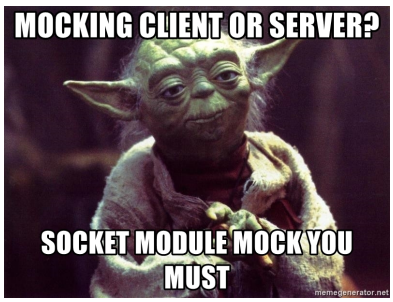
Istanbul Technical University
BLG 312E Operating Systems
Homework 2

Sadettin Fidan

May 5, 2025

1 Preamble



2 Introduction

In this homework, we have developed a multiplayer simulation system using Docker Compose. The system simulates four player clients, a central matchmaker, and a game server. The main objective is to understand inter-process communication, networked container management, and process synchronization using a realistic multiplayer game environment.

3 Architecture

The system simulates a distributed multiplayer game using six Docker containers, each fulfilling a distinct role. These containers communicate over a shared user-defined bridge network named `multiplayer_game_net`, which allows direct hostname-based socket communication between services. The containers are:

- **game_server**: Implements the core game logic. It keeps track of player health, processes attacks, and logs combat results. It listens for socket connections from players during matches.
- **matchmaker**: Manages the registration and pairing of players. It accepts socket connections from players, maintains a matchmaking queue, and pairs players for matches. Once a match is formed, it instructs both players to connect to the game server.

- **player1–player4**: Autonomous bot containers representing players. Each connects to the matchmaker, waits to be matched, and then participates in combat via the game server. All players share the same codebase and Docker image, but each is assigned a unique identity using the `PLAYER_NAME` environment variable.

The services are defined in the `docker-compose.yml` file. Shared project files are mounted using Docker volumes:

- `./common/`: Shared across all services. Contains the core threading, semaphore, and queue logic in files such as `thread.c`, `semaphore.c`, and `queue.c`, along with their corresponding headers.
- `./srcs/*/src/`: Mounted uniquely per service to include their source code (`game_server.c`, `matchmaker.c`, and `player_bot.c`).

Figure 1 shows a visual representation of this architecture. The matchmaker sits at the center, connected to all four players and the game server. Players register with the matchmaker, which pairs them and instructs them to begin combat by connecting to the game server:

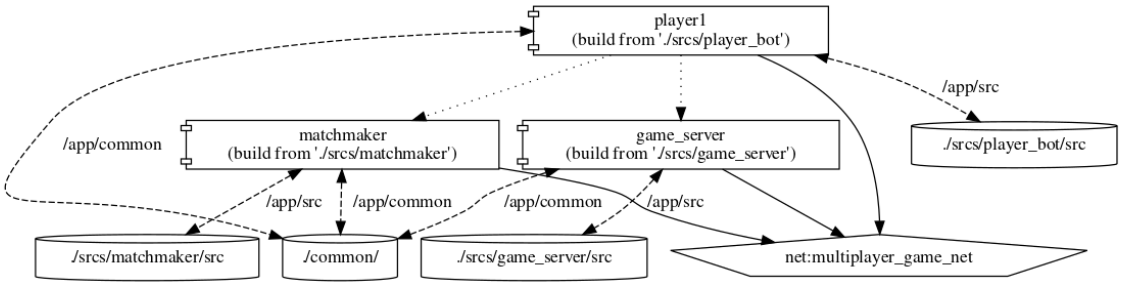


Figure 1: Container Architecture of Multiplayer Simulation

The Docker Compose file also includes `depends_on` fields to ensure proper initialization order. For example, `player1` declares dependencies on the matchmaker, game server, and player2 to avoid race conditions at startup. However, actual matchmaking and combat logic is governed at runtime by the matchmaker's logic, not container instantiation order.

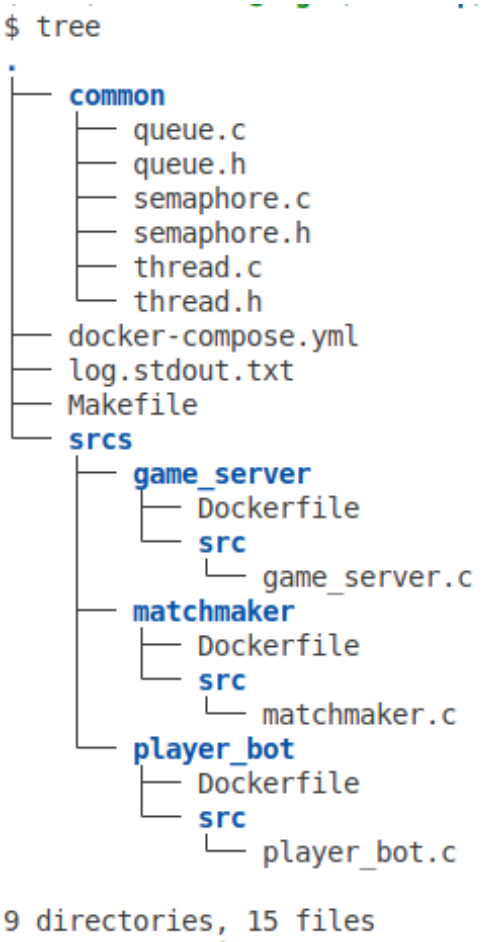


Figure 2: Project Directory Structure

For clarity, Figure 2 outlines the project's directory structure, highlighting key source files and their organization.

Build Logs: The file `log.stdout.txt` was generated using the shell command: `make > log.stdout.txt 2> log.stderr.txt`. This redirection captures the standard output of the `make` process into `log.stdout.txt` and the standard error into `log.stderr.txt`. The resulting `log.stdout.txt` file serves as a detailed build log, containing compiler messages, Docker image build output, and other informational messages produced during compilation and setup. It is a useful artifact for verifying the build process, diagnosing issues, and documenting reproducible builds.

4 Game Flow

When the simulation is launched with 'docker compose up –build', the following events occur:

1. Each player announces itself with a name and connects to the matchmaker.
2. The matchmaker logs new connections and pairs the players (e.g., Player4 vs Player3, Player2 vs Player1).
3. The game server processes combat interactions. Each attack updates the opponent's health and logs the outcome with details like damage type and remaining health.
4. Players continue to fight until one of them is defeated. A "you are dead" or "you defeated the player" message is printed accordingly.
5. Once both players in a match are finished, their containers exit with code 0.

5 Overview of the Multiplayer Game Processes

This system consists of three main components implemented in C: the **player bot**, the **matchmaker**, and the **game server**. Each runs as a separate process and communicates over TCP sockets to simulate multiplayer game sessions.

5.1 Player Bot

Each player bot acts as a client that connects to both the matchmaker and the game server. It first announces its name to the matchmaker and then joins a fight on the game server.

Listing 1: Connecting to Matchmaker and Game Server

```
connect_and_send(MATCHMAKER_PORT, "matchmaker", name, &
   match_sock);
read(match_sock, buf, sizeof(buf));
connect_and_send(SERVER_PORT, "game_server", name, &game_sock);
```

After sending the player name, the bot waits for fight messages from the server and prints them.

Listing 2: Receiving Fight Updates

```
while (1)
{
        int n = read(game_sock, buf, sizeof(buf));
        if (n <= 0)
                break;
        buf[n] = '\0';
        dprintf(1, "%s: %s\n", name, buf);
}
```

5.2 Matchmaker

The matchmaker waits for a fixed number of player connections (4 in this case), pairs them up, and sends them fight instructions.

Listing 3: Accepting Players and Creating Matches

```
int index = player_count;
players[index] = client_sock;
strcpy(names[index], name);
player_count++;
```

Once all players are connected, it informs the two players in a pair about their match:

Listing 4: Notifying Players of Match

```
snprintf(buf, sizeof(buf), "%s,%s",
                           names[index],
                           names[index+1]);
dprintf(players[index], buf);
dprintf(players[index+1], buf);
```

5.3 Game Server

The game server listens for player connections and creates a ‘Fight‘ struct for each pair. A thread is spawned to simulate the fight in turns.

Listing 5: Handling Fights with Threads

```
thread\_create(&t, handle_fight, sockets);
...
thread\_create(&t, fight_thread, f);
```

Each ‘fight_thread‘ alternates turns between two players, calculating random damage, updating health, and sending updates over the socket:

Listing 6: Fight Turn Logic

```
int damage = (rand() % 26) + 5;
defender->health -= damage;
dprintf(attacker->socket, "You␣damaged␣%s", defender->name);
dprintf(defender->socket, "You␣are␣hurt␣by␣%s", attacker->name);
```

When one player's health reaches 0, the thread notifies both sides and closes their sockets.

6 Requirements and Evaluation Criteria

The following features and questions must be implemented or answered for full credit:

Implementation Requirements

- **Queue Implementation:** Provide a basic FIFO queue for player or task handling:
  - void queue_init(Queue* q);
  - void queue_enqueue(Queue* q, void* data);
  - void *queue_dequeue(Queue* q);
  - int queue_is_empty(Queue* q);
  - void *queue_remove(Queue* q, void* data);
  - void queue_destroy(Queue* q);
- **Thread Functions:** Implement user-level threading using `ucontext.h` with round-robin switching via timer interrupt (e.g., every 10ms):
  - static void thread_init_helper(ThreadFunc func, void* arg);
  - int thread_create(Thread **thread, ThreadFunc func, void* arg);
  - void thread_yield(void);
  - void thread_join(Thread *thread);
  - void thread_init(void);

- **Semaphore Functions:** Implement semaphore operations to control access to shared resources:
  - void sem_init(Semaphore* sem, int value);
  - void sem_wait(Semaphore* sem);
  - void sem_post(Semaphore* sem);
  - void sem_destroy(Semaphore* sem);
- **Priority Queue:** Extend your scheduler to support multiple queues for different priority levels, executing them in round-robin within their level. For that, it will be enough to assign higher priority to Player4 and Player2. In the end Player4 will match with Player2, and Player3 will match with Player1 despite their order of arrival is Player4, Player3, Player2, Player1 respectively.

Conceptual Questions (to be answered in report)

1. **Semaphore Fundamentals:**
   (a) What is a semaphore and what does its value represent?
   (b) What happens if we initialize a semaphore with a value of `-1`, `0`, `1`, or a value >1?
   (c) Why does a semaphore include a queue of threads in its structure?
   (d) How we obtain Mutex from Semaphore? What is the difference? (Hint: semaphore.h, line 18-24)
2. **Thread Scheduling and `ucontext.h`:**
   (a) How is `ucontext.h` used to switch threads in your implementation?
   (b) What happens each time the timer fires an interrupt?
   (c) How would you extend the current round-robin scheduler into a priority-based scheduler using two queues?
   (d) Suppose Player1 and Player3 are higher-priority than Player2 and Player4. How can we schedule the higher-priority players first, regardless of Docker container creation order?
3. **Synchronization:**
   (a) We use mutexes to prevent race conditions — but how can race conditions still happen *inside* a mutex-protected region?
   (b) What techniques can help avoid this?

File Descriptor and Communication Questions

- Why can everything in Linux be treated like a file? What is the advantage of it?
- Compare and contrast the following:
  - `ssize_t write(int fildes, const void *buf, size_t nbyte);`
    vs.
    `int dprintf(int fd, const char *format, ...);`
  - `ssize_t send(int socket, const void *buffer, size_t length, int flags);`
    vs.
    `int dprintf(int fd, const char *format, ...);`
  - `ssize_t recv(int socket, void *buffer, size_t length, int flags);`
    vs.
    `ssize_t read(int fildes, void *buf, size_t nbyte);`
  - Describe a scenario where you might use `dscanf` or a similar custom input format. Actually there is no dscanf, guess why?

7 Sample Output

Below is a real snapshot of the output generated during the simulation. It shows how the matchmaker pairs players and how the game server handles each interaction:

```
 matchmaker-1 | Matchmaker:  Player4 vs Player3
game_server-1 | Player4 damaged Player3 by 29 (heavy attack)
game_server-1 | Player3 health is 71
...
player3-1 | Player3:  you are dead, killed by opponent
player4-1 | Player4:  you defeated the player
```

8 Submission

- **Codes:** Archive your project into a single zip file named "StudentNo.NameSurname.OS.HW2.zip" the project should be runnable with the single "make" command with the same Makefile, docker-compose.yml, and Dockerfiles.
- **Report:** We expect your reports not to be long and wordiness, just use your own sentences, no LLM babblings. We also expect your report to include the references (webpage urls, youtube videos, udemy/coursera/etc. courses, lecture notes). We do not want to feel like you always ask to AI, that's all. No need for introduction, conclusion etc. Only answers to the questions and explanations to your implementation with your algorithm by providing code segments. We also expect you to explain every function whose prototypes are provided inside semaphore.h, thread.h, and queue.h. Please create your report as "StudentNo.NameSurname.OS.HW2.pdf".
- **Logs:** Please run 'make > log.stdout.txt 2> log.stderr.txt' and, after player containers exit, hit CTRL+C once and only once, wait for all the containers to exit, then submit these two text documents as is.

9 Constraints

Attention: Points will be deducted! So please follow these:

- Please do not modify any header file
- You could modify C source files unders srcs/ folder.
- You may not modify C source files under common/ folder, except the required function bodies
- Please do not leave print statements you previously put there for debugging.
- Please do not modify docker-compose.yml and Makefile.
- Not necessarily each of the code lines you implement needs commenting, but each code segment needs a clear explanation right above it. Please do not put comments similar to "I assigned this pointer to that pointer", so we expect explanations in pseudo format like "This code segment initializes the system timer to send interrupts every 10ms, and for that I set this variable under this struct to 10000.".

10 Conclusion

This project demonstrates the use of Docker Compose for simulating a distributed multiplayer game environment. It effectively showcases how isolated containers can communicate and coordinate using socket programming, while also modeling realistic gameplay scenarios. The matchmaker and game server act as synchronization points and decision-makers, reflecting key operating systems concepts such as concurrency, inter-process communication, and distributed systems.

In addition to mastering container orchestration, the project emphasizes core systems programming skills by requiring the implementation of user-level threads and semaphores. Developing a thread library using `ucontext.h` teaches how to manage execution contexts, handle cooperative multitasking, and implement scheduling policies such as round-robin or priority-based dispatching. The custom semaphore implementation introduces synchronization primitives from the ground up, illustrating how blocking, queuing, and resource sharing are managed without relying on kernel-level constructs. These implementations deepen understanding of race conditions, critical sections, and low-level synchronization, which are fundamental for building reliable concurrent software.