UE610 Operating Systems Architectures

Final Project Report

Cooking Recipe Server

Date: 10 May 2023

Team № 10:

Surname	Name	E-mail	Student ID
AHMADOV	KAMAL	kamal.ahmadov1@ufaz.az	22022692
MURADZADA	FARID	farid.muradzada@ufaz.az	22022751
MURSALOV	AKIF	akif.mursalov@ufaz.az	22022723
OSMANLI	RAVAN	ravan.osmanli@ufaz.az	22022753

Teachers: Dr. Pierre Parrend, PhD. Candidate Rauf Fatali

Introduction:

The "Operating Systems Architecture" course project is aimed at developing a cooking recipe generator server. This server is triggered by signals and prints a cooking recipe on the console. The cooking recipes are divided into three categories: "Student," "Azeri," and "French."

In the first version of the project, a simple server is created that receives signals and prints the corresponding type of cooking recipe on the screen. A client program is also created, which randomly sends one of the three signals "`SIGINT`", "`SIGQUIT`", and "`SIGTERM`" to the server, and this operation is repeated 100 times. The client and the server are dedicated programs written in the C programming language, and the association between signal type and cooking recipe category is stored in a matrix. The project report for version 1 includes the application requirements in UML Use Case diagram, the application architecture as UML Component Diagram, the code for the client and server with comments, and the output of their execution.

In version 2 of the project, a cooking recipe reader program and a cooking recipe writer program are added, which create a message queue for exchanging cooking recipe texts. All cooking recipes are stored in three directories, one for each category, and the reader reads two cooking recipes of each type. The writer program stores the cooking recipes in the respective directories and writes messages of type "1" that contain cooking recipes for students, of type "2" that contain Azeri cooking recipes, and of type "3" that contain French cooking recipes. All available cooking recipes must be read from the files and stored in the message queue. The project report for version 2 includes an updated application requirement in the UML Use Case diagram, the updated application architecture as the UML Component Diagram, the code for the reader and writer programs with comments, and the output of their execution.

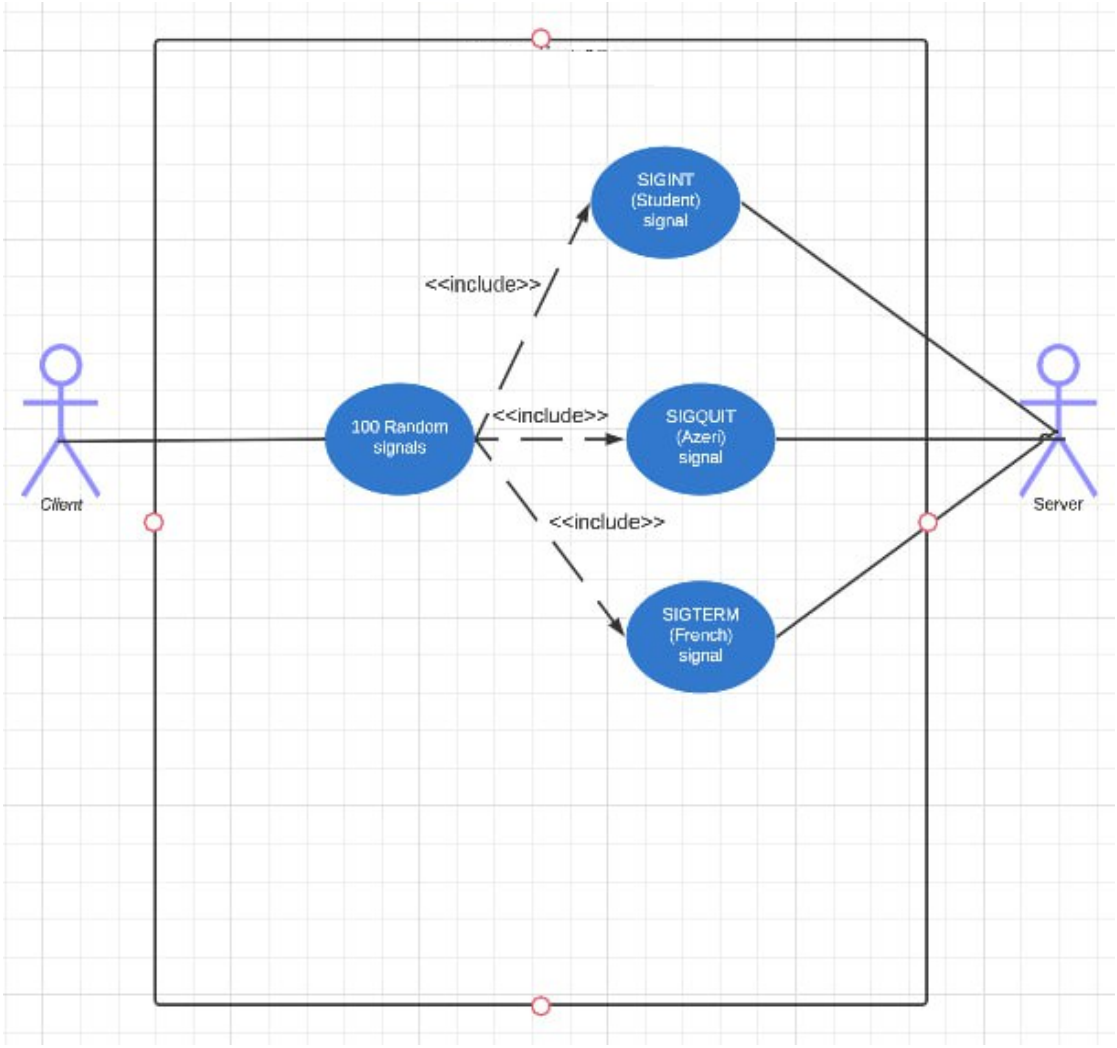
In version 3, we merge the functionalities implemented in version 1 and version 2. The cooking recipe server program utilizes the reader program's method, "read_cooking_recipe(int category)," to retrieve a cooking recipe of the appropriate type when a matching signal is received. For instance, the server will generate a "Student" recipe when it receives the "SIGINT" signal, an "Azeri" recipe upon receiving the "SIGQUIT" signal, and a "French" recipe when the "SIGTERM" signal is detected. Furthermore, in the cooking recipe writer program, we incorporate a mechanism to ensure that each category contains at least one entry. If any category is found to be empty, it will be refilled with cooking recipes read from the corresponding files. The project report for version 3 will encompass an updated UML Use Case diagram reflecting the refined application requirements, an updated UML Component Diagram showcasing the integrated application architecture, the comprehensive commented code of the entire application, and screenshots illustrating the output obtained from the execution.

Version 1:

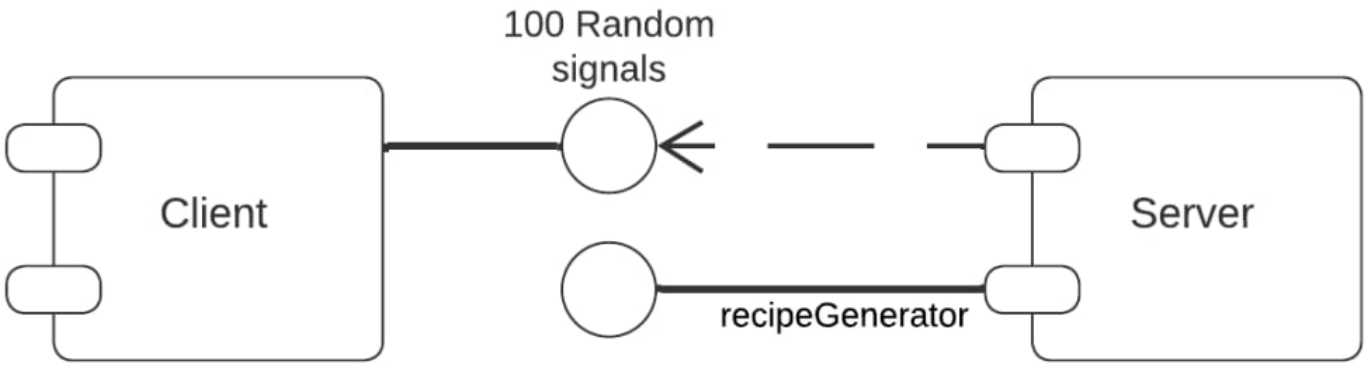
The first version of the cooking recipe server is a simple implementation that meets the basic requirements of the project. The interprocess communication is handled via signals. The server is triggered by signals and prints a cooking recipe of the corresponding category to the console. The categories of recipes are 'Student', 'Azeri', and 'French'. For version 1, the server and client are each a dedicated program written in C language.

The client ***randomly*** selects one of the three signals (`'SIGINT'`, `'SIGQUIT'`, or `'SIGTERM'`) and sends it to the server. This operation is repeated 100 times. The association between signal type and cooking recipe category is stored in a matrix. `'SIGINT'` is associated with the 'Student' category, `'SIGQUIT'` with the 'Azeri' category, and `'SIGTERM'` with the 'French' category.

Use Case Diagram:



Component Diagram:



For the first version, there are two C files: client.c and server.c

"client.c" is a program that takes a single command-line argument (a process ID) and sends a series of signals to the specified process. The program uses the "kill" function to send the signals, and it generates random signal numbers (2, 3, or 15) using the "rand" function. The program also includes a sleep function that causes it to wait for one second between sending each signal.

"server.c" is a program that listens for three specific signals (`SIGINT`, `SIGQUIT`, and `SIGTERM`) and responds to each signal by printing the details of a randomly selected recipe to the console. The program defines a struct "recipe" that contains three fields: "name", "ingredients", and "type". The program includes an array of six recipe structs that represent different dishes from various cuisines. The "`handle_signal`" function is called when one of the signals is received, and it uses the "rand" function to select a recipe from the array based on the signal received. The program prints the recipe name, ingredients, and cuisine type to the console. The program also includes an infinite loop that causes it to sleep for one second at a time.

In the server file, it was decided to store recipes in dedicated structures like this:

```
struct recipe {
    char* name;
    char* ingredients;
    char* type;
};
```

It makes it much easier to handle the recipes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>

int main(int argc, char *argv[]){
    if (argc != 2){
        printf("Usage: %s <pid>\n", argv[0]);
        exit(1);
    }
    pid_t pid = atoi(argv[1]);
    srand(time(NULL)); // Initialize the random number generator
    for (int i = 0; i < 100; i++){
        int signal = rand() % 3 + 2; // Generate a random signal number (2, 3, or 15)
        if(signal == 2 || signal == 3){
            printf("Sending signal %d to process %d\n", signal, pid);
            kill(pid, signal);
            sleep(1); // Wait for 1 second before sending the next signal
        }
        else{
            signal = 15;
            printf("Sending signal %d to process %d\n", signal, pid);
            kill(pid, signal);
            sleep(1); // Wait for 1 second before sending the next signal
        }
    }
    return 0;
}
```

client.c (Version 1)

In the client.c, the program sends randomly generated signals to a specific process ID (PID). It takes a PID as a command line argument (from argc and argv) and checks if it has been provided with the correct number of arguments.

It then initializes a random number generator and generates a random signal number between 2, 3, or 15, and sends that signal to the specified process ID using the kill() function. The program waits for 1 second before sending the next signal to the process.

If the generated signal number is not 2 or 3, the program sends signal 15 to the process instead. This loop is repeated 100 times.

The purpose of this code is to simulate various signals sent to a process, which is useful for testing the response of the process to signals.

server.c:

```
void handle_signal(int signal){
    srand(time(NULL));
    int rand_index;
    switch (signal){
    case SIGINT:
        rand_index = rand() % 2;
        printf("---------------------------------\n");
        printf("Received SIGINT. Here is a recipe for a %s:\n", recipes[rand_index].type);
        printf("Name: %s\n", recipes[rand_index].name);
        printf("Ingredients: %s\n", recipes[rand_index].ingredients);
        printf("---------------------------------\n");
        printf("\n\n");
        break;
    case SIGQUIT:
        rand_index = rand() % 2 + 2;
        printf("---------------------------------\n");
        printf("Received SIGQUIT. Here is a recipe for a %s:\n", recipes[rand_index].type);
        printf("Name: %s\n", recipes[rand_index].name);
        printf("Ingredients: %s\n", recipes[rand_index].ingredients);
        printf("---------------------------------\n");
        printf("\n\n");
        break;
    case SIGTERM:
        rand_index = rand() % 2 + 4;
        printf("---------------------------------\n");
        printf("Received SIGTERM. Here is a recipe for a %s:\n", recipes[rand_index].type);
        printf("Name: %s\n", recipes[rand_index].name);
        printf("Ingredients: %s\n", recipes[rand_index].ingredients);
        printf("---------------------------------\n");
        printf("\n\n");
        break;
    }
}
```

We have a function to handle the incoming signals. The function is called "handle_signal" and takes an integer argument "signal" which represents the type of signal received. Inside the function, the random number generator is initialized using the current time.

Depending on the value of the "signal" argument, the function selects a random index of a recipe array using the "rand()" function. The recipe array contains six elements and the indices are selected based on the signal type received. If the signal type is SIGINT, the function selects one of the first two recipes in the array. If the signal type is SIGQUIT, the function selects one of the next two recipes. Finally, if the signal type is SIGTERM, the function selects one of the last two recipes in the array.

After selecting a random index, the function prints out the recipe details for the selected index including the recipe type, recipe name, and ingredients. The function also prints out a separator line before and after printing the recipe details. Finally, the function prints out two new line characters for formatting.

For storing the recipes, we implemented a structure in the code to store the recipes and put the into an array:

```
struct recipe{
    char *name;
    char *ingredients;
    char *type;
};
struct recipe recipes[] = {
    {"Cheesy Ramen Noodles", "2 packs of ramen noodles\n2 cups of water\n1/4 cup of milk\n1/2 cup of shredded cheddar cheese\n1/4 cup of grated parmesan cheese\n1/4 tsp of garlic powder\nSalt and pepper to taste", "student"},
    {"Quesadilla Pizza", "4 tortillas\n1/2 cup of pizza sauce\n1/2 cup of shredded mozzarella cheese\n1/2 cup of chopped pepperoni\n1/4 cup of chopped green onions", "student"},
    {"Plov (Azerbaijani Rice Pilaf)", "2 cups of long-grain rice\n2 cups of water\n1/4 cup of vegetable oil\n1 large onion, chopped\n1 lb of lamb or beef, cut into bite-sized pieces\n2 carrots, grated\n1/4 cup of golden raisins\nSalt and pepper to taste", "azeri"},
    {"Dolma (Stuffed Vegetables)", "8-10 bell peppers or tomatoes\n1 lb of ground lamb or beef\n1 onion, chopped\n1/2 cup of long-grain rice\n1/4 cup of chopped fresh parsley\nSalt and pepper to taste\n2 cups of water", "azeri"},
    {"Coq au Vin (Chicken in Red Wine)", "4 chicken thighs\n4 chicken drumsticks\nSalt and pepper to taste\n4 slices of bacon, chopped\n1 onion, chopped\n2 garlic cloves, minced\n1/4 cup of all-purpose flour\n2 cups of red wine\n1 cup of chicken broth\n2 tbsp of tomato paste\n2 bay leaves\n1 tsp of dried thyme\n1 cup of sliced mushrooms\n1/4 cup of chopped fresh parsley", "french"},
    {"Ratatouille (Vegetable Stew)", "2 tbsp of olive oil\n1 onion, chopped\n3 garlic cloves, minced\n1 eggplant, chopped\n2 zucchinis, chopped\n1 red bell pepper, chopped\n1 yellow bell pepper, chopped\n2 tomatoes, chopped\n2 tbsp of tomato paste\n1 tsp of dried thyme\nSalt and pepper to taste\nChopped fresh basil for garnish", "french"}};
```

And finally, the main function of the server for Version 1:

```
int main(){
    pid_t pid = getpid();
    printf("Server process ID: %d\n", pid);
    signal(SIGINT, handle_signal);
    signal(SIGQUIT, handle_signal);
    signal(SIGTERM, handle_signal);
    while (1) sleep(1);
    return 0;
}
```

The main function starts by getting the process ID of the server and printing it to the console. Then, it sets up signal handlers for three different signals (`SIGINT`, `SIGQUIT`, and `SIGTERM`) using the `handle_signal` function. Finally, it enters an infinite loop that simply sleeps for one second at a time. This loop is used to keep the server running indefinitely until it is manually terminated or receives a signal that is handled by the `handle_signal` function.

How to execute Version 1?

1) Open a terminal.
2) Compile the server and client files using the C compiler. Run the following commands in the terminal:

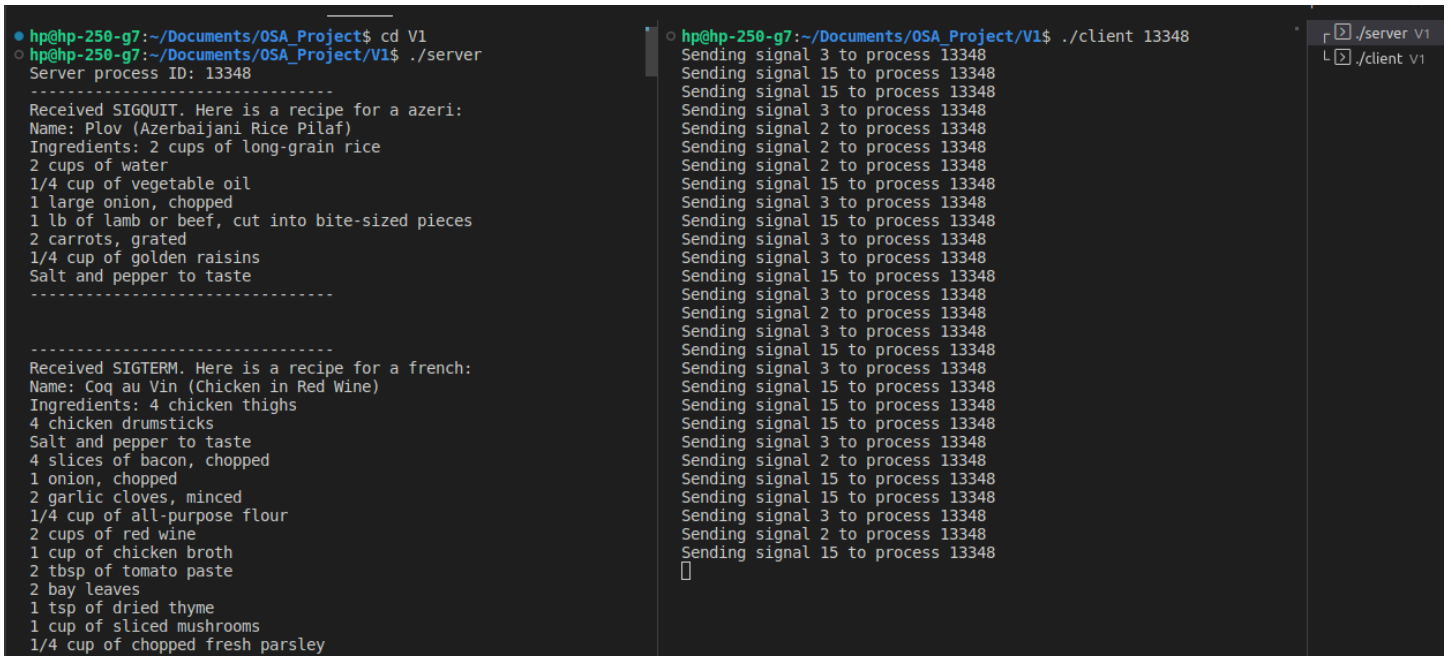
   ```
   gcc -o server server.c
   ```

   ```
   gcc -o client client.c
   ```

3) Run the server program by entering the following command in the terminal: `./server`
4) Note the process ID (PID) of the server displayed in the console. It will be needed to run the client program.
5) Run the client program with the server's process ID as an argument. Use the following command in the new terminal: `./client <pid>` (Replace <pid> with the process ID obtained in step 4.)
6) Observe the output in the server terminal window. The server will receive signals from the client and print randomly selected cooking recipes based on the signals received.

7) The client will send 100 signals to the server, and the server will respond by printing the corresponding recipe details. The recipe details will include the recipe name, ingredients, and cuisine type.
8) After the execution, you can examine the output and observe the randomly generated recipes based on the signals.

Please note that this execution assumes you are using a UNIX-based system, preferably a Linux distribution. Additionally, ensure that the necessary dependencies and libraries are installed on your system for successful compilation and execution of the C programs.



Example of running code for version 1

Version 2:

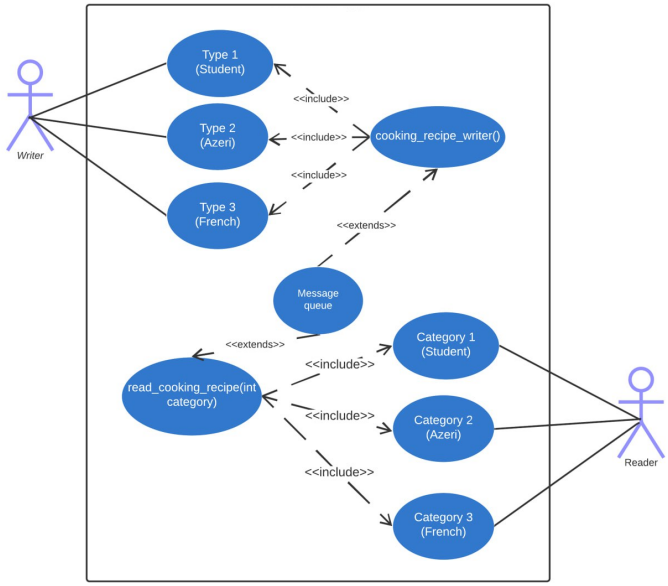
In version 2 of the Cooking Recipe Server project, we have two main programs: the Cooking Recipe Reader program and the Cooking Recipe Writer program.

The Cooking Recipe Reader program creates a message queue for exchanging cooking recipe text. All cooking recipes are stored in three directories: Student Cooking recipes, Azeri Cooking recipes, and French Cooking recipes. The program has a method read_cooking_recipe(int category) which reads the message queue of type 1 when the category is 1 (for Student recipes), the message queue of type 2 when the category is 2 (for Azeri recipes), and the message queue of type 3 when the category is 3 (for French recipes). In version 2, the reader must read two cooking recipes of each type.

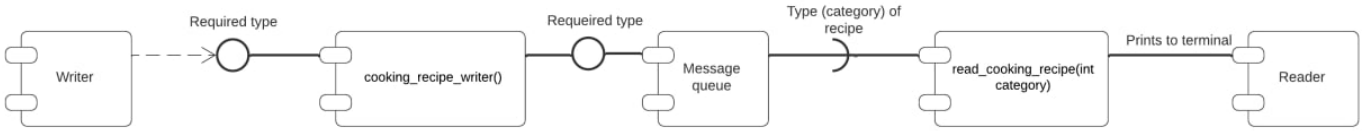
The Cooking Recipe Writer program goes through the various cooking recipe categories (represented as an array). It first stores the cooking recipes in three directories: 'students', 'Azeri', and 'French', with each cooking recipe in an individual file. The Cooking Recipe Writer then writes messages of type 1 that contain cooking recipes for students, messages of type 2 that contain Azeri cooking recipes, and messages of type 3 that contain French cooking recipes. All available cooking recipes must be read from the files and stored in the message queue.

Overall, the version 2 of the project adds more complexity by introducing a message queue and file reading/writing functionality to allow for the exchange of cooking recipe information between the reader and writer programs.

Use Case Diagram for the second version:



Component Diagram for the Second Version:



For this method, we must have 2 files: the writer and the reader.

reader.c

```c
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
// structure for message queue
struct mesg_buffer {
        long mesg_type;
        char mesg_text[500];
} message;
void read_cooking_recipe(int category){
        key_t key;
        int msgid;
        // ftok to generate unique key
        key = ftok("recipe", 65);
        // msgget creates a message queue
        // and returns identifier
        msgid = msgget(key, 0666 | IPC_CREAT);
        // msgrcv to receive message
        msgrcv(msgid, &message, sizeof(message), category, 0);
        printf("----------------------\n");
        // display the message
        printf("Data Received is:\n %s",message.mesg_text);
        printf("----------------------\n\n");
        msgctl(msgid, IPC_RMID, NULL);
}

int main(){
        key_t key;
        int msgid;
        key = ftok("recipe", 65);
        msgid = msgget(key, 0666 | IPC_CREAT);
        while(1){
                // Take the user's input
                read_cooking_recipe(1);
                read_cooking_recipe(2);
                read_cooking_recipe(3);
        }
        return 0;
}
```

This program implements a message queue system using the System V IPC (Inter-Process Communication) mechanism. It creates a message queue and allows the user to receive messages from it.

The program first defines a message buffer structure mesg_buffer, which contains two fields: `mesg_type` and `mesg_text`. `mesg_type` is a long integer and `mesg_text` is a character array of size 500.

The program defines a function `read_cooking_recipe()` which takes an integer parameter category. This function reads messages from the message queue identified by the key generated using the `ftok()` function. The `msgget()` function is used to create a message queue and returns a unique identifier. The `msgrcv()` function is used to receive messages from the message queue. The `msgctl()` function is used to delete the message queue.

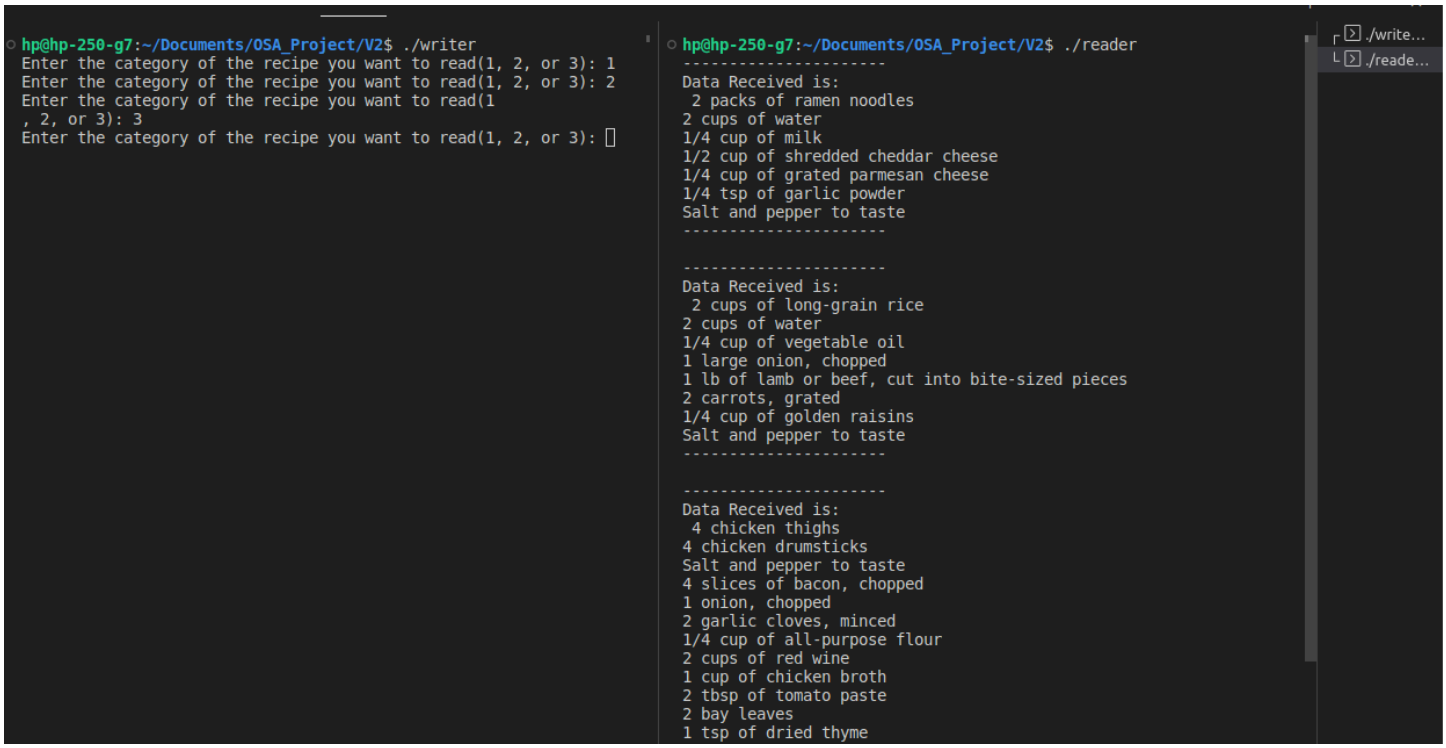
The `main()` function creates a message queue, and enters an infinite loop that repeatedly calls `read_cooking_recipe()` function with three different categories of messages (1, 2, and 3). This allows the user to receive messages from the message queue until the program is terminated.

The program outputs the message received to the console, wrapped in a horizontal line and a new line.

For the `writer.c`, it has been decided to give a shorter description.

Basically, it defines a structure for the message buffer and initiates a message queue to communicate with the reader. In the reader program, the user specifies the category and gets the recipe according to the requested category. Also the recipes are stored in the header files.

To run the code, it is recommended to first run the writer code, and then the reader (it must work either way, but that is our recommendation)



Example of running version 2

Version 3:

Version 3 can be described as the mix of the two previous versions.

In version 3 of our code, we made several enhancements to improve the modularity and organization of the cooking recipe system. We introduced header files for different cuisines, such as azeri.h and french.h, to store specific recipe information. This modular approach allows for easy management and expansion of recipes for different cuisines in the future.

The main functionality of the code remains the same: a reader process and a writer process communicate through a message queue using signals. The reader process reads cooking recipes from the message queue based on different categories, while the writer process generates random signals and sends corresponding recipes to the message queue.

The code consists of two main files: `reader.c` and `writer.c`. These files implement the reader process and writer process, respectively. Additionally, there are header files for different cuisines, including `azeri.h, french.h, and student.h,` which define structures and store recipe information.

Let's go through each component in detail:

Reader Process (reader.c)

The read_cooking_recipe function still exists in this version. It receives a category parameter and reads the cooking recipe from the message queue based on the specified category. Here are the key points of the code:

1. The `read_cooking_recipe function` receives a category parameter, which specifies the category of the recipe to be read.
2. It uses `ftok` to generate a unique key for the message queue.
3. The function calls `msgget` to create a message queue and obtain its identifier.
4. Using `msgrcv`, the process receives a message from the queue based on the specified category.
5. The received message is stored in the message structure.
6. The function then prints the received recipe to the console, adding separators for readability.

Writer Process (writer.c)

The writer process generates random signals and sends corresponding recipes to the message queue. Here's an overview of the code:

1. The `cooking_recipe_writer` function is responsible for sending cooking recipes to the message queue.
2. It generates a random signal number (2, 3, or 15) using `rand() % 3 + 2`.
3. If the signal is either 2 or 3, it prints the signal number and waits for 1 second before proceeding. If the signal is 15, it prints the signal number and waits for 1 second as well.
4. The function uses `ftok` to generate a unique key for the message queue.
5. It calls `msgget` to create the message queue and obtain its identifier.
6. The selected signal number is assigned to `message.mesg_type`.
7. Based on the signal number, the function sets the corresponding cooking recipe in `message.mesg_text`.
8. Finally, the message is sent to the message queue using `msgsnd`.

Header Files for Recipe Information

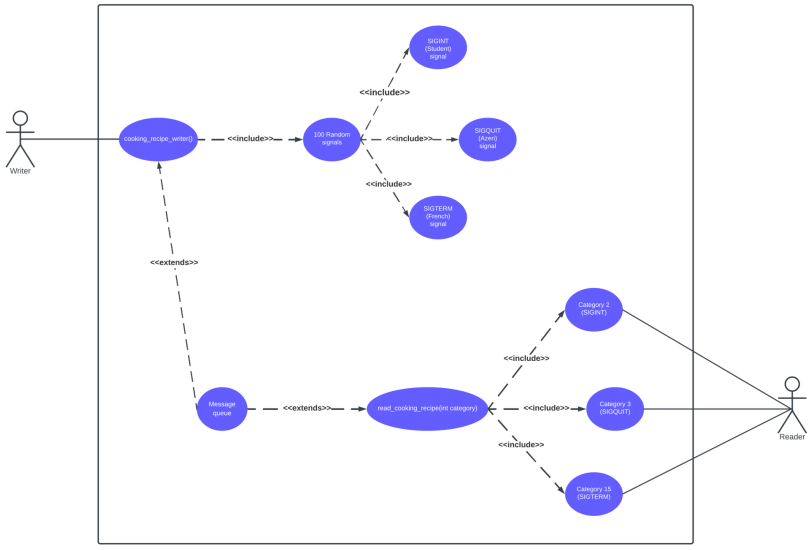
There are three header files (`azeri.h, french.h,` and `student.h`) that store recipe information for different cuisines. Each header file defines a structure for a specific cuisine and contains an array of recipe information. Here are some key points:

1. In `azeri.h`, the struct azeri defines fields such as name, ingredients, and type for Azerbaijani recipes. The `azeri_recipes` array stores specific Azerbaijani recipes along with their ingredients and type.
2. Similarly, `french.h` defines the struct french and includes the `french_recipes` array for French recipes.
3. `student.h` defines the struct student and includes the `student_recipes` array for student recipes.

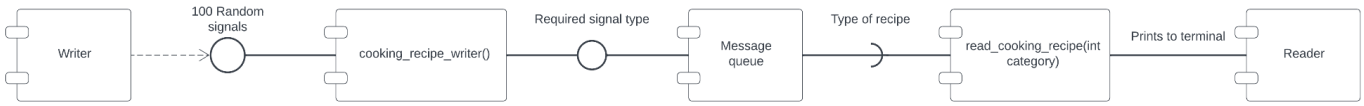
These header files provide a modular way to store and manage recipe information for different cuisines. By separating recipes into separate header files, it becomes easier to add, modify, or remove recipes without affecting the core functionality of the system.

Overall, the code demonstrates a message queue-based cooking recipe system, where the reader process reads recipes from the queue based on categories, and the writer process generates random signals and sends corresponding recipes to the queue. The inclusion of header files for different cuisines allows for easy management and expansion of recipes.

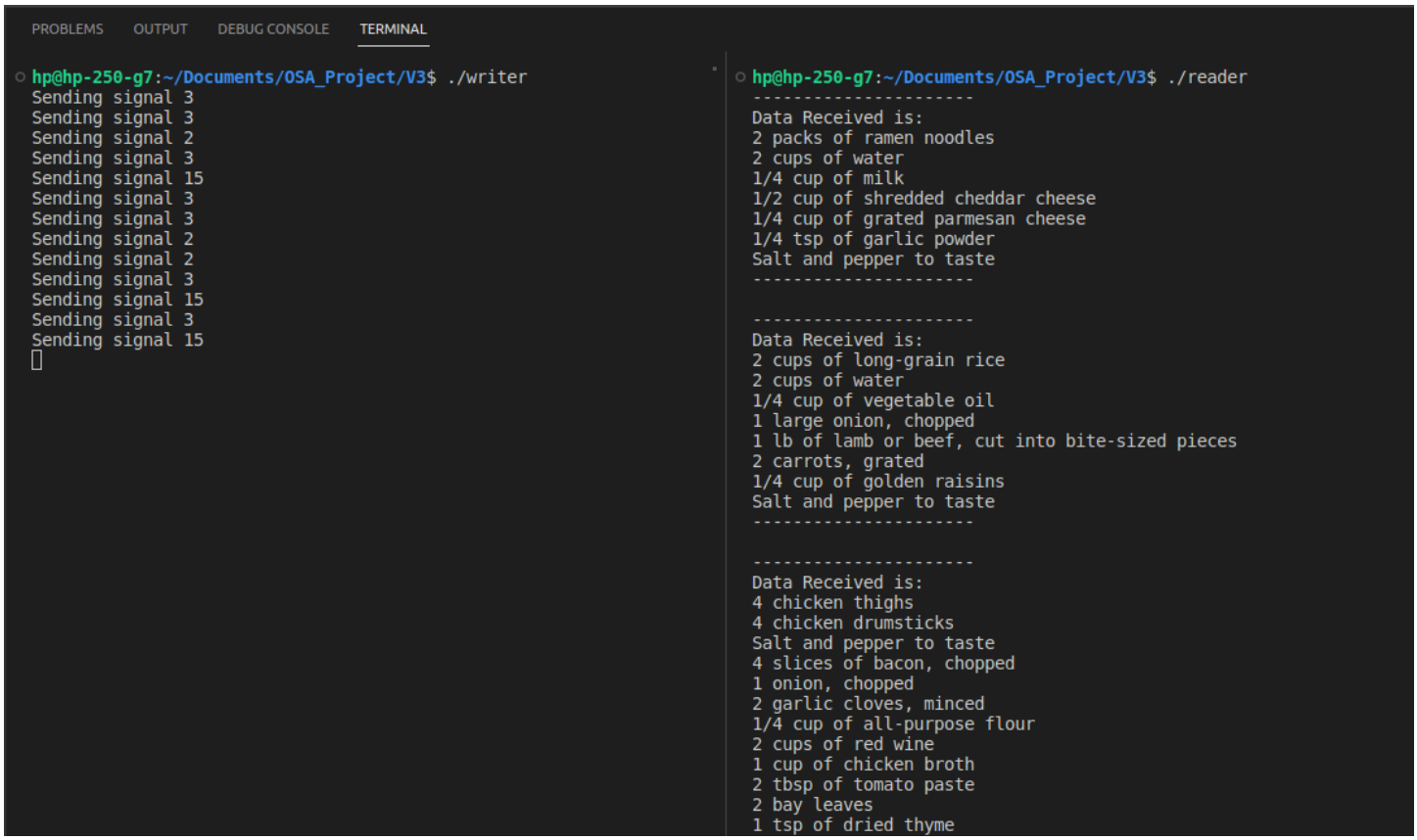
Use Case Diagram (Version 3):



Component Diagram (Version 3):



Example of running version 3



Conclusion

The "Operating Systems Architecture" course project aimed to develop a cooking recipe generator server that could be triggered by signals and print cooking recipes on the console. The project went through three versions, each adding new features and functionality.

In Version 1, a simple server and client program were created using the C programming language. The server responded to signals (SIGINT, SIGQUIT, and SIGTERM) by printing randomly selected cooking recipes of different categories (Student, Azeri, and French) to the console. The association between signals and recipe categories was stored in a matrix. The client program randomly sent signals to the server, simulating various signals being received.

Version 2 introduced the Cooking Recipe Reader and Cooking Recipe Writer programs. The reader program created a message queue for exchanging cooking recipe texts. Cooking recipes were stored in separate directories for each category, and the reader program read two cooking recipes of each type from the directories and stored them in the message queue. The writer program wrote messages to the message queue, containing cooking recipes for each category, by reading them from the files in the respective directories.

In Version 3, the functionalities of Version 1 and Version 2 were merged. The server program utilized the reader program's method, "read_cooking_recipe(int category)," to retrieve a cooking recipe of the appropriate type when a matching signal was received. Additionally, the writer program included a mechanism to refill any empty category with cooking recipes read from the corresponding files.

The project reports for each version included UML Use Case diagrams, UML Component diagrams, the code for the programs with comments, and screenshots of the program's output.

Overall, the project successfully implemented a cooking recipe generator server with the ability to handle signals and exchange cooking recipes through a message queue. The different versions added increasing complexity and functionality, making the server more robust and versatile.

Thanks for your attention :)