
110         }
111
112         remaining_rounds--;
113         pthread_mutex_unlock(&mutex);
114
115         // Generate two random ingredients
116         Ingredient ing1 = rand() % NUM_CHEFS;
117         Ingredient ing2;
118         do {
119             ing2 = rand() % NUM_CHEFS;
120         } while (ing1 == ing2);
121
122         // Place the ingredients on the table
123         put_on_table(ing1, ing2);
124
125         sleep(1); // Simulate master chef's work time
126     }
127     return NULL;
128 }
129
130 /**
131  * Main function for each chef. Takes ingredients from the table and prepares ramen.
132  */
133 void *chef(void *arg) {
134     Ingredient my_ingredient = *(Ingredient *)arg;
135
136     while (1) {
137         Ingredient ing1, ing2;
138
139         pthread_mutex_lock(&mutex);
140
141         // Wait for available ingredients or end of rounds
142         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",
161         ↪ my_ingredient, ing1, ing2);
162
163     // Ramen preparation (simulation)
164     safe_printf("Chef %d: Prepares ramen.\n", my_ingredient);
165     sleep(1); // Simulation
166     safe_printf("Chef %d: Finishes and notifies the Master Chef.\n",
167         ↪ my_ingredient);
168
169     // Clear the table (delete the file to refill it with new ingredients)
170     clear_table();
171 }
172 }
173
174 return NULL;
175 }
176
177 int main() {
178     srand(time(NULL));
179
180     pthread_t master_thread; // Thread for the Master Chef
181     pthread_t chefs[NUM_CHEFS]; // Threads for the chefs
182     Ingredient ingredients[NUM_CHEFS] = {NOODLES, WATER, SEEDS}; // Ingredients

```

```
182 pthread_create(&master_thread, NULL, master_chef, NULL);
183
184 for (int i = 0; i < NUM_CHEFS; i++) {
185     pthread_create(&chefs[i], NULL, chef, &ingredients[i]);
186 }
187
188 // Wait for the Master Chef to finish
189 pthread_join(master_thread, NULL);
190
191 // Wait for the chefs to finish
192 for (int i = 0; i < NUM_CHEFS; i++) {
193     pthread_join(chefs[i], NULL);
194 }
195
196 safe_printf("All chefs have finished.\n");
197 return 0;
198 }
```

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)

```
Master Chef: Coloca 2 y 0 en la mesa (archivo).
Chef 1: Toma 2 y 0 de la mesa (archivo).
Chef 1: Prepara el ramen.
Chef 1: Termina y avisa al Master Chef.
Todos los chefs terminaron.
==48088==
==48088== HEAP SUMMARY:
==48088==    in use at exit: 0 bytes in 0 blocks
==48088==   total heap usage: 710,357 allocs, 710,357 frees, 1,622,446,080 bytes
==48088==
==48088== All heap blocks were freed -- no leaks are possible
==48088==
==48088== For lists of detected and suppressed errors, rerun with: -s
==48088== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
darkblackv32@pop-os:~/Documents/CS/CS3015/e4$
```

Figure 3: No memory errors.

1.3.4 Thread Concurrency Problems (valgrind --tool=helgrind)

```
darkblackv32@pop-os:~/Documents/CS/CS3015/e4$ make
gcc -pthread -o ramen main.c && valgrind --tool=helgrind ./ramen
==49679== Helgrind, a thread error detector
==49679== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==49679== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==49679== Command: ./ramen
==49679==
Master Chef: Coloca 2 y 0 en la mesa (archivo).
Chef 1: Toma 2 y 0 de la mesa (archivo).
Chef 1: Prepara el ramen.
Chef 1: Termina y avisa al Master Chef.
Master Chef: Coloca 1 y 2 en la mesa (archivo).
Chef 0: Toma 1 y 2 de la mesa (archivo).
Chef 0: Prepara el ramen.
Chef 0: Termina y avisa al Master Chef.
Master Chef: Coloca 2 y 0 en la mesa (archivo).
Chef 1: Toma 2 y 0 de la mesa (archivo).
Chef 1: Prepara el ramen.
Chef 1: Termina y avisa al Master Chef.
Todos los chefs terminaron.
==49679==
==49679== Use --history-level=approx or =none to gain increased speed, at
==49679== the cost of reduced accuracy of conflicting-access information
==49679== For lists of detected and suppressed errors, rerun with: -s
==49679== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 293050 from 47)
```

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

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <stdarg.h>
6  #include <string.h>
7
8  // Monitor structure for synchronization
9  typedef struct {
10     pthread_mutex_t lock;           // Mutex for access control
11     pthread_cond_t room_available; // Condition variable for available rooms
12     int capacity[3];               // Capacity of each room (4, 5, and 7 people)
13 } CinemaMonitor;
14
15 // Global monitor for the cinema
16 CinemaMonitor cinema = {
17     .lock = PTHREAD_MUTEX_INITIALIZER,
18     .room_available = PTHREAD_COND_INITIALIZER,
19     .capacity = {4, 5, 7} // Room capacities
20 };
21
22 int active_projections = 3; // Number of active projections
23
24 /**
25  * Safely prints messages from threads.
26  */
27 void safe_printf(const char *format, ...) {
28     va_list args;
29     va_start(args, format);
30     vprintf(format, args);
31     va_end(args);
32 }
33
34 /**
```

```

35  * Initializes room files with their initial data.
36  */
37  void initialize_rooms() {
38      FILE *file;
39      for (int i = 0; i < 3; i++) {
40          char filename[20];
41          sprintf(filename, "room%d.txt", i + 1);
42          file = fopen(filename, "w");
43          if (file == NULL) {
44              perror("Error initializing room file");
45              exit(1);
46          }
47          // Write initial occupancy (0), capacity, and projection state (0)
48          fprintf(file, "0 %d 0\n", cinema.capacity[i]);
49          fclose(file);
50          printf("File %s successfully created.\n", filename);
51      }
52  }
53
54  /**
55   * Simulates a customer entering a room.
56   */
57  void *enter_room(void *arg) {
58      int customer_id = *(int *)arg; // Customer ID
59      int room_id = rand() % 3 + 1; // Random room selection
60      free(arg);
61
62      pthread_mutex_lock(&cinema.lock); // Lock monitor mutex
63
64      char filename[20];
65      sprintf(filename, "room%d.txt", room_id);
66
67      while (1) {
68          if (active_projections == 0) {
69              // No active projections, customer cannot enter
70              pthread_mutex_unlock(&cinema.lock);
71              return NULL;
72          }
73
74          FILE *file = fopen(filename, "r+");
75          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",
92             ↪ customer_id, room_id, occupancy, capacity);
93         break;
94     }
95
96     fclose(file);
97     // Wait for room availability
98     pthread_cond_wait(&cinema.room_available, &cinema.lock);
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

```

```

116 FILE *file = fopen(filename, "r+");
117 if (file == NULL) {
118     perror("Error opening room file");
119     pthread_mutex_unlock(&cinema.lock);
120     return NULL;
121 }
122
123 // Read current data
124 int occupancy, capacity, in_projection;
125 fscanf(file, "%d %d %d", &occupancy, &capacity, &in_projection);
126
127 // Update the room state to "in projection"
128 in_projection = 1;
129 rewind(file); // Move pointer to the start of the file
130 fprintf(file, "%d %d %d\n", occupancy, capacity, in_projection);
131 fclose(file);
132
133 safe_printf("Room %d started the projection. Occupancy: %d/%d.\n", room_id,
134 ↪ occupancy, capacity);
135
136 pthread_mutex_unlock(&cinema.lock); // Unlock mutex to allow other threads to
137 ↪ access
138
139 sleep(2); // Simulate projection time
140
141 pthread_mutex_lock(&cinema.lock); // Lock mutex to finish projection
142
143 file = fopen(filename, "r+");
144 if (file == NULL) {
145     perror("Error opening room file");
146     pthread_mutex_unlock(&cinema.lock);
147     return NULL;
148 }
149
150 // Read room data
151 fscanf(file, "%d %d %d", &occupancy, &capacity, &in_projection);
152 in_projection = 0; // Mark projection as finished
153 rewind(file);
154 fprintf(file, "%d %d %d\n", occupancy, capacity, in_projection);
155 fclose(file);

```

```

155     safe_printf("Room %d finished the projection. Occupancy: %d/%d.\n", room_id,
156         ↪ occupancy, capacity);
157
158     active_projections--; // Reduce active projections
159     pthread_cond_broadcast(&cinema.room_available); // Notify all waiting threads
160
161     pthread_mutex_unlock(&cinema.lock); // Unlock mutex
162     return NULL;
163 }
164
165 int main() {
166     initialize_rooms(); // Initialize room files
167
168     pthread_t customers[100]; // Threads for 100 customers
169     pthread_t projections[3]; // Threads for 3 rooms
170
171     // Create customer threads
172     for (int i = 0; i < 100; i++) {
173         int *customer_id = malloc(sizeof(int));
174         *customer_id = i + 1;
175         pthread_create(&customers[i], NULL, enter_room, customer_id);
176     }
177
178     // Create projection threads
179     for (int i = 0; i < 3; i++) {
180         int *room_id = malloc(sizeof(int));
181         *room_id = i + 1;
182         pthread_create(&projections[i], NULL, start_projection, room_id);
183     }
184
185     for (int i = 0; i < 100; i++) {
186         pthread_join(customers[i], NULL);
187     }
188
189     for (int i = 0; i < 3; i++) {
190         pthread_join(projections[i], NULL);
191     }
192
193     safe_printf("Simulation completed.\n");
194     return 0;
195 }

```



2.3 Console Output (Valgrind Tests)

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

```
Sala 1 terminó la proyección. Ocupación: 4/4.
Sala 2 terminó la proyección. Ocupación: 5/5.
Sala 3 terminó la proyección. Ocupación: 7/7.
Simulación completada.
==72377==
==72377== HEAP SUMMARY:
==72377==    in use at exit: 0 bytes in 0 blocks
==72377== total heap usage: 593 allocs, 593 frees, 911,076 bytes allocated
==72377==
==72377== All heap blocks were freed -- no leaks are possible
==72377==
==72377== For lists of detected and suppressed errors, rerun with: -s
==72377== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
darkblackv32@pop-os:~/Documents/CS/CS3015/e4$
```

Figure 5: Leak Summary.

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

```
Cliente 24 entró a la sala 3. Ocupación: 5/7.
Cliente 26 entró a la sala 1. Ocupación: 4/4.
Cliente 28 entró a la sala 3. Ocupación: 6/7.
Cliente 8 entró a la sala 3. Ocupación: 7/7.
Sala 1 inició la proyección. Ocupación: 4/4.
Sala 2 inició la proyección. Ocupación: 5/5.
Sala 3 inició la proyección. Ocupación: 7/7.
Sala 1 terminó la proyección. Ocupación: 4/4.
Sala 2 terminó la proyección. Ocupación: 5/5.
Sala 3 terminó la proyección. Ocupación: 7/7.
Simulación completada.
==72636==
==72636== HEAP SUMMARY:
==72636==    in use at exit: 0 bytes in 0 blocks
==72636== total heap usage: 591 allocs, 591 frees, 906,508 bytes allocated
==72636==
==72636== All heap blocks were freed -- no leaks are possible
==72636==
==72636== For lists of detected and suppressed errors, rerun with: -s
==72636== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
darkblackv32@pop-os:~/Documents/CS/CS3015/e4$
```

Figure 6: No memory leaks.

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

```
Sala 3 terminó la proyección. Ocupación: 7/7.
Cliente 56 entró a la sala 3. Ocupación: 3/7.
Cliente 30 entró a la sala 3. Ocupación: 4/7.
Cliente 20 entró a la sala 3. Ocupación: 5/7.
Cliente 34 entró a la sala 3. Ocupación: 6/7.
Cliente 74 entró a la sala 3. Ocupación: 7/7.
Sala 1 terminó la proyección. Ocupación: 4/4.
Sala 2 terminó la proyección. Ocupación: 5/5.
Simulación completada.
==72920==
==72920== HEAP SUMMARY:
==72920==    in use at exit: 0 bytes in 0 blocks
==72920== total heap usage: 603 allocs, 603 frees, 933,916 bytes allocated
==72920==
==72920== All heap blocks were freed -- no leaks are possible
==72920==
==72920== For lists of detected and suppressed errors, rerun with: -s
==72920== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 7: No memory errors.

2.3.4 Thread Concurrency Issues (valgrind --tool=helgrind)

```
gcc -pthread -o cine cine.c -g -DVALGRIND --tool=helgrind ./cine
==73156== Helgrind, a thread error detector
==73156== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==73156== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==73156== Command: ./cine
==73156==
Archivo sala1.txt creado exitosamente.
Archivo sala2.txt creado exitosamente.
Archivo sala3.txt creado exitosamente.
Cliente 1 entró a la sala 2. Ocupación: 1/5.
Cliente 2 entró a la sala 2. Ocupación: 2/5.
Cliente 15 entró a la sala 3. Ocupación: 1/7.
Cliente 24 entró a la sala 1. Ocupación: 1/4.
Cliente 34 entró a la sala 1. Ocupación: 2/4.
Cliente 45 entró a la sala 3. Ocupación: 2/7.
Cliente 7 entró a la sala 2. Ocupación: 3/5.
Cliente 50 entró a la sala 1. Ocupación: 3/4.
Cliente 54 entró a la sala 3. Ocupación: 3/7.
Cliente 60 entró a la sala 2. Ocupación: 4/5.
Cliente 62 entró a la sala 1. Ocupación: 4/4.
Cliente 69 entró a la sala 2. Ocupación: 5/5.
Cliente 75 entró a la sala 3. Ocupación: 4/7.
Cliente 77 entró a la sala 3. Ocupación: 5/7.
Cliente 93 entró a la sala 3. Ocupación: 6/7.
Sala 2 inició la proyección. Ocupación: 5/5.
Cliente 27 entró a la sala 3. Ocupación: 7/7.
Sala 1 inició la proyección. Ocupación: 4/4.
Sala 3 inició la proyección. Ocupación: 7/7.
Sala 2 terminó la proyección. Ocupación: 5/5.
Sala 1 terminó la proyección. Ocupación: 4/4.
Sala 3 terminó la proyección. Ocupación: 7/7.
Simulación completada.
==73156==
==73156== Use --history-level=approx or =none to gain increased speed, at
==73156== the cost of reduced accuracy of conflicting-access information
==73156== For lists of detected and suppressed errors, rerun with: -s
==73156== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2365 from 36)
```

Figure 8: No thread concurrency errors.