PA6: Client–Server Communication in C Using Interprocess Communication

Objective

To implement a client–server architecture in C that demonstrates interprocess communication (IPC) using UNIX mechanisms such as pipes or sockets. Students will explore how separate processes exchange structured data and maintain synchronization.

Design Decisions

For this programming assignment, I selected Option A (UNIX Named Pipes / FIFOs) as the IPC mechanism. FIFOs provide a simple and reliable method for local interprocess communication without requiring socket addresses or network configuration. The server creates a well-known FIFO /tmp/arith_req_fifo to receive all client requests. Each client creates its own FIFO named /tmp/arith_resp_<PID>.fifo. This design gives every client a private response path, prevents response collisions, and allows concurrency naturally.

To meet the assignment requirement of demonstrating fork(), the server uses a fork-per-request model. Once a full request structure is read from the request FIFO, the server forks. The child process handles computation and writes the result to the specific client FIFO, while the parent returns immediately to listen for new requests. This design ensures that a long-running or blocked computation from one client does not delay other clients.

Requests and responses are transmitted as fixed-size binary C structs (request_msg_t and response_msg_t). Each request struct carries the operation (add/sub/mul/div), two 64-bit integer operands, the client PID, and the exact response FIFO path. The response struct contains a 64-bit result, a success flag, and an optional error message. This results in constant-size framing, no parsing overhead, and low risk of partial message interpretation errors.

For robustness, the server uses full-frame read/write helper functions to avoid partial reads or partial writes on pipes. The server also handles SIGCHLD to reap children automatically and avoid zombie processes. Error handling is done in both directions: invalid operations and divide-by-zero generate structured error responses instead of crashing or leaving the client hanging.

Verbose server-side printing was added to support demo evaluation — the server prints messages when it receives a request, computes a result, and sends the response. All interactions are also appended to server.log for traceability.

Testing Results

Multiple tests were executed using multiple client terminals simultaneously to confirm concurrency behavior. Each client successfully created its unique FIFO, sent requests, and received the correct responses. Verified test operations include addition, subtraction, multiplication, and division. Testing also included non-numeric input at the client, invalid operations (e.g. “pow”), and divide-by-zero. In all cases, the server returned a formatted error response message rather than crashing or hanging.

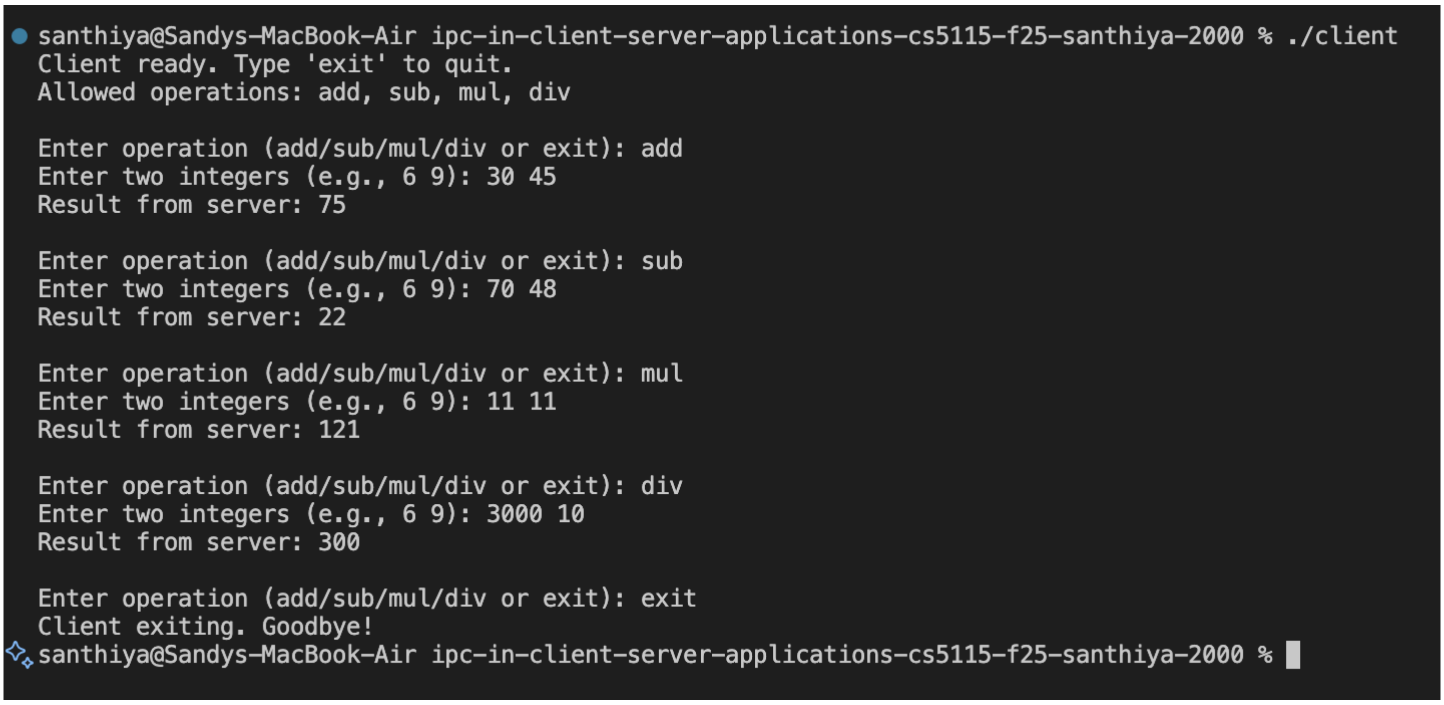
Concurrency testing: three separate client processes were run in parallel, each looping different operations. While one client was slow (intentionally idle between inputs), other clients continued receiving immediate replies — confirming that fork() isolates each request independently and the parent server remains responsive.

Robustness testing was performed deliberately by force closing clients mid-transfer. The server handled missing response FIFO errors correctly; it logged when it could not open a client FIFO for writing and continued processing future requests normally. Stale FIFOs under /tmp were removable manually without impacting stable operation.

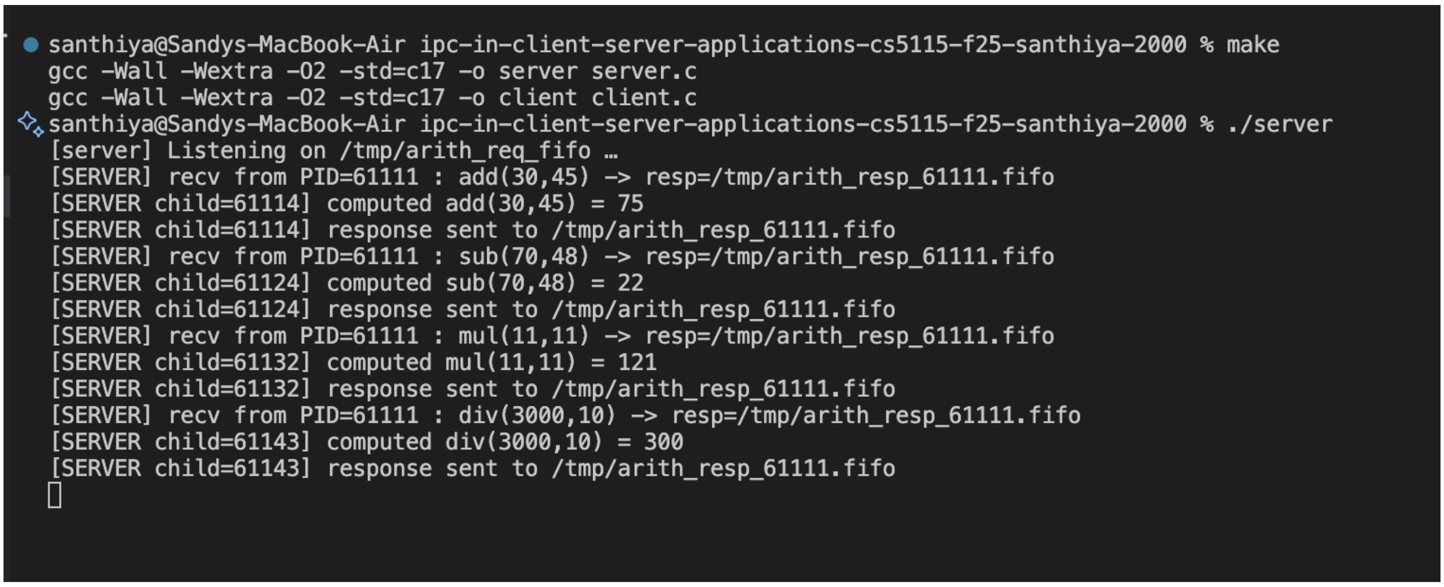
Performance is effectively instant for arithmetic workloads, and fork-per-request latency is negligible for this scale. The FIFO approach proved very stable and predictable.

Overall, the tested system meets all stated PA6 functional objectives: reliable request–response IPC, use of fork(), structured data passing, concurrency, error handling, logging, and correct cleanup behavior.

Results:



```
● santhiya@Sandys-MacBook-Air ipc-in-client-server-applications-cs5115-f25-santhiya-2000 % ./client
  Client ready. Type 'exit' to quit.
  Allowed operations: add, sub, mul, div

  Enter operation (add/sub/mul/div or exit): add
  Enter two integers (e.g., 6 9): 30 45
  Result from server: 75

  Enter operation (add/sub/mul/div or exit): sub
  Enter two integers (e.g., 6 9): 70 48
  Result from server: 22

  Enter operation (add/sub/mul/div or exit): mul
  Enter two integers (e.g., 6 9): 11 11
  Result from server: 121

  Enter operation (add/sub/mul/div or exit): div
  Enter two integers (e.g., 6 9): 3000 10
  Result from server: 300

  Enter operation (add/sub/mul/div or exit): exit
  Client exiting. Goodbye!
  santhiya@Sandys-MacBook-Air ipc-in-client-server-applications-cs5115-f25-santhiya-2000 %
```



```
● santhiya@Sandys-MacBook-Air ipc-in-client-server-applications-cs5115-f25-santhiya-2000 % make
  gcc -Wall -Wextra -O2 -std=c17 -o server server.c
  gcc -Wall -Wextra -O2 -std=c17 -o client client.c
  santhiya@Sandys-MacBook-Air ipc-in-client-server-applications-cs5115-f25-santhiya-2000 % ./server
  [server] Listening on /tmp/arith_req_fifo …
  [SERVER] recv from PID=61111 : add(30,45) -> resp=/tmp/arith_resp_61111.fifo
  [SERVER child=61114] computed add(30,45) = 75
  [SERVER child=61114] response sent to /tmp/arith_resp_61111.fifo
  [SERVER] recv from PID=61111 : sub(70,48) -> resp=/tmp/arith_resp_61111.fifo
  [SERVER child=61124] computed sub(70,48) = 22
  [SERVER child=61124] response sent to /tmp/arith_resp_61111.fifo
  [SERVER] recv from PID=61111 : mul(11,11) -> resp=/tmp/arith_resp_61111.fifo
  [SERVER child=61132] computed mul(11,11) = 121
  [SERVER child=61132] response sent to /tmp/arith_resp_61111.fifo
  [SERVER] recv from PID=61111 : div(3000,10) -> resp=/tmp/arith_resp_61111.fifo
  [SERVER child=61143] computed div(3000,10) = 300
  [SERVER child=61143] response sent to /tmp/arith_resp_61111.fifo
```