Assignment 5: Non-Blocking Task Queue Server for Distributed Job Processing

Deadline: March 17, 2025 EOD

SUBMISSION INSTRUCTIONS

Zip all these files as instructed as the end of this assignment into a single zip file <NAME>_<ROLL NO>.zip and submit on MS Teams.

NOTES:

1. You should mention your name (as per ERP), roll number and the Assignment number within the comment line at the beginning of each program. A sample header will look as follows.
   ==========
   Assignment 3 Submission
   Name: <Your_Name>
   Roll number: <Your_Roll_Number>
   ==========
2. The code should have proper documentation and indentation. Unreadable codes will not be evaluated. You should submit a server and a client program as a part of this assignment.
3. The code should get executed to the lab machines. If we get a compilation error or runtime error during executing your code on the lab machines, appropriate marks will be deducted.
4. Any form of plagiarism will incur severe penalties. You should not copy the code from any sources. You may consult online sources or your friend, but at the end, you should type the program yourself. We may ask you to explain the code, and if you fail to do so, you'll be awarded zero marks.

This assignment will introduce students to non-blocking I/O using `fcntl()` and `O_NONBLOCK` while implementing a task queue system. This use case models real-world distributed job processing, where multiple worker clients fetch tasks from a central server.

This assignment simulates a distributed job processing system where a central Task Queue Server manages a list of computational tasks (simple arithmetic operations) and assigns them to Worker Clients that connect to the server. The worker clients request tasks, compute results, and send the results back to the server. The key components are:

1. **Task Queue Server** (Handles task assignments and multiple worker clients using non-blocking I/O).
2. **Worker Client** (Requests tasks, processes them, and sends back the results).

You are required to implement a Task Queue Server that allows multiple worker clients to fetch and process tasks asynchronously. The Task Queue Server will do the following.

1. The server maintains a queue of computational tasks, where each task is represented as a simple arithmetic operation (e.g., 23 + 47, 56 * 2). The tasks are

stored in a config file, where each line denotes a task. A sample task config file may look as follows:

```
20 + 67
32 * 34
22 - 10
78 / 12
45 + 67
```

2. The server handles multiple worker clients using non-blocking I/O (`fcntl()` and `O_NONBLOCK`) to prevent blocking while waiting for client requests. The clients connect to the server (as discussed below) and requests for a task.
3. The server allows workers to request new tasks by sending "`GET_TASK`" and respond with the computed result.
4. The server then spawns a child process for each client using `fork()`, where each child fetches and processes a task from the queue. Note that there can be multiple clients requesting for a task. Therefore, you need to safely assign a task to a client. Also, a worker client may get terminated without processing a task. You need to also handle the same in your code.
5. Also, a single worker client may request for multiple tasks through multiple "`GET_TASK`" requests. However, it cannot request for a new task until it has processed the previous task assigned to it. The server should check the same.
6. The server removes completed tasks from the queue and dynamically assigns new ones to requesting clients.
7. It allows workers to disconnect safely by sending "`exit`".
8. When a client (worker) disconnects, a zombie process may be created. The server code should handle such zombie processes.

The Worker Clients will implement the followings:

1. The worker sends "GET_TASK" to the server.
2. The server responds with an arithmetic task (sample format for the message: "`Task: 23 + 47`").
3. The client parses the received task (e.g., "`Task: 23 + 47`") and performs the arithmetic operation
4. The worker sends the result in the form "`RESULT XX`" back to the server, where `XX` is the result
5. If the server responds with "No tasks available", the worker disconnects.

Constraints:

- Use `fcntl()` to set the client socket to non-blocking mode.
- Do not use `select(), poll(), or epoll()`.
- Handle process termination correctly using `SIGCHLD` to avoid zombie processes.
- Ensure the task queue is shared safely among multiple processes.