KAMİL DURU 200104004064
CSE 344 SYSTEM PROGRAMMING
HOMEWORK #2

1. Introduction

This program is an implemmentation of a Linux-based daemon process that manages inter-process communication using named pipes (FIFOs). The daemon is structured to handle two integer inputs, call child processes, and coordinate data flow through FIFO channels while ensuring handling of system signals, timeouts, and logging.

The main objectives of the program are:

- To convert a standard user process into a background daemon process using the double-fork technique.
- To demonstrate effective inter-process communication through FIFOs.
- To utilize signal handling for process management and system robustness.

The daemon forks two child processes: the first reads integers from a FIFO, compares them, and writes the result to another FIFO. The second child reads a command and then the result, printing the final output to a log. Logging is handled in daemon process, allowing the process's internal state, operations, and any detected errors to be recorded consistently.

2. Code Explanation

`void timeout_handler()`

Handles the SIGALRM signal, which is triggered when the timeout occurs.

- Writes a message to the log file indicating a timeout has occurred.
- Uses kill(0, SIGTERM) to send a termination signal to all processes in the same process group.
- Sets the running flag to 0 to exit the main loop.

`void sigchld_handler()`

Handles the SIGCHLD signal triggered when a child process terminates.

- Uses waitpid() with WNOHANG to reap all finished child processes.
- Logs whether the child exited normally or via a signal.
- Increments child_count each time a child terminates.
- Stops the main loop when all children have finished.

`void signal_handler(int sig)`

Generic handler for SIGUSR1, SIGHUP, and SIGTERM.

- Logs which signal was received.
- If SIGTERM is received, sets the running flag to 0 to initiate shutdown.

`void clean_data()`

Performs cleanup before exiting.

- Removes all FIFO files (FIFO1, FIFO2, FIFO3) using unlink().
- Logs a final message and closes the log file (fd_log) if it's open.

`void create_daemon()`

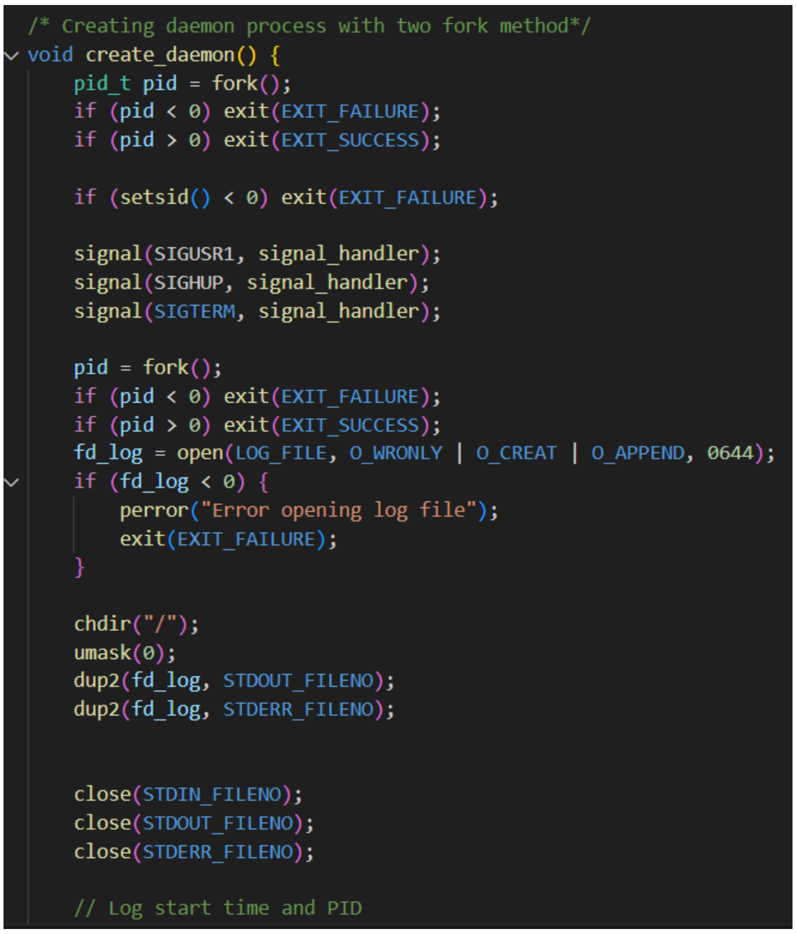
Converts the process into a background daemon using the double-fork method.

- First fork detaches from the parent shell.
- setsid() creates a new session.
- Second fork ensures the daemon can’t reacquire a terminal.
- Registers signal handlers for relevant signals.
- Opens a log file and redirects standard output and error to the log.
- Closes standard input/output/error to fully detach from the terminal.

- Logs the daemon's start time and PID.

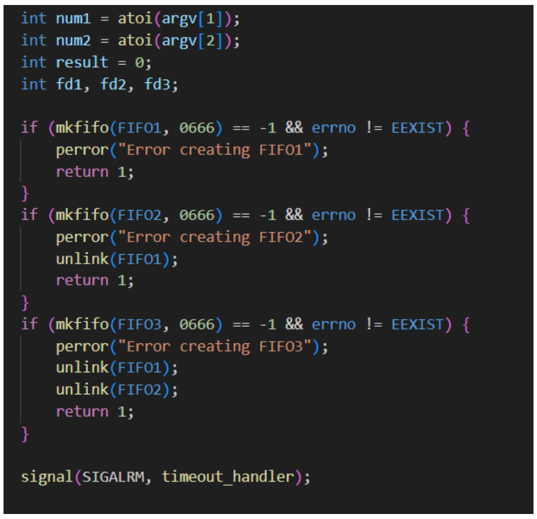
Mandatory Section

Daemon process starts immidiately at the beginning of the program.



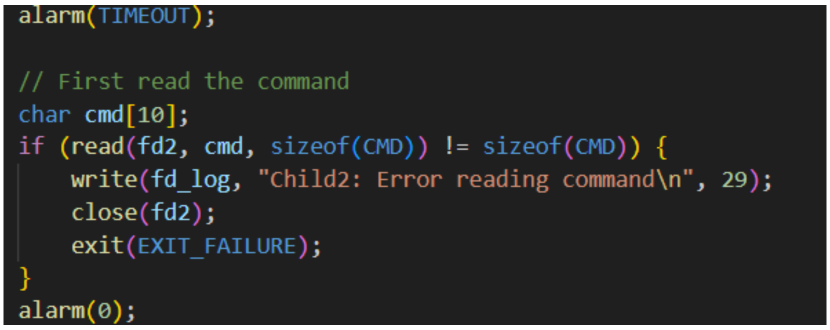
```c
/* Creating daemon process with two fork method*/
void create_daemon() {
    pid_t pid = fork();
    if (pid < 0) exit(EXIT_FAILURE);
    if (pid > 0) exit(EXIT_SUCCESS);

    if (setsid() < 0) exit(EXIT_FAILURE);

    signal(SIGUSR1, signal_handler);
    signal(SIGHUP, signal_handler);
    signal(SIGTERM, signal_handler);

    pid = fork();
    if (pid < 0) exit(EXIT_FAILURE);
    if (pid > 0) exit(EXIT_SUCCESS);
    fd_log = open(LOG_FILE, O_WRONLY | O_CREAT | O_APPEND, 0644);
    if (fd_log < 0) {
        perror("Error opening log file");
        exit(EXIT_FAILURE);
    }

    chdir("/");
    umask(0);
    dup2(fd_log, STDOUT_FILENO);
    dup2(fd_log, STDERR_FILENO);


    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);

    // Log start time and PID
```

Created daemon process with two forks method. It handles SIGUSR1, SIGHUP and SIGTERM signals. Creates log file to log all operations, errors etc. Changes directory to root to avoid problems in system maintanance. Redirects output and error streams to log file.



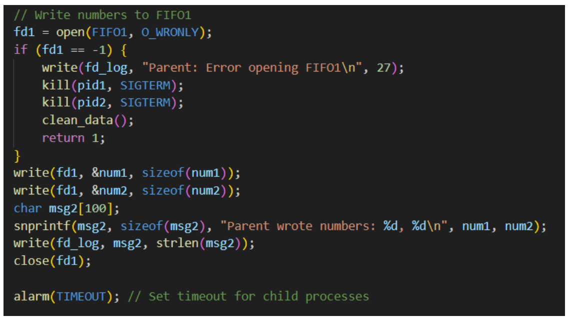
```c
int num1 = atoi(argv[1]);
int num2 = atoi(argv[2]);
int result = 0;
int fd1, fd2, fd3;

if (mkfifo(FIFO1, 0666) == -1 && errno != EEXIST) {
    perror("Error creating FIFO1");
    return 1;
}
if (mkfifo(FIFO2, 0666) == -1 && errno != EEXIST) {
    perror("Error creating FIFO2");
    unlink(FIFO1);
    return 1;
}
if (mkfifo(FIFO3, 0666) == -1 && errno != EEXIST) {
    perror("Error creating FIFO3");
    unlink(FIFO1);
    unlink(FIFO2);
    return 1;
}

signal(SIGALRM, timeout_handler);
```

Program turns number arguments to integer, create three FIFOs and sets signal function to catch SIGALRM that will handle timeout mechanism.

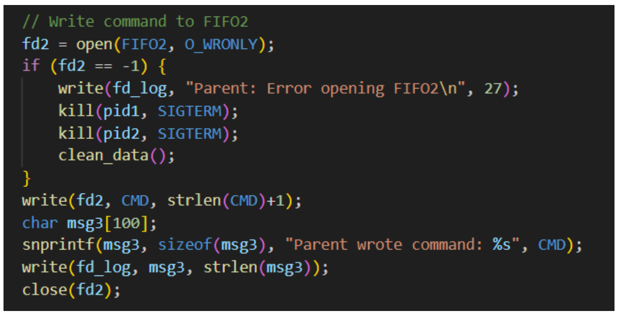


```c
alarm(TIMEOUT);

// First read the command
char cmd[10];
if (read(fd2, cmd, sizeof(CMD)) != sizeof(CMD)) {
    write(fd_log, "Child2: Error reading command\n", 29);
    close(fd2);
    exit(EXIT_FAILURE);
}
alarm(0);
```

Timeout mechanism implemented with alarm function. Parent sets alarm for child processes and child process starts the timeout while reading from FIFO. If child process get blocked while reading, alarm function will send SIGALRM signal when time is up and all child processes will terminate.

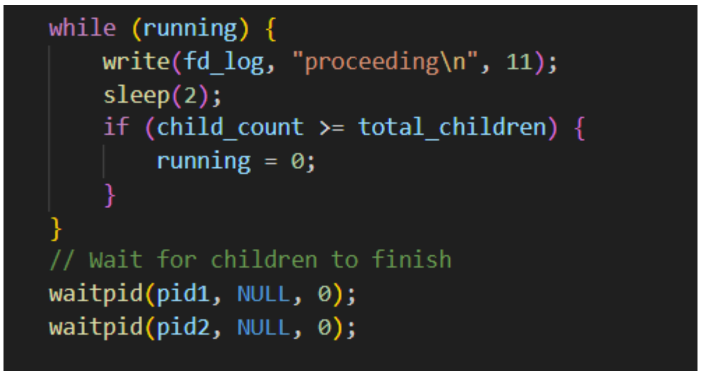


```c
// Write numbers to FIFO1
fd1 = open(FIFO1, O_WRONLY);
if (fd1 == -1) {
    write(fd_log, "Parent: Error opening FIFO1\n", 27);
    kill(pid1, SIGTERM);
    kill(pid2, SIGTERM);
    clean_data();
    return 1;
}
write(fd1, &num1, sizeof(num1));
write(fd1, &num2, sizeof(num2));
char msg2[100];
snprintf(msg2, sizeof(msg2), "Parent wrote numbers: %d, %d\n", num1, num2);
write(fd_log, msg2, strlen(msg2));
close(fd1);

alarm(TIMEOUT); // Set timeout for child processes
```



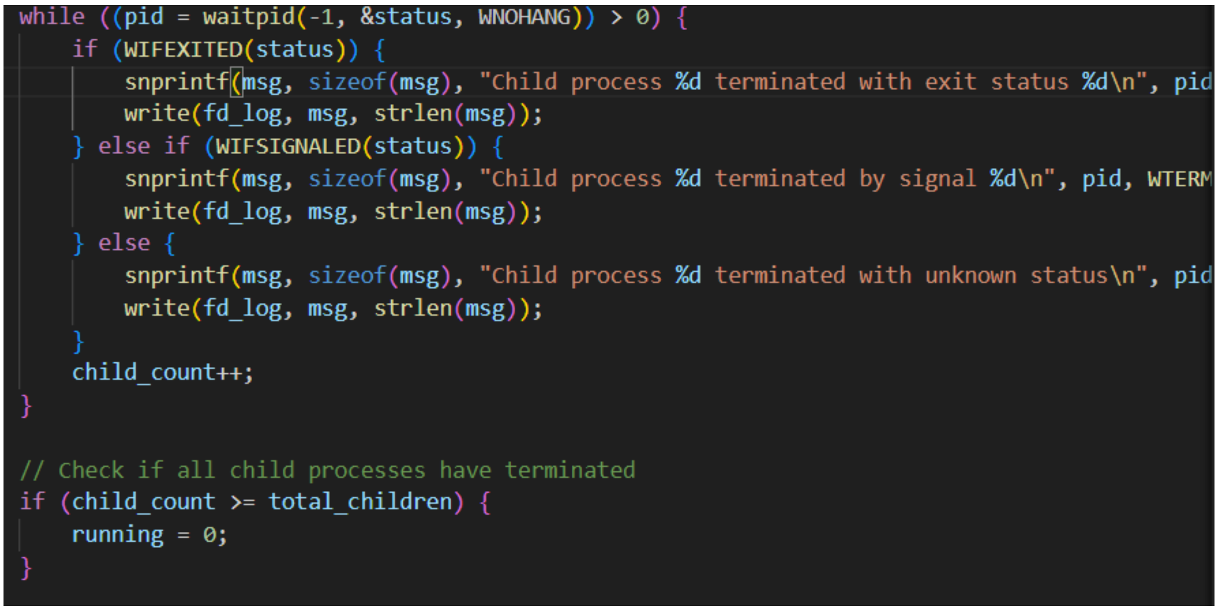
```c
// Write command to FIFO2
fd2 = open(FIFO2, O_WRONLY);
if (fd2 == -1) {
    write(fd_log, "Parent: Error opening FIFO2\n", 27);
    kill(pid1, SIGTERM);
    kill(pid2, SIGTERM);
    clean_data();
}
write(fd2, CMD, strlen(CMD)+1);
char msg3[100];
snprintf(msg3, sizeof(msg3), "Parent wrote command: %s", CMD);
write(fd_log, msg3, strlen(msg3));
close(fd2);
```

Parent process writes number to first FIFO and command “COMPARE” to second FIFO.



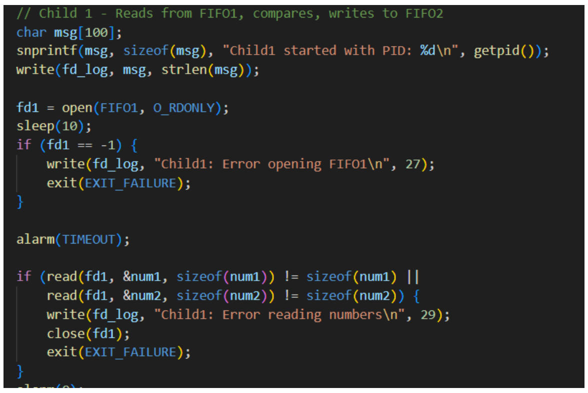
```c
while (running) {
    write(fd_log, "proceeding\n", 11);
    sleep(2);
    if (child_count >= total_children) {
        running = 0;
    }
}
// Wait for children to finish
waitpid(pid1, NULL, 0);
waitpid(pid2, NULL, 0);
```

After sending information to FIFOs, parent process sets sigchld_handler to handle child process termination. Enters a while loop until all child processes terminate.

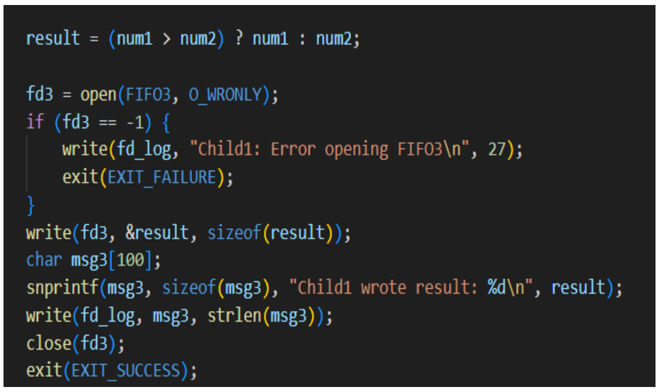


```c
while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
    if (WIFEXITED(status)) {
        snprintf(msg, sizeof(msg), "Child process %d terminated with exit status %d\n", pid
        write(fd_log, msg, strlen(msg));
    } else if (WIFSIGNALED(status)) {
        snprintf(msg, sizeof(msg), "Child process %d terminated by signal %d\n", pid, WTERM
        write(fd_log, msg, strlen(msg));
    } else {
        snprintf(msg, sizeof(msg), "Child process %d terminated with unknown status\n", pid
        write(fd_log, msg, strlen(msg));
    }
    child_count++;
}

// Check if all child processes have terminated
if (child_count >= total_children) {
    running = 0;
}
```

This loop uses waitpid() with WNOHANG to non-blockingly check if any child processes have terminated. It logs whether the child exited normally, was killed by a signal, or had an unknown termination status. After each child finishes, child_count is incremented, and once all expected children have exited, the running flag is set to 0 to stop the main loop.

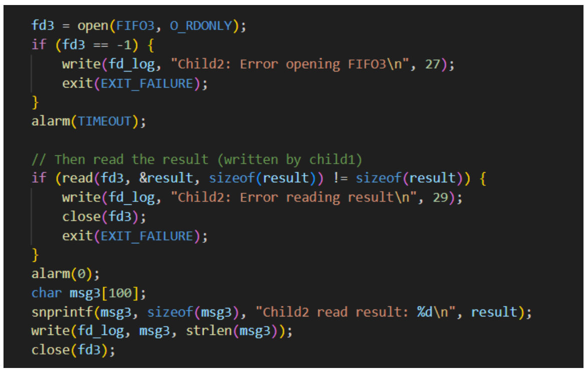


```c
// Child 1 - Reads from FIFO1, compares, writes to FIFO2
char msg[100];
snprintf(msg, sizeof(msg), "Child1 started with PID: %d\n", getpid());
write(fd_log, msg, strlen(msg));

fd1 = open(FIFO1, O_RDONLY);
sleep(10);
if (fd1 == -1) {
    write(fd_log, "Child1: Error opening FIFO1\n", 27);
    exit(EXIT_FAILURE);
}

alarm(TIMEOUT);

if (read(fd1, &num1, sizeof(num1)) != sizeof(num1) ||
    read(fd1, &num2, sizeof(num2)) != sizeof(num2)) {
    write(fd_log, "Child1: Error reading numbers\n", 29);
    close(fd1);
    exit(EXIT_FAILURE);
}
```

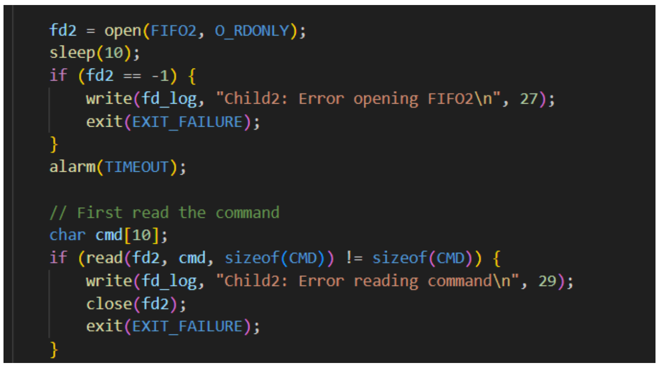


```c
result = (num1 > num2) ? num1 : num2;

fd3 = open(FIFO3, O_WRONLY);
if (fd3 == -1) {
    write(fd_log, "Child1: Error opening FIFO3\n", 27);
    exit(EXIT_FAILURE);
}
write(fd3, &result, sizeof(result));
char msg3[100];
snprintf(msg3, sizeof(msg3), "Child1 wrote result: %d\n", result);
write(fd_log, msg3, strlen(msg3));
close(fd3);
exit(EXIT_SUCCESS);
```

Child 1 reads numbers from FIFO1 and determines the larger number. Writes the larger number to FIFO3.

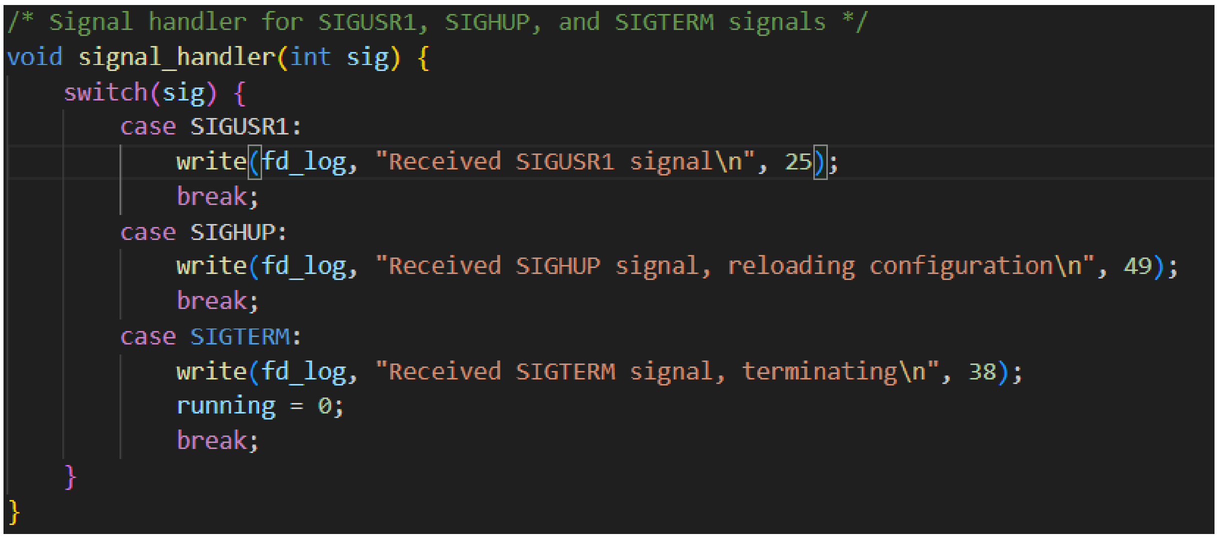


```
fd3 = open(FIFO3, O_RDONLY);
if (fd3 == -1) {
    write(fd_log, "Child2: Error opening FIFO3\n", 27);
    exit(EXIT_FAILURE);
}
alarm(TIMEOUT);

// Then read the result (written by child1)
if (read(fd3, &result, sizeof(result)) != sizeof(result)) {
    write(fd_log, "Child2: Error reading result\n", 29);
    close(fd3);
    exit(EXIT_FAILURE);
}
alarm(0);
char msg3[100];
snprintf(msg3, sizeof(msg3), "Child2 read result: %d\n", result);
write(fd_log, msg3, strlen(msg3));
close(fd3);
```



```
fd2 = open(FIFO2, O_RDONLY);
sleep(10);
if (fd2 == -1) {
    write(fd_log, "Child2: Error opening FIFO2\n", 27);
    exit(EXIT_FAILURE);
}
alarm(TIMEOUT);

// First read the command
char cmd[10];
if (read(fd2, cmd, sizeof(CMD)) != sizeof(CMD)) {
    write(fd_log, "Child2: Error reading command\n", 29);
    close(fd2);
    exit(EXIT_FAILURE);
}
```

Child 2 reads command from FIFO2 and larger number from FIFO3. Finally writes the larger number to log file.

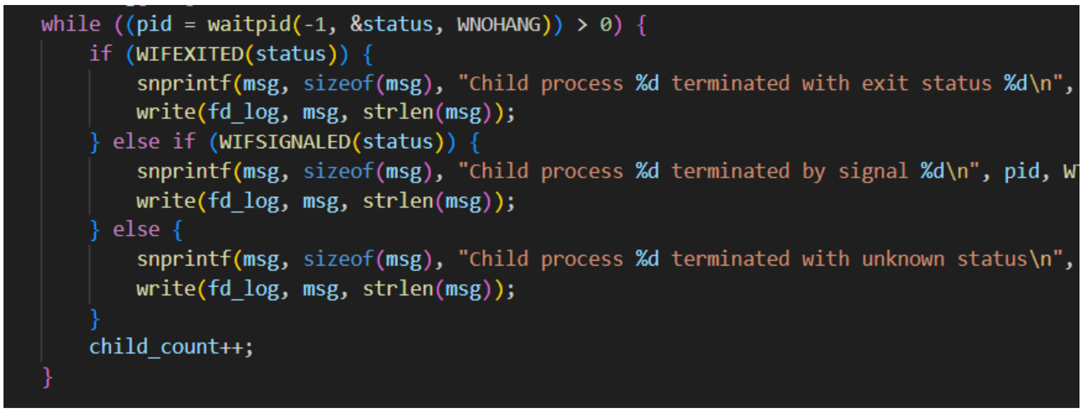


```
/* Signal handler for SIGUSR1, SIGHUP, and SIGTERM signals */
void signal_handler(int sig) {
    switch(sig) {
        case SIGUSR1:
            write(fd_log, "Received SIGUSR1 signal\n", 25);
            break;
        case SIGHUP:
            write(fd_log, "Received SIGHUP signal, reloading configuration\n", 49);
            break;
        case SIGTERM:
            write(fd_log, "Received SIGTERM signal, terminating\n", 38);
            running = 0;
            break;
    }
}
```

Daemon process handles SIGTERM signal with assigning running flag to 0. SIGUSR1 and SIGHUP signals just writes a message to log.

Bonus Section

Zombie Protection and Exit Status.

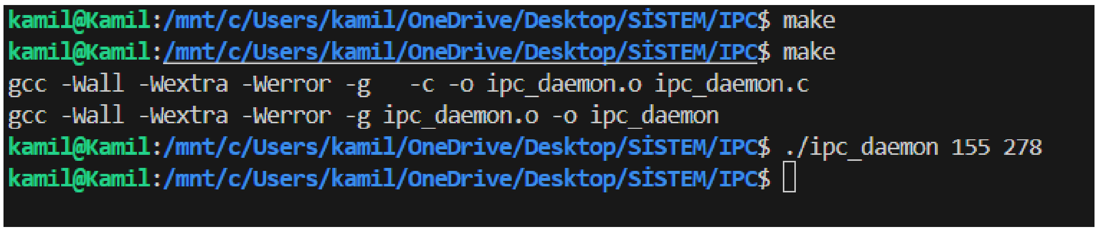


```c
while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
    if (WIFEXITED(status)) {
        snprintf(msg, sizeof(msg), "Child process %d terminated with exit status %d\n",
        write(fd_log, msg, strlen(msg));
    } else if (WIFSIGNALED(status)) {
        snprintf(msg, sizeof(msg), "Child process %d terminated by signal %d\n", pid, W
        write(fd_log, msg, strlen(msg));
    } else {
        snprintf(msg, sizeof(msg), "Child process %d terminated with unknown status\n",
        write(fd_log, msg, strlen(msg));
    }
    child_count++;
}
```

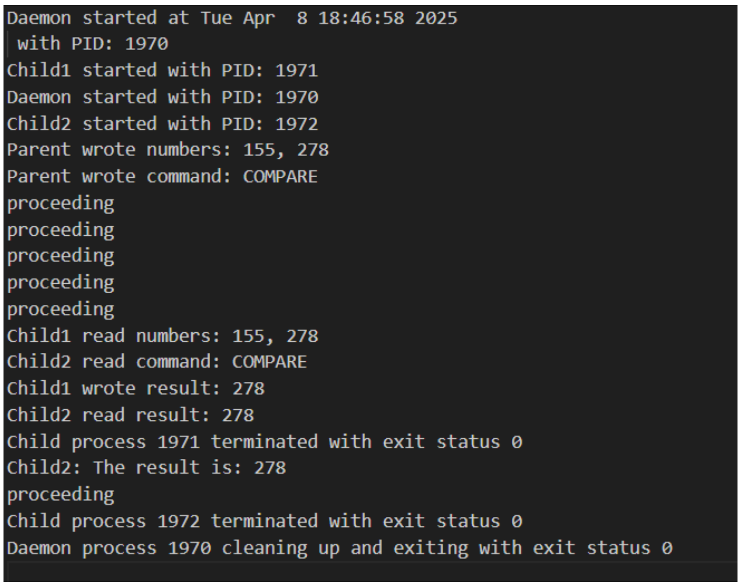
When a child process terminates, it becomes a zombie until the parent reads its exit status using wait() or waitpid(). This loop reaps all dead children immidiately, preventing accumulation of zombie processes.

With WEEXITSTATUS function, program prints the exit status of terminating child process.

Testing



```
kamil@Kamil:/mnt/c/Users/kamil/OneDrive/Desktop/SİSTEM/IPC$ make
kamil@Kamil:/mnt/c/Users/kamil/OneDrive/Desktop/SİSTEM/IPC$ make
gcc -Wall -Wextra -Werror -g   -c -o ipc_daemon.o ipc_daemon.c
gcc -Wall -Wextra -Werror -g ipc_daemon.o -o ipc_daemon
kamil@Kamil:/mnt/c/Users/kamil/OneDrive/Desktop/SİSTEM/IPC$ ./ipc_daemon 155 278
kamil@Kamil:/mnt/c/Users/kamil/OneDrive/Desktop/SİSTEM/IPC$
```



```
Daemon started at Tue Apr  8 18:46:58 2025
 with PID: 1970
Child1 started with PID: 1971
Daemon started with PID: 1970
Child2 started with PID: 1972
Parent wrote numbers: 155, 278
Parent wrote command: COMPARE
proceeding
proceeding
proceeding
proceeding
proceeding
Child1 read numbers: 155, 278
Child2 read command: COMPARE
Child1 wrote result: 278
Child2 read result: 278
Child process 1971 terminated with exit status 0
Child2: The result is: 278
proceeding
Child process 1972 terminated with exit status 0
Daemon process 1970 cleaning up and exiting with exit status 0
```

Conclusion

Challenges Faced

Developing this daemon-based system presented several challenges, especially around process management and inter-process communication (IPC). Handling timeouts required careful synchronization to avoid deadlocks or missed signals. Additionally, ensuring proper cleanup and preventing zombie processes through signal handling added complexity to the signal management logic.

Final Thoughts

Overall, this project demonstrated the essential components of building a daemon in a UNIX environment, including forking, detaching from terminals, logging, and using FIFOs for communication. The system successfully launches child processes, handles timeouts, logs activities, and gracefully shuts down after task completion. This experience deepened the understanding of daemon architecture and the importance of careful process control, making it a valuable learning exercise in system programming.