School of Computing Sciences
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology, Haripur, Pakistan

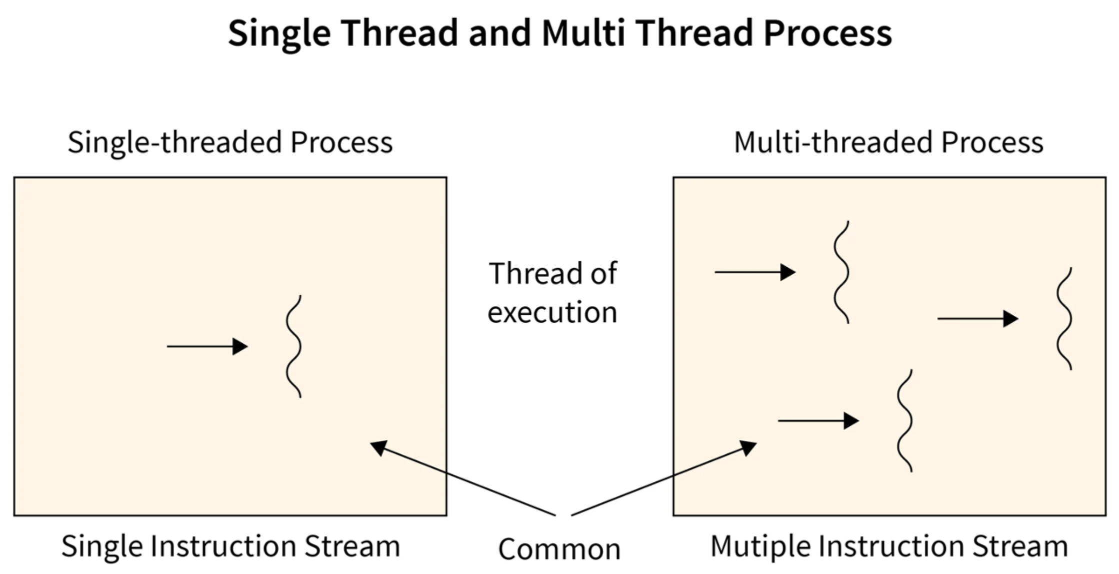
Lab 06
Multithreaded Server

Course: COMP-352L (Computer Networks Lab)

Lab Demonstrator: Jarullah Khan

Multithreading

- Multithreading is a technique that allows a single CPU to execute multiple parts of a program (*threads*) simultaneously, improving application responsiveness, speed, and resource utilization.
- Threads
  - A thread is the smallest unit of execution within a process.
  - It's a distinct, independent flow of execution within an application that can run concurrently with other threads of the same process.
  - Threads are "lightweight" because they share the same memory space and resources as their parent process, allowing for efficient multitasking and improved application performance by leveraging multiple CPU cores or by creating the illusion of parallel execution on a single core.
- Instead of running a single thread at a time, it splits a program into smaller, independent threads that can run in parallel on multiple CPU cores or concurrently on a single core.
- This is particularly beneficial for applications that handle multiple user requests or perform complex, non-dependent tasks.


Single Thread and Multi Thread Process
Single-threaded Process
Multi-threaded Process
Thread of execution
Single Instruction Stream
Common
Mutiple Instruction Stream

<u>Multithreading in Python</u>

For multithreaded programming in python, we use python's `threading` module.
The threading module is particularly useful for **I/O-bound tasks**, operations that involve waiting for external resources, such as network requests, file I/O, or database interactions, can benefit from threading by allowing other tasks to proceed while waiting.

Key features and functionalities of the `threading` module:

- **Thread Creation:** Threads are typically created by instantiating (creating an object of) the `threading.Thread` class, passing a target function and optional arguments to be executed by the thread.

```
import threading

def my_function(arg1, arg2):
    print(f"Thread executing with args: {arg1}, {arg2}")

my_thread = threading.Thread(target=my_function, args=("value1", "value2"))
```

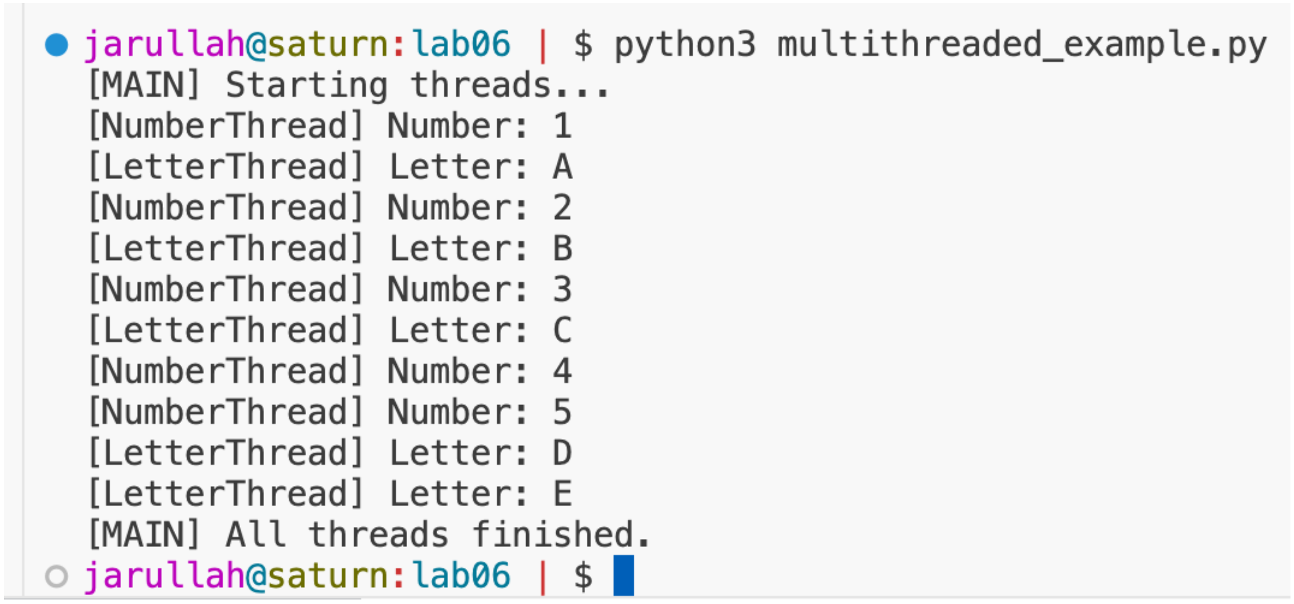
- **Starting and Stopping Threads:** The `start()` method initiates the execution of a thread, while the `join()` method can be used to block the main program's execution until a specific thread completes.

```
my_thread.start()  # Start the thread
my_thread.join()   # Wait for the thread to finish
```

- **Synchronization Basics:**
  The module offers various synchronization mechanisms to manage shared resources and prevent race conditions between threads, including:
  - Locks (`threading.Lock`): Used to ensure that only one thread can access a critical section of code at a time.
  - Events (`threading.Event`): Allow threads to signal each other about the occurrence of an event.

Program(multithreaded_example.py)

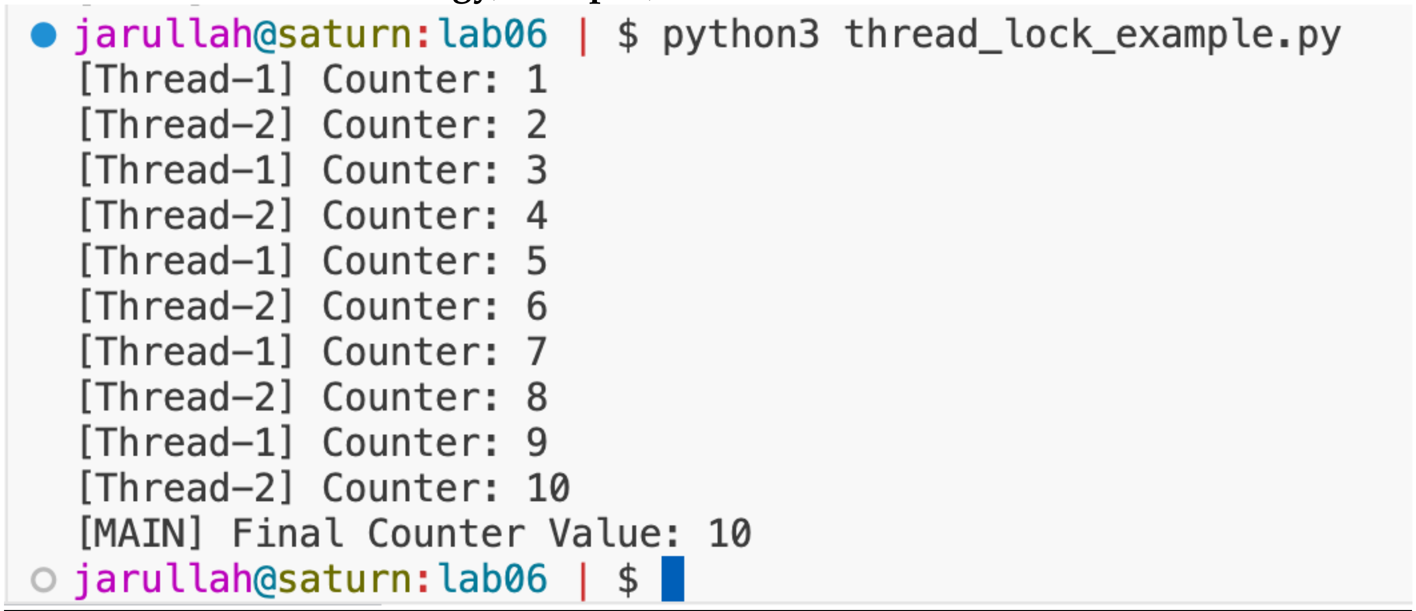
```python
import threading
import time

def print_numbers():
    for i in range(1, 6):
        print(f"[{threading.current_thread().name}] Number: {i}")
        time.sleep(1)

def print_letters():
    for letter in "ABCDE":
        print(f"[{threading.current_thread().name}] Letter: {letter}")
        time.sleep(1.5)

if __name__ == "__main__":
    # Create two threads
    t1 = threading.Thread(target=print_numbers, name="NumberThread")
    t2 = threading.Thread(target=print_letters, name="LetterThread")

    print("[MAIN] Starting threads...")
    t1.start()
    t2.start()

    # Wait for both threads to complete
    t1.join()
    t2.join()

    print("[MAIN] All threads finished.")
```



```
jarullah@saturn:lab06 | $ python3 multithreaded_example.py
[MAIN] Starting threads...
[NumberThread] Number: 1
[LetterThread] Letter: A
[NumberThread] Number: 2
[LetterThread] Letter: B
[NumberThread] Number: 3
[LetterThread] Letter: C
[NumberThread] Number: 4
[NumberThread] Number: 5
[LetterThread] Letter: D
[LetterThread] Letter: E
[MAIN] All threads finished.
jarullah@saturn:lab06 | $
```

Program (Synchronization example: shared access)

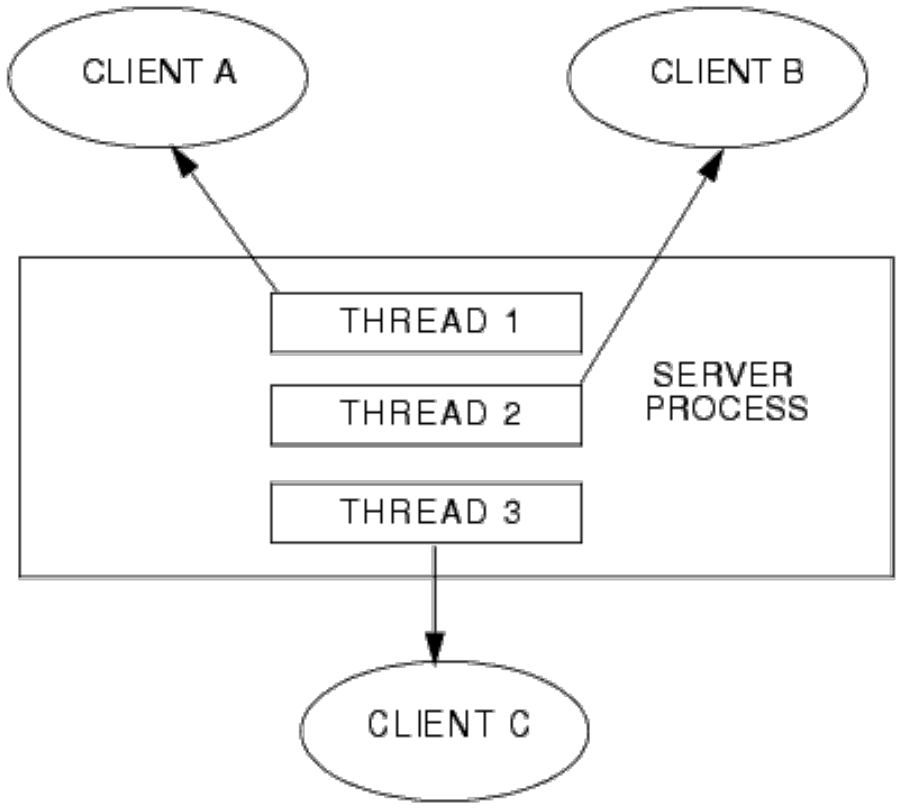
```python
import threading
import time

# Shared resource
counter = 0

# Create a lock
lock = threading.Lock()

def increment_counter():
    global counter
    for _ in range(5):
        time.sleep(0.5)
        with lock:  # acquire lock before modifying shared resource
            local_copy = counter
            local_copy += 1
            time.sleep(0.1)  # simulate some work
            counter = local_copy
            print(f"[{threading.current_thread().name}] Counter: {counter}")

#Execute these statements when the file is run as a program
#and not imported as a module
if __name__ == "__main__":
    # Create two threads that increment the same counter
    t1 = threading.Thread(target=increment_counter, name="Thread-1")
    t2 = threading.Thread(target=increment_counter, name="Thread-2")

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print(f"[MAIN] Final Counter Value: {counter}")
```



```
● jarullah@saturn:lab06 | $ python3 thread_lock_example.py
  [Thread-1] Counter: 1
  [Thread-2] Counter: 2
  [Thread-1] Counter: 3
  [Thread-2] Counter: 4
  [Thread-1] Counter: 5
  [Thread-2] Counter: 6
  [Thread-1] Counter: 7
  [Thread-2] Counter: 8
  [Thread-1] Counter: 9
  [Thread-2] Counter: 10
  [MAIN] Final Counter Value: 10
○ jarullah@saturn:lab06 | $ 
```

Multithreaded Server

- A multithreaded server is a server that uses multiple threads of execution to handle client requests concurrently, allowing it to serve multiple clients at the same time and improving performance, efficiency, and responsiveness.
- Instead of processing one client request at a time, a multithreaded server can assign a separate thread to each new incoming connection, enabling parallel handling of tasks.



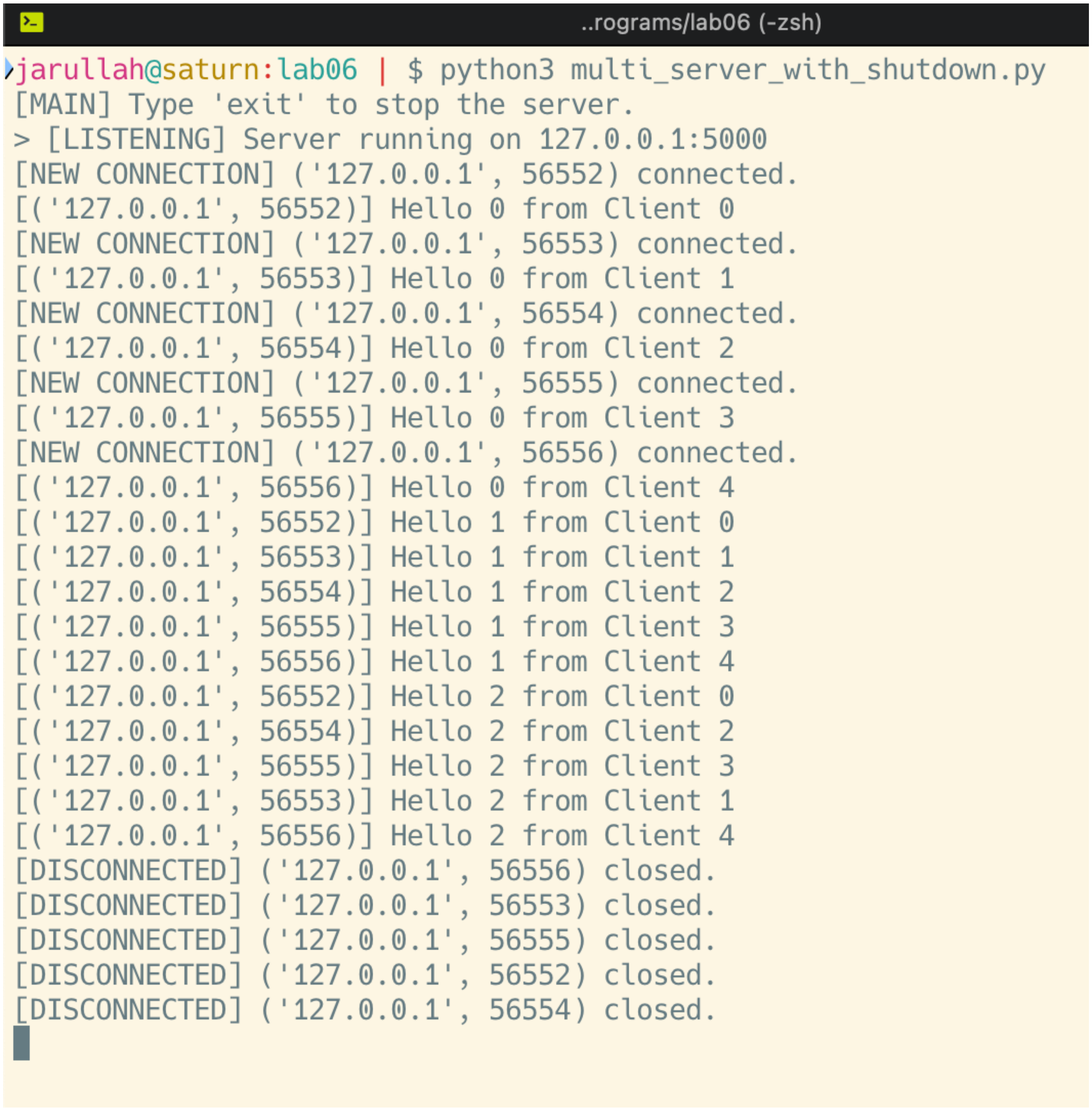
multithreaded_server.py

```
import socket
import threading

# Define server host and port
HOST = "127.0.0.1"  # localhost
PORT = 5000         # non-privileged port

# Function to handle client connections
def handle_client(client_socket, client_address):
    print(f"[NEW CONNECTION] {client_address} connected.")
    try:
        while True:
            # Receive message from client
            message = client_socket.recv(1024).decode("utf-8")
            if not message:
                break  # client disconnected

            print(f"[{client_address}] {message}")

            # Send response back to client
            response = f"Echo: {message}"
            client_socket.send(response.encode("utf-8"))
    except ConnectionResetError:
        print(f"[ERROR] Connection reset by {client_address}")
    finally:
        client_socket.close()
        print(f"[DISCONNECTED] {client_address} closed.")

# Main server loop
def start_server():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((HOST, PORT))
    server.listen(5)  # max queued connections
    print(f"[LISTENING] Server is listening on {HOST}:{PORT}")

    while True:
        client_socket, client_address = server.accept()
        # Start a new thread for each client
        thread = threading.Thread(target=handle_client, args=(client_socket, client_address))
        thread.start()
        print(f"[ACTIVE CONNECTIONS] {threading.active_count() - 1}")

if __name__ == "__main__":
    print("[STARTING] Server is starting...")
    start_server()
```

client.py

```python
import socket

HOST = "127.0.0.1"
PORT = 5000

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((HOST, PORT))

print("Connected to server. Type messages...")
while True:
    msg = input("You: ")
    if msg.lower() == "quit":
        break
    client.send(msg.encode("utf-8"))
    response = client.recv(1024).decode("utf-8")
    print(f"Server: {response}")

client.close()
```

Server output after conversation with multiple clients



```
..rograms/lab06 (-zsh)
jarullah@saturn:lab06 | $ python3 multi_server_with_shutdown.py
[MAIN] Type 'exit' to stop the server.
> [LISTENING] Server running on 127.0.0.1:5000
[NEW CONNECTION] ('127.0.0.1', 56552) connected.
[('127.0.0.1', 56552)] Hello 0 from Client 0
[NEW CONNECTION] ('127.0.0.1', 56553) connected.
[('127.0.0.1', 56553)] Hello 0 from Client 1
[NEW CONNECTION] ('127.0.0.1', 56554) connected.
[('127.0.0.1', 56554)] Hello 0 from Client 2
[NEW CONNECTION] ('127.0.0.1', 56555) connected.
[('127.0.0.1', 56555)] Hello 0 from Client 3
[NEW CONNECTION] ('127.0.0.1', 56556) connected.
[('127.0.0.1', 56556)] Hello 0 from Client 4
[('127.0.0.1', 56552)] Hello 1 from Client 0
[('127.0.0.1', 56553)] Hello 1 from Client 1
[('127.0.0.1', 56554)] Hello 1 from Client 2
[('127.0.0.1', 56555)] Hello 1 from Client 3
[('127.0.0.1', 56556)] Hello 1 from Client 4
[('127.0.0.1', 56552)] Hello 2 from Client 0
[('127.0.0.1', 56554)] Hello 2 from Client 2
[('127.0.0.1', 56555)] Hello 2 from Client 3
[('127.0.0.1', 56553)] Hello 2 from Client 1
[('127.0.0.1', 56556)] Hello 2 from Client 4
[DISCONNECTED] ('127.0.0.1', 56556) closed.
[DISCONNECTED] ('127.0.0.1', 56553) closed.
[DISCONNECTED] ('127.0.0.1', 56555) closed.
[DISCONNECTED] ('127.0.0.1', 56552) closed.
[DISCONNECTED] ('127.0.0.1', 56554) closed.
```

School of Computing Sciences
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology, Haripur, Pakistan

TASKS

Task 01	Add registration and login functionality to the multithreaded server. Multiple clients should be able to register and login at the same time. Create a test program that creates multiple clients which simultaneously interact with the server. Half of the clients should be registering a new account. Half of the clients should simultaneously: • Login. • Have a simple conversation(a few messages back and forth) • Logout
Task 02	Add the ability to **terminate** the server cleanly. • Add a command (exit) in the server console that shuts it down. • Ensure all threads are joined before exiting.
Task 03	Keep track of connected clients. • Maintain a dictionary of {client_id: socket}. • Display the number of active clients. • Add a list command (server-side) to print all connected clients.
Task 04	Build a simple group chat. • Add a `broadcast` command so a client's message is sent to **all clients**. • Include the sender's ID in messages.