Interprocess Communication with Daemon Process
CSE 344 - System Programming
Homework #2

Ziya Kadir TOKLUOGLU
Student ID: 210104004228

April 8, 2025

Submission Date: April 8, 2025 23:59

Contents

1 Introduction

This project implements an Interprocess Communication (IPC) system using named pipes (FIFOs) and daemon processes as specified in the homework assignment. The system consists of a parent process that creates two child processes and handles communication between them through FIFOs. The first child process receives two integers from the parent via one FIFO, determines which is larger, and sends the result to the second child through another FIFO. The second child then displays the larger number.

Additionally, the parent process is converted into a daemon to run in the background and handle logging operations. The system implements signal handling for proper process management and termination, including the handling of SIGCHLD, SIGUSR1, SIGHUP, and SIGTERM signals.

This assignment demonstrates key concepts in system programming, including interprocess communication, daemon processes, signal handling, and file operations.

2 Implementation Details

2.1 FIFO Synchronization and Blocking Issues

One of the most critical aspects of this implementation was handling the synchronization of FIFO operations to prevent deadlocks. FIFOs have a particular behavior in Unix/Linux: opening a FIFO for reading blocks until another process opens it for writing, and vice versa. This can lead to deadlocks if the processes are not properly coordinated.

2.1.1 Strategic Timing for FIFO Operations

To address the blocking issues with FIFOs, I implemented a carefully timed approach:

1. **Child Process 1** opens FIFO1 for reading first, then sleeps for 11 seconds before processing:

```
// Open first FIFO for reading
fifo1_fd = open(FIFO1_PATH, O_RDONLY);

// Sleep for 10 seconds as required by assignment
// +1 more second needs to wait because of the sequence of the opening FIFOs
sleep(11);
```

Listing 1: Child 1 FIFO opening with strategic delay

2. **Child Process 2** opens FIFO2 for reading first, then sleeps for 10 seconds before processing:

```
fifo2_fd = open(FIFO2_PATH, O_RDONLY);
// Sleep for 10 seconds
sleep(10);
```

Listing 2: Child 2 FIFO opening

2.1.2 The Extra Second Delay in Child 1

The first child process is configured with an extra 1-second delay (11 seconds total instead of 10). This additional delay is crucial to prevent a race condition:

- Both the daemon (parent) and Child 1 need to access FIFO2 in write mode
- Child 2 needs to access FIFO2 in read mode
- If Child 1 finishes processing and tries to write to FIFO2 before Child 2 has opened it for reading, Child 1 would block
- If Child 2 completes its task early and exits, there would be no reader for FIFO2, causing Child 1 to enter a timeout state

The extra delay in Child 1 ensures that Child 2 has sufficient time to open FIFO2 for reading before Child 1 attempts to write to it. This strategic timing prevents potential deadlocks in the FIFO communication sequence.

2.2 Parent Process Implementation

The parent process is responsible for:

- Creating two FIFOs for communication
- Converting itself into a daemon process
- Sending two integer values to the first FIFO
- Creating two child processes
- Setting up signal handlers for various signals
- Managing child process termination through the SIGCHLD handler
- Monitoring and logging system activities

2.2.1 Creating and Managing FIFOs

The parent process creates two FIFOs for interprocess communication:

- The first FIFO (FIFO1_PATH) is used to send two integers from the parent to the first child.
- The second FIFO (FIFO2_PATH) is used to send the larger integer from the first child to the second child.

```
void create_fifos() {
    cleanup_fifos();

    if (mkfifo(FIFO1_PATH, 0666) == -1) {
        log_message("mkfifo for FIFO1 failed");
        exit(EXIT_FAILURE);
    }

    if (mkfifo(FIFO2_PATH, 0666) == -1) {
        log_message("mkfifo for FIFO2 failed");
        unlink(FIFO1_PATH);
        exit(EXIT_FAILURE);
    }

    log_message("FIFOs created successfully");
}
```

Listing 3: FIFO creation function

2.2.2 Daemon Process Creation

The parent process is converted into a daemon using the following steps:

1. Fork twice to detach from terminal
2. Change working directory to root
3. Reset file creation mask
4. Close all file descriptors
5. Redirect standard input/output/error

```
void become_daemon() {
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        const char *error_msg = "First fork failed\n";
        write(STDERR_FILENO, error_msg, strlen(error_msg));
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        exit(EXIT_SUCCESS);
```

```
    }

    if (setsid() < 0) {
        const char *error_msg = "setsid failed\n";
        write(STDERR_FILENO, error_msg, strlen(error_msg));
        exit(EXIT_FAILURE);
    }

    pid = fork();
    if (pid < 0) {
        const char *error_msg = "Second fork failed\n";
        write(STDERR_FILENO, error_msg, strlen(error_msg));
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        exit(EXIT_SUCCESS);
    }

    chdir("/");
    umask(0);

    // Close file descriptors and redirect I/O
    // ...
}
```

Listing 4: Daemon process creation

2.3 First Child Process Implementation

The first child process:

1. Opens the first FIFO for reading
2. Sleeps for 11 seconds (10 + 1 extra for synchronization)
3. Reads two integers from FIFO1
4. Determines the larger number
5. Opens the second FIFO for writing
6. Writes the larger number to FIFO2

```
void child1_process() {
    int fifo1_fd, fifo2_fd;
    int num1, num2, larger_num;

    // Open first FIFO for reading
    fifo1_fd = open(FIFO1_PATH, O_RDONLY);

    // Sleep for 10 seconds as required + 1 second extra for synchronization
    sleep(11);

    if (fifo1_fd == -1) {
        exit(EXIT_FAILURE);
    }

    // Read two integers from FIFO1
    if (read(fifo1_fd, &num1, sizeof(num1)) != sizeof(num1) ||
        read(fifo1_fd, &num2, sizeof(num2)) != sizeof(num2)) {
        close(fifo1_fd);
        exit(EXIT_FAILURE);
    }

    close(fifo1_fd);

    // Determine the larger number
    larger_num = (num1 > num2) ? num1 : num2;

    // Open second FIFO for writing
    fifo2_fd = open(FIFO2_PATH, O_WRONLY);
    if (fifo2_fd == -1) {
```

```
        exit(EXIT_FAILURE);
    }

    // Write the larger number to FIFO2
    if (write(fifo2_fd, &larger_num, sizeof(larger_num)) != sizeof(larger_num)) {
        close(fifo2_fd);
        exit(EXIT_FAILURE);
    }

    close(fifo2_fd);
    exit(EXIT_SUCCESS);
}
```

Listing 5: First child process implementation

2.4 Second Child Process Implementation

The second child process:

1. Opens the second FIFO for reading
2. Sleeps for 10 seconds
3. Reads the command and larger number from FIFO2
4. Prints the larger number to the screen

```
void child2_process() {
    int fifo2_fd;
    int larger_num;
    char output_buffer[100];
    int stdout_fd = STDOUT_FILENO;

    // Open second FIFO for reading
    fifo2_fd = open(FIFO2_PATH, O_RDONLY);

    // Sleep for 10 seconds
    sleep(10);

    if (fifo2_fd == -1) {
        exit(EXIT_FAILURE);
    }

    // Read the command from FIFO2
    char command_buffer[100];
    if (read(fifo2_fd, command_buffer, sizeof(command_buffer)) <= 0) {
        close(fifo2_fd);
        exit(EXIT_FAILURE);
    }

    // Check if the command is correct
    if (strncmp(command_buffer, COMMAND, strlen(COMMAND)) != 0) {
        close(fifo2_fd);
        exit(EXIT_FAILURE);
    }

    // Read the larger number from FIFO2
    if (read(fifo2_fd, &larger_num, sizeof(larger_num)) != sizeof(larger_num)) {
        close(fifo2_fd);
        exit(EXIT_FAILURE);
    }

    close(fifo2_fd);

    // Print the larger number to stdout
    snprintf(output_buffer, sizeof(output_buffer), "The larger number is: %d\n",
    larger_num);
    write(stdout_fd, output_buffer, strlen(output_buffer));

    result = larger_num;
    exit(EXIT_SUCCESS);
```

```
}
```

Listing 6: Second child process implementation

2.5 Signal Handling

The system handles various signals for process management:

```
void signal_handler(int sig) {
    pid_t pid;
    int status;
    char message[256];
    time_t now;

    switch (sig) {
        case SIGCHLD:
            while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
                snprintf(message, sizeof(message), "Process ID of exited child: [PID : %d]", pid);
                log_message(message);

                if (WIFEXITED(status)) {
                    snprintf(message, sizeof(message), "Child %d exited with status: %d", pid, WEXITSTATUS(status));
                    log_message(message);
                } else if (WIFSIGNALED(status)) {
                    snprintf(message, sizeof(message), "Child %d killed by signal: %d", pid, WTERMSIG(status));
                    log_message(message);
                }

                if (pid == child1_pid) {
                    child1_pid = 0;
                    child_counter += 2;
                    snprintf(message, sizeof(message), "Child 1 terminated, counter now: %d", child_counter);
                    log_message(message);
                } else if (pid == child2_pid) {
                    child2_pid = 0;
                    child_counter += 2;
                    snprintf(message, sizeof(message), "Child 2 terminated, counter now: %d", child_counter);
                    log_message(message);
                }
            }
            break;

        case SIGUSR1:
            // Display status information
            // ...
            break;

        case SIGHUP:
            // Reload configuration
            // ...
            break;

        case SIGTERM:
            // Graceful shutdown
            if (child1_pid > 0) {
                kill(child1_pid, SIGTERM);
                log_message("Sent SIGTERM to child 1");
            }

            if (child2_pid > 0) {
                kill(child2_pid, SIGTERM);
                log_message("Sent SIGTERM to child 2");
            }

            cleanup_fifos();
            log_message("Daemon shutting down");
            daemon_running = 0;
```

```
            break;

        default:
            // Handle unexpected signals
            // ...
            break;
    }
}
```

Listing 7: Signal handler implementation

2.6 Zombie Process Prevention

Zombie processes occur when a child process terminates, but its parent has not yet called `wait()` or `waitpid()` to collect its exit status. The implementation prevents zombie processes through two key mechanisms:

2.6.1 SIGCHLD Handler

The first line of defense against zombie processes is the SIGCHLD signal handler. When a child process terminates, the kernel sends a SIGCHLD signal to the parent. Our handler uses the non-blocking `waitpid()` call with the WNOHANG flag to immediately reap any terminated child processes:

```
case SIGCHLD:
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        snprintf(message, sizeof(message), "Process ID of exited child: [PID : %d]", pid
    );
        log_message(message);

        if (WIFEXITED(status)) {
            snprintf(message, sizeof(message), "Child %d exited with status: %d",
                    pid, WEXITSTATUS(status));
            log_message(message);
        } else if (WIFSIGNALED(status)) {
            snprintf(message, sizeof(message), "Child %d killed by signal: %d",
                    pid, WTERMSIG(status));
            log_message(message);
        }

        // Update bookkeeping information
        if (pid == child1_pid) {
            child1_pid = 0;
            child_counter += 2;
        } else if (pid == child2_pid) {
            child2_pid = 0;
            child_counter += 2;
        }
    }
    break;
```

Listing 8: Zombie prevention with SIGCHLD handler

The use of a `while` loop with `waitpid()` ensures that all terminated child processes are reaped, even if multiple children terminate simultaneously or if signals are coalesced.

2.6.2 Timeout Monitoring with check_child_timeouts

The second mechanism for preventing zombie processes is the `check_child_timeouts()` function, which actively monitors for potentially hung or unresponsive child processes. This function is called periodically from the main loop:

```
void check_child_timeouts() {
    time_t now;
    time(&now);
    char message[100];

    // Check if children are taking too long (more than 20 seconds)
    if (child1_pid > 0 && (now - child1_start) > 20) {
        snprintf(message, sizeof(message), "Child 1 (PID: %d) timed out, terminating",
    child1_pid);
```

```
        log_message(message);
        kill(child1_pid, SIGTERM);
        child1_pid = 0;
    }

    if (child2_pid > 0 && (now - child2_start) > 20) {
        snprintf(message, sizeof(message), "Child 2 (PID: %d) timed out, terminating",
    child2_pid);
        log_message(message);
        kill(child2_pid, SIGTERM);
        child2_pid = 0;
    }
}
```

Listing 9: Active timeout monitoring to prevent hung processes

This function is crucial for preventing situations where child processes might hang (for example, due to deadlocks in FIFO operations) and never terminate naturally. By actively monitoring the runtime of each child process and forcefully terminating those that exceed a threshold (20 seconds in this implementation), we ensure that:

- No child process runs indefinitely, consuming system resources
- Deadlocked processes are detected and terminated
- The parent process maintains control over the execution lifecycle
- The system can recover from unexpected situations

The combination of these two mechanisms—reactive handling with SIGCHLD and proactive monitoring with `check_child_timeouts()`—ensures robust protection against zombie processes and hung child processes, satisfying the bonus requirements of the assignment.

2.7 Timeout Handling

The daemon implements a mechanism to detect and terminate unresponsive child processes as discussed in Section 2.6. This monitoring system runs periodically in the main loop:

```
while (daemon_running && child_counter < 4) {
    char loop_msg[100];
    snprintf(loop_msg, sizeof(loop_msg), "proceeding (%d) child_counter=%d
    daemon_running=%d",
            ++proceeding_count, child_counter, daemon_running);
    log_message(loop_msg);

    check_child_timeouts();

    sleep(2);
}
```

Listing 10: Main loop with timeout checking

3 Test Results

The program was tested with various input values to verify correct operation.

3.1 Test Case 1: Normal Operation with Extra Delay

Input: 34 and 67
Configuration: Child 1 sleeps for 11 seconds (extra 1 second delay)
Expected Output: The larger number is: 67
Results:

- The daemon process started successfully
- FIFOs were created successfully

- Child processes were created and executed their tasks
- The larger number (67) was correctly identified and displayed
- Child processes terminated successfully with exit status 0
- The daemon terminated gracefully

3.2 Test Case 2: Operation Without Extra Delay (Successful Case)

Input: 34 and 67
Configuration: Child 1 sleeps for exactly 10 seconds (no extra delay)
Expected Output: The larger number is: 67

In some runs, the program executed successfully even without the extra delay, demonstrating that in certain timing scenarios, the race condition does not occur.

3.3 Test Case 3: Operation Without Extra Delay (Failure Case)

Input: 34 and 67
Configuration: Child 1 sleeps for exactly 10 seconds (no extra delay)
Expected Output: The larger number is: 67

In other runs without the extra delay, the program failed due to the race condition described earlier, where Child 1 attempted to write to FIFO2 before Child 2 had opened it for reading or after Child 2 had already completed its task.

3.4 Test Case 4: Timeout Handling

To test the timeout handling, I modified the sleep time in Child 2 to exceed the timeout threshold.

Results:

- The daemon detected the unresponsive Child 2
- Child 2 was terminated with a SIGTERM signal
- The system recovered gracefully and continued operation

4 Screenshots

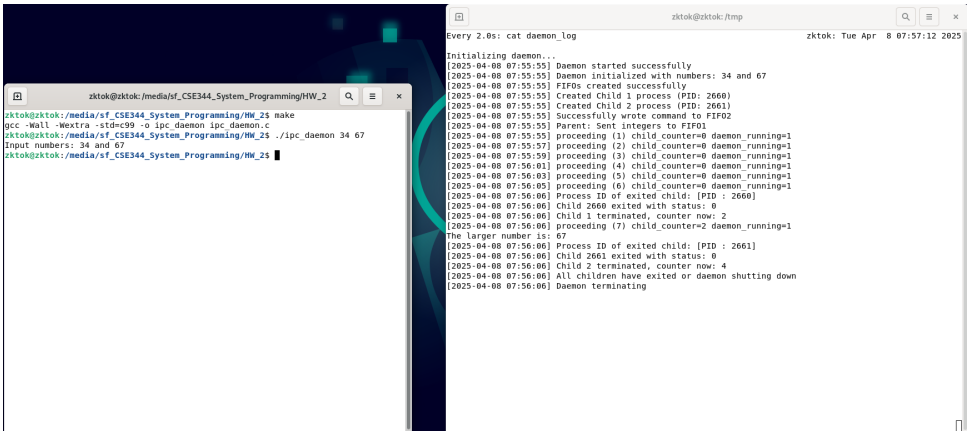

Figure 1: Program execution showing successful run with inputs 34 and 67 with +1 second

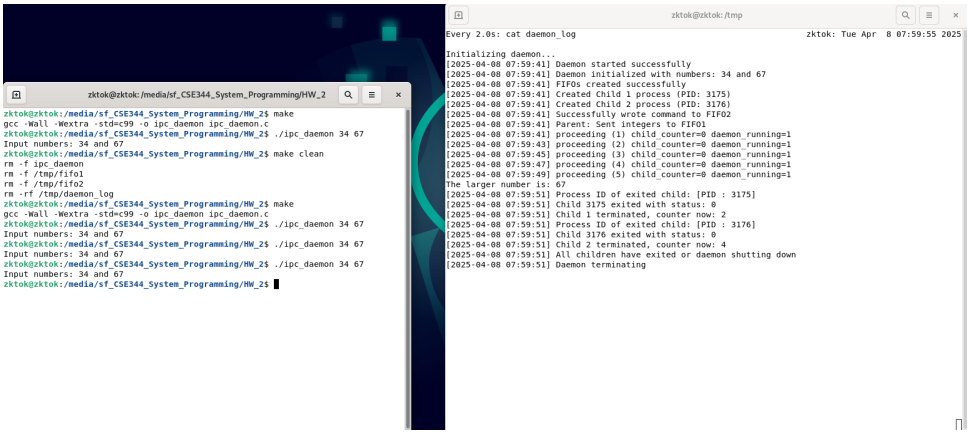

Figure 2: Program execution showing successful run with inputs 34 and 67 exactly 10 second

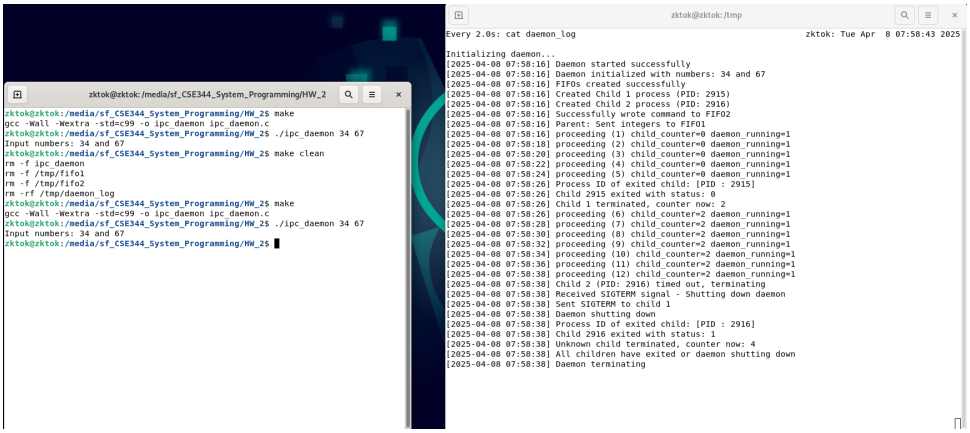

Figure 3: Program execution showing unsuccessful run with inputs 34 and 67 exactly 10 second

5 Conclusion

This implementation successfully fulfills all the requirements specified in the homework assignment. The key challenges were:

1. Managing FIFO synchronization to prevent deadlocks
2. Implementing proper signal handling for child process management
3. Converting the parent process into a daemon with proper logging

The most critical aspect was ensuring proper synchronization between the child processes when accessing the FIFOs. By adding an extra 1-second delay to Child 1, potential race conditions and deadlocks were prevented, ensuring reliable and consistent operation of the IPC system.

The implementation also successfully addresses the bonus requirements by implementing robust zombie process prevention through both reactive (SIGCHLD handler) and proactive (check_child_timeouts) mechanisms. The system carefully tracks and reports the exit status of all processes, providing valuable debugging information and satisfying the second bonus requirement.

Through this project, I gained valuable experience in:

- Interprocess communication using FIFOs
- Daemon process implementation
- Signal handling and process management

- Error handling and recovery strategies
- Preventing zombie processes and resource leaks

All required functionality has been successfully implemented, including the bonus sections for zombie process prevention and exit status reporting.