CSE344
Systems Programming Course
Homework #4 Report

Oğuzhan Agkuş
161044003

Deadline: May 24^{th}, 2020

1 Problem Definition

This homework is some kind of "cigarette smokers problem" in the computer science terminology. But we have chefs instead of smokers. Chefs are responsible for preparing desserts. A chef must have 4 distinct ingredients (flour, milk, sugar, walnuts) to prepare a dessert. There are 6 chefs and each one has endless supply of 2 distinct ingredients but lacks 2 distinct ingredients. Beside the chefs, there is a wholesaler who is delivers 2 distinct ingredients at each visit.

Chefs are waiting for the wholesaler to deliver the ingredients they needed. When the wholesaler arrived and delivered ingredients, s/he starts to wait for the dessert to ready. When a chef prepared a desert, the wholesaler takes it and go to sell it. And it repeats.

The main synchronization problem here is getting all ingredients or not. Think about a situation like that. The wholesaler delivered sugar and milk. There are 2 chefs, first one is waiting for sugar and milk, and the other one is waiting for sugar and walnuts. The first one gets the milk, and the other one gets the sugar. They need one more ingredient but there will not be a new ingredient until a dessert is prepared. So, the wholesaler starts wait for chefs to prepare dessert, and chefs are waiting for the wholesaler to deliver ingredients. There will occur a deadlock. The solution is ensuring that a chef gets both ingredients or none of them.

2 Solution Approach

The POSIX semaphores does not provide use solution for this problem. System V semaphores are more useful for this. Also, I have used POSIX semaphores before. I want to have experience with System V semaphores. So, I used them.

3 Code Flow

3.1 Input Checking

It checks the input file. If it is invalid (less than 10 line, more than 2 characters in a line, etc.), it exits with -1 by printing error message to stderr.

```
...

/* -------------------- */ // Command line arguments checking

if (argc != 3)
  error_exit("Less/more arguments!\n
              Usage: ./program -i input_path", -1, NULL);

/* -------------------- */ // Getopt parsing

while ((opt = getopt(argc, argv, "i:")) != -1) {
  switch (opt) {
    case "i":
      input_path = optarg; break;
    default:
      error_exit("Invalid options!\n
                  Usage: ./program -i input_path", -1, NULL);
  }
}

/* -------------------- */ // Input checking

if ((input_fd = open(input_path, O_RDONLY)) == -1)
  error_exit("Failed to open input path", errno, NULL);

if (check_input(input_fd) == -1)
  error_exit("Invalid input file.", -1, NULL);

/* -------------------- */

...
```

3.2 Initializing Shared Memory in the Heap

It creates a shared memory in the heap. Because I want to all thread can access this memory location. This location represents a space where wholesaler put the ingredients and chefs gets from. I prefer to use an array with size for this space. First two index stores ingredients and last index stores prepared dessert. I defined an enumeration to represent these indexes. Also, I defined another enumeration for representing stock information for chefs and shared memory.

```
enum index { ingr_1 = 0, ingr_2 = 1, product = 2 };
enum stock { empty = -1, lack = 0, endless = 1 };
```

```
shared_memory = init_shared_memory(); // shared_memory is global
```

Function for initializing shared memory:

```
int *init_shared_memory() {
  int *shared_memory;

  shared_memory = (int *) malloc(3 * sizeof(int));
  if (shared_memory == NULL)
    error_exit("malloc() error", -1, NULL);

  shared_memory[ingr_1] = empty;
  shared_memory[ingr_2] = empty;
  shared_memory[product] = 0;

  return shared_memory;
}
```

3.3 Initializing Semaphores

System V standards does not provide a single semaphores, it provides semaphores sets. There 6 semaphore in my set. I defined an enumeration to represent these semaphores.

```
enum semaphore { flour = 0, milk = 1, sugar = 2, walnuts = 3,
                 dessert = 4, busy = 5 };
```

First four semaphores represent availability of ingredients in the shared memory. Fifth one represents availability of the dessert. Sixth one represents if the shared memory is available for accessing and updating. Because the shared memory operations must be considered critical section.

```
if (init_sem(&sem_id) == -1) // sem_id is global
  error_exit("init_sem() error", errno, shared_memory);
```

Function for initializing shared memory:

```
int init_sem(int *sem_id) {
  int error = 0;
  union semun ar;

  *sem_id = semget(IPC_PRIVATE, 6, IPC_CREAT | IPC_EXCL | 0600);
  if (*sem_id == -1) return -1;
  ar.array = (unsigned short *) malloc(6 * sizeof(unsigned short));
  if (ar.array == NULL) return -1;

  ar.array[flour] = 0;
  ar.array[milk] = 0;
  ar.array[sugar] = 0;
  ar.array[walnuts] = 0;
  ar.array[dessert] = 0;
  ar.array[busy] = 1;

  if (semctl(*sem_id, 0, SETALL, arg) == -1) error = 1;
  free(ar.array);

  if (error == 1) return -1;
  else return 0;
}
```

3.4 Creating Threads

The wholesaler is considered as main thread, and other threads are considered as chefs. All chefs use same function. They differ in given arguments. The given arguments include these variables:

```
struct argument {
  int chef_id;
  int ingredients[4];
};
```

```
struct argument arguments[CHEF_COUNT]; // CHEF_COUNT is 6
init_arguments(arguments);
```

The init_arguments() function fill the array like below:

Chef ID	Flour	Milk	Sugar	Walnuts
1	Endless	Endless	Lack	Lack
2	Endless	Lack	Endless	Lack
3	Endless	Lack	Lack	Endless
4	Lack	Endless	Endless	Lack
5	Lack	Endless	Lack	Endless
6	Lack	Lack	Endless	Endless

After arguments is initialized it creates chef threads.

```
for (i = 0; i < CHEF_COUNT; i++) {
  result = pthread_create(&chefs[i], NULL, chef_function,
                          &arguments[i]);

  if(result != 0)
    error_exit("pthread_create() error", result, shared_memory);
}
```

3.4.1 Chef Code

There are two important points: informing about wholesaler and waiting for two ingredients atomically.

The "finished" variable is global. It is initially 0, just after the wholesaler stopped delivering ingredients it make the variable's value 1. And increase semaphores' values. Thus, semop succeed and the condition is failed. It breaks the loop and exit. In other words, chefs will know that the wholesaler will not deliver again.

The fill_ops() function fill a semaphore array according to arguments. By doing this, the thread will know what semaphores it must wait. Waiting for semaphore array is atomic. In other words, each chef will know what ingredients s/he will wait.

```
void *chef_function(void *arg) {
  int count, error = 0, delivered = 0;
  char write_buffer[256];

  struct argument *arguments = (struct argument *) arg;
  struct sembuf sops[2];

  fill_sops(arguments, sops);

  while (1) {
    // Waiting for ingredients
    if (semop(sem_id, sops, 2) == -1) { error = 1; break; }
    if (finished == 1) break;

    // Getting ingredients from shared_memory
    if (wait(sem_id, busy) == -1) { error = 1; break; }
    shared_memory[ingr_1] = empty;
    shared_memory[ingr_2] = empty;
    if (post(sem_id, busy) == -1) { error = 1; break; }

    // Preparing
    sleep(rand() % 5 + 1);

    // Putting the dessert to shared_memory
    if (wait(sem_id, busy) == -1){ error = 1; break; }
    shared_memory[product] += 1;
    if (post(sem_id, dessert) == -1) { error = 1; break; }
    if (post(sem_id, busy) == -1) { error = 1; break; }

    delivered += 1;
  }

  if (error == 1)
    error_exit("Error occured in thread", errno, shared_memory);

  pthread_exit(0);
}
```

Making operations on shared memory must be in critical section. So getting ingredients and putting desserts is done between wait(busy) and post(busy) calls. In case of any error it exits with -1. Exiting will be examined in detail in next sections.

Preparing a dessert is simulating with sleep. Its duration changes randomly between 1 and 5.

3.4.2 Wholesaler Code

Wholesaler read from input file. It make changes over shared_memory and semaphores according to characters read. When there is no character the loop ends. The critical section stuff is exactly same with the chef code.

```
 while (read(input_fd, read_buffer, 3) >= 2) {
  // Putting ingredients to shared_memory
  if (wait(sem_id, busy) == -1)
    { error = 1; break; }
  shared_memory[ingr_1] = food_type(read_buffer[0]);
  shared_memory[ingr_2] = food_type(read_buffer[1]);
  if (post(sem_id, food_type(read_buffer[0])) == -1)
    { error = 1; break; }
  if (post(sem_id, food_type(read_buffer[1])) == -1)
    { error = 1; break; }
  if (post(sem_id, busy) == -1)
    { error = 1; break; }

  // Waiting for the dessert
  if (wait(sem_id, dessert) == -1)
    { error = 1; break; }

  // Getting the dessert from shared_memory
  if (wait(sem_id, busy) == -1)
    { error = 1; break; }
  shared_memory[product] -= 1;
  if (post(sem_id, busy) == -1)
    { error = 1; break; }
}

if (error == 1)
  error_exit("semop() error in wholesaler", errno, shared_memory);

finished = 1;

for (i = 0; i < 4; i++) {
  sop.sem_num = i;
  sop.sem_op = 3;
  sop.sem_flg = 0;

  if (semop(sem_id, &sop, 1) == -1)
    error_exit("semop() error", errno, shared_memory);
}
```

After file reading is finished, in other words wholesaler stopped delivering, it make "finished" variable's value 1. And increase all ingredient semaphores values by 3. Because there are 3 chefs waiting for each ingredient.

3.5 Exiting

Before exiting main thread, it must wait for all other threads. After that, it should close the input file and remove the semaphore set. Finally it must call free() function for shared_memory which is dynamically allocated in the heap.

```c
int main(int argc, char *argv[]) {
  ...

  /* -------------------- */ // Joining threads

  for (i = 0; i < CHEF_COUNT; i++) {
    result = pthread_join(chefs[i], NULL);

    if(result != 0)
      error_exit("pthread_join() error", result, shared_memory);
  }

  /* -------------------- */ // Clean-up

  if (close(input_fd) == -1)
    error_exit("Error occured while closing the input file", errno,
               shared_memory);

  if (semctl(sem_id, 0, IPC_RMID) == -1)
    error_exit("The semaphore set cannot removed", errno,
               shared_memory);

  free(shared_memory);
  exit(EXIT_SUCCESS);
}
```

3.6 Error Handling

In case of any error, program calls error_exit(..., ..., ...) function. It prints given error message to stderr and free the allocated memory which is shared_memory in the program. Then it exits with -1. If it exit directly, the there can be occur memory leaks. Its size is not too big but its better to free manually. It can be thought that a segmentation fault occur but it does not.

```c
void error_exit(char *message, int error, int *to_free) {
  char buffer[256];
  int count;

  if (to_free != NULL) // Free shared_memory
    free(to_free);

  if (error == -1)
    count = sprintf(buffer, "%s\n", message);
  else
    count = sprintf(buffer, "%s: %s\n", message, strerror(error));

  write(STDERR_FILENO, buffer, count);
  exit(EXIT_FAILURE);
}
```

3.7 Interrupt Handling

In case of CTRL-C, the program will exit immediately. But it should not do this because it have dynamically allocated memory. It should be freed before exiting.

```
void sigint_handler(int signal) {
  int count;
  char write_buffer[256];

  free(shared_memory);
  count = sprintf(write_buffer, "%s signal caught!
                                 Terminated with CTRL-C.
                                 Shared memory has been freed.\n",
                                 sys_siglist[signal]);
  write(STDOUT_FILENO, write_buffer, count);

  exit(EXIT_FAILURE);
}
```

Setting up interrupt handler in main:

```
int main(int argc, char *argv[]) {
  ...

  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;
  sa.sa_handler = &sigint_handler;

  if (sigaction(SIGINT, &sa, NULL) == -1)
    error_exit("Cannot set new signal handler", errno, NULL);

  ...
}
```

3.8 Helper Functions

System V semaphores have a little bit complex interface. I want to simplify it by writing some wrapper functions such as wait() and post(). The wait() function is given below. The post() is exactly same form except the sem_op parameter, it is 1 in post() function. They increase/decrease a semaphore value in given set.

```
int wait(int sem_id, int semaphore) {
  struct sembuf sop;

  sop.sem_num = semaphore;
  sop.sem_op = -1;
  sop.sem_flg = 0;

  if (semop(sem_id, &sop, 1) == -1)
    return -1;

  return 0;
}
```

4 Output

All threads prints some messages to terminal.

Chefs prints when waiting for ingredients, getting ingredients, preparing dessert, delivering the dessert and stopping the production.

The wholesaler prints when delivering ingredients, waiting for dessert, getting dessert and stopping the delivery.

5 Compiling and Running

The source code includes a makefile. It compiles all cources with needed parameters like -pthread. Also it compiles with -Wall and -Wextra for warnings. In my system (Ubuntu 18.04.4, Kernel version: 5.3.0) it is compiling without any error or warning.

Before you run, you should provide a valid input file. You can edit run part in the make file or directly run like that ./program -i input_file