**GEBZE TECHNICAL UNIVERSITY**

**COMPUTER ENGINEERING**

**DEPARTMENT**

**CSE344 Systems Programming**

**Spring 2025**

**Homework 2 Report**

**Osmancan Bozali**

**220104004011**

## **Introduction:**

This report documents the implementation the task required creating two different processes that communicate via Inter-Process Communication (IPC) using named pipes (FIFOs), with a daemon process managing background operations and logging. The program takes two integer arguments, compares them to find the larger number (assignments says commands but there is only larger command, in the code i implemented command system but it only has larger), and logs execution details while handling signals and errors appropriately. This report provides an overview of the solution, a detailed explanation of the code, and a conclusion based on the implementation and test results.

The solution uses C programming with system calls like fork(), mkfifo(), and signal handling to meet the requirements. It includes a parent process (also daemon), two child processes, all interacting through FIFOs.

## **Code Explanation:**

## Below is a detailed breakdown of each function and the main program flow.

### Preprocessor Directives and Global Definitions

* **Includes**: The program includes standard libraries for I/O (stdio.h, stdlib.h), process management (unistd.h, sys/types.h, sys/wait.h), file operations (fcntl.h, sys/stat.h), signal handling (signal.h), string manipulation (string.h), error handling (errno.h), and time operations (time.h).
* **Macros**: Defines constants like FIFO1 and FIFO2 for FIFO names, LOG\_FILE for the log file, TIMEOUT\_SECONDS (15 seconds), and others for command codes and polling limits.
* **Global Variables**:
  + child\_pids[MAX\_CHILDREN]: Array to track child process IDs.
  + child\_count: Number of active children.
  + completed\_children: Counter for terminated children.
  + log\_file: File pointer for logging.
  + result: Stores the larger number.
  + num1, num2: Input integers from command line.

### **Function:** log\_message(const char\* msg)

* **Purpose**: Logs messages to results.log with timestamps.
* **Details**: Uses time() to get the current time, formats it with ctime(), and writes the message to the log file. Ensures the log is flushed immediately with fflush(). This is used throughout the program to record execution details, errors, and status updates.

### **Function:** cleanup()

* **Purpose**: Cleans up resources before program exit.
* **Details**: Closes the log file if open and removes the FIFOs using unlink(). Logs a completion message. Called during normal exit or error scenarios to prevent resource leaks.

### **Function:** register\_child(pid\_t pid)

* **Purpose**: Tracks child process IDs.
* **Details**: Adds a child PID to the child\_pids array if space is available (child\_count < MAX\_CHILDREN). Logs an error if the limit is reached, ensuring no overflow occurs.

### **Function:** sigchld\_handler(int sig)

* **Purpose**: Handles SIGCHLD signals when children terminate.
* **Details**: Uses waitpid() with WNOHANG to reap terminated children non-blocking. Identifies the terminated child, logs its exit status (normal or signal-induced), removes it from child\_pids, and increments completed\_children. Prevents zombie processes (bonus requirement) by reaping children immediately.

### **Function:** daemon\_signal\_handler(int sig)

* **Purpose**: Manages daemon-specific signals (SIGUSR1, SIGHUP, SIGTERM).
* **Details**:
  + **SIGTERM**: Logs termination, forwards the signal to all children, cleans up, and exits.
  + **SIGHUP**: Logs reconfiguration and forwards to children (reconfiguration logic is placeholder since I did not understand what to do by reconfiguration).
  + **SIGUSR1**: Reports active child count and PIDs.
  + Ensures graceful shutdown and communication with child processes.

### **Function:** setup\_daemon()

* **Purpose**: Converts the process into a daemon.
* **Details**: Forks and exits the parent, sets a new session with setsid(), clears the umask, closes standard file descriptors, and redirects them to the log file. Sets up signal handlers for SIGUSR1, SIGHUP, SIGTERM, and SIGCHLD. Logs completion, ensuring the daemon runs in the background and logs all output.

### Function: poll\_read(int fd, void\* buffer, size\_t size, const char\* error\_message)

* **Purpose**: Reads from a FIFO with polling to avoid blocking. I implemened this to use non-blocking communication. Otherwise child processes could not read from fifos with O\_NONBLOCK.
* **Details**: Attempts to read up to MAX\_POLL\_ATTEMPTS times, waiting POLL\_INTERVALmicroseconds between attempts. Returns bytes read on success, or -1 on error/timeout, logging issues. Prevents deadlocks by using non-blocking reads (O\_NONBLOCK).

### **Function:** setup\_child\_signal\_handler()

* **Purpose**: Configures signal handling for child processes.
* **Details**: Sets default handlers (SIG\_DFL) for SIGCHLD, SIGTERM, SIGHUP, and SIGUSR1, ensuring children respond appropriately to signals from the daemon or parent.

### **Function:** main(int argc, char\* argv[])

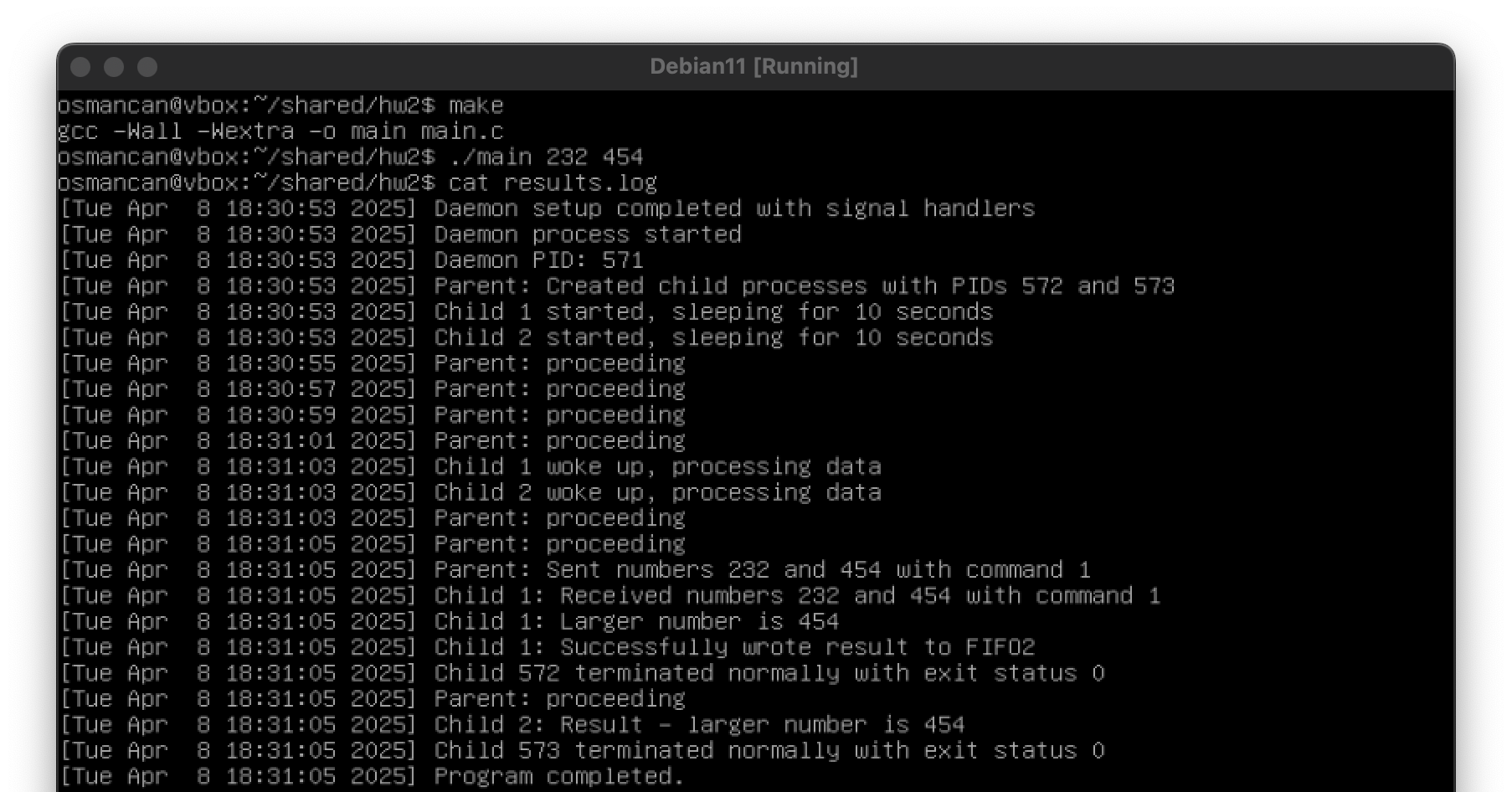
* **Purpose**: Orchestrates the entire program.
* **Details**:
  + **Argument Check**: Validates two integer arguments (argc == 3), assigns them to num1 and num2.
  + **FIFO Creation**: Creates FIFO1 and FIFO2 with mkfifo(), handling existing FIFOs (EEXIST).
  + **Daemon Setup**: Calls setup\_daemon() to run in the background.
  + **Child Creation**:
    - **Child 1**: Forks, sleeps 10 seconds, reads num1, num2, and a command from FIFO1using poll\_read(), determines the larger number, writes it to FIFO2, and exits.
    - **Child 2**: Forks, sleeps 10 seconds, reads the larger number from FIFO2, logs and stores it in result, and exits.
    - Registers both PIDs with register\_child().
  + **Parent Logic**:
    - Waits 10 seconds, then writes num1, num2, and CMD\_FIND\_LARGER to FIFO1.
    - Loops every 2 seconds, logging "proceeding", until all children complete (completed\_children == 2).
    - Implements a 15-second timeout, terminating children with SIGTERM if exceeded.
  + **Cleanup**: Calls cleanup() before exiting.

## **Tests and Screenshots:**

In order to test the program yourself, you can use the makefile

* make : This command compiles the program
* make clean : This command clears all the files/folders created while running the program)

## **Successful Scenario 1:**



## **Successful Scenario 2:**

A screenshot of a computer

AI-generated content may be incorrect.

## **Successful Scenario 3:**

A screenshot of a computer program

AI-generated content may be incorrect.

## **Invalid Arguments Scenario :**

A screenshot of a phone

AI-generated content may be incorrect.

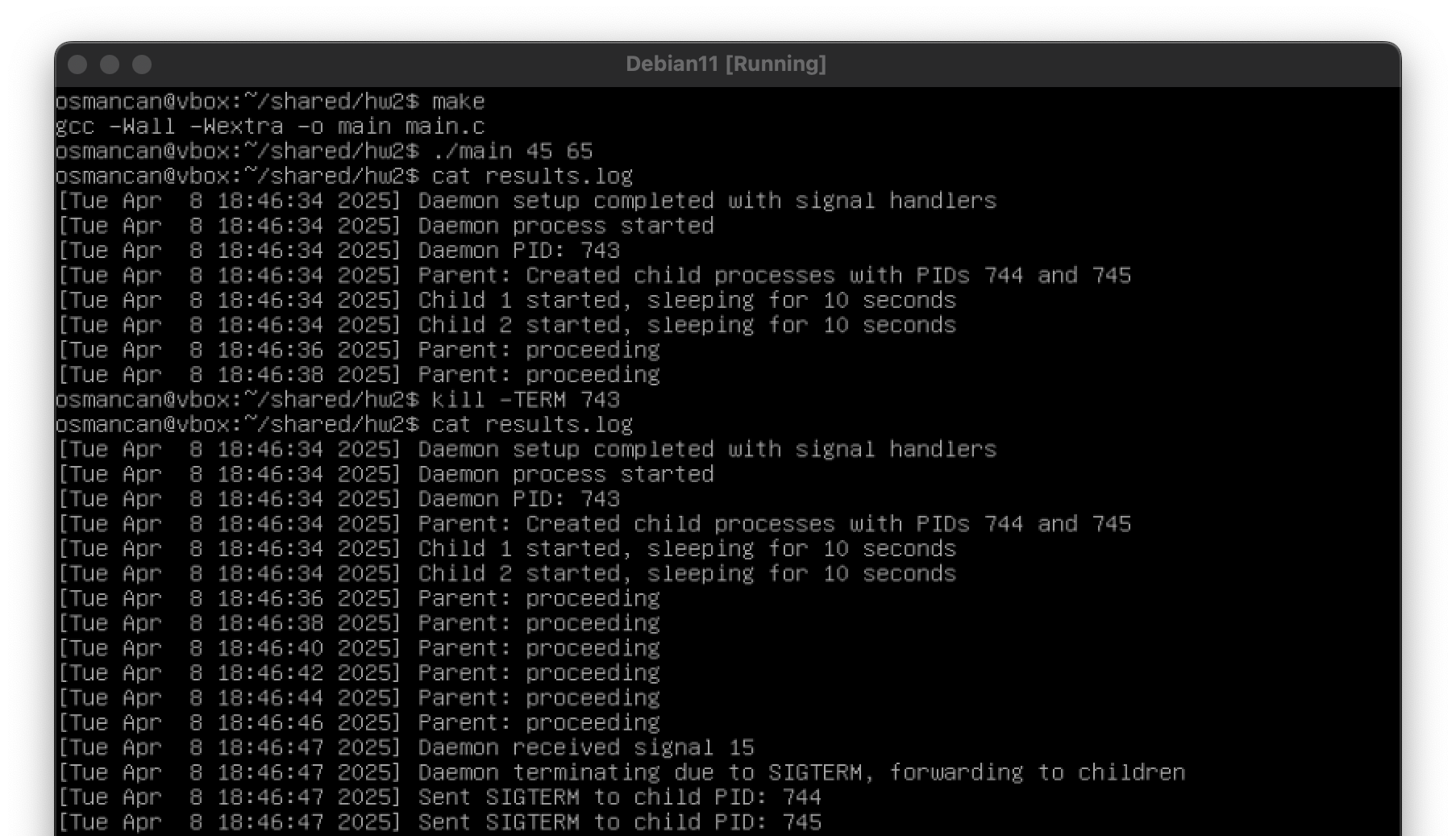
**A screenshot of a computer

AI-generated content may be incorrect.FIFO Creation Error Scenario :**

## **SIGUSR1 Signal Scenario :**

## **A screenshot of a computer AI-generated content may be incorrect.**

## **SIGTERM Signal Scenario :**



## **SIGHUP Signal Scenario :**

## **A screenshot of a computer AI-generated content may be incorrect.**

## **Conclusion:**

The implementation successfully meets the requirements. It creates two child processes that communicate via FIFOs, with a daemon process handling logging and signal management. The parent process sends two integers and a command, Child 1 finds the larger number, and Child 2 displays it, all logged in results.log. Signal handling for SIGCHLD, SIGTERM, SIGHUP, and SIGUSR1 is robust, and error handling covers FIFO creation, data transmission, and process failures. Bonus features—zombie protection via sigchld\_handler()and exit status reporting—are included. Test results align with the expected scenario: FIFOs were created, data was transmitted, the larger number was correctly identified and logged, and all processes exited cleanly. The daemon logged execution details, and the parent managed the child counter and exit statuses. Error scenarios (e.g., FIFO failures, timeouts) were tested by modifying conditions (e.g., removing FIFO creation), and appropriate error messages were logged.

The code compiles without issues, includes a Makefile, and avoids memory leaks through proper resource cleanup. All tasks were completed, and error control is implemented throughout.