CSE 344

Homework #2

200104004074

Bedirhan Ömer Aksoy

1) Introduction

In my implementation, the program takes two arguments from the command line, creates fifos, then daemonizes the program and sets a signal handler for child termination, then forks the program and creates the first child, then creates the second child, then writes the argument numbers into the first fifo, then enters a while loop to wait the childs, then logs the exit statuses of the childs and exits. I have implemented the daemonizing part at the beginning of the program after creating fifos as we are allowed to which has been announced in the teams page.

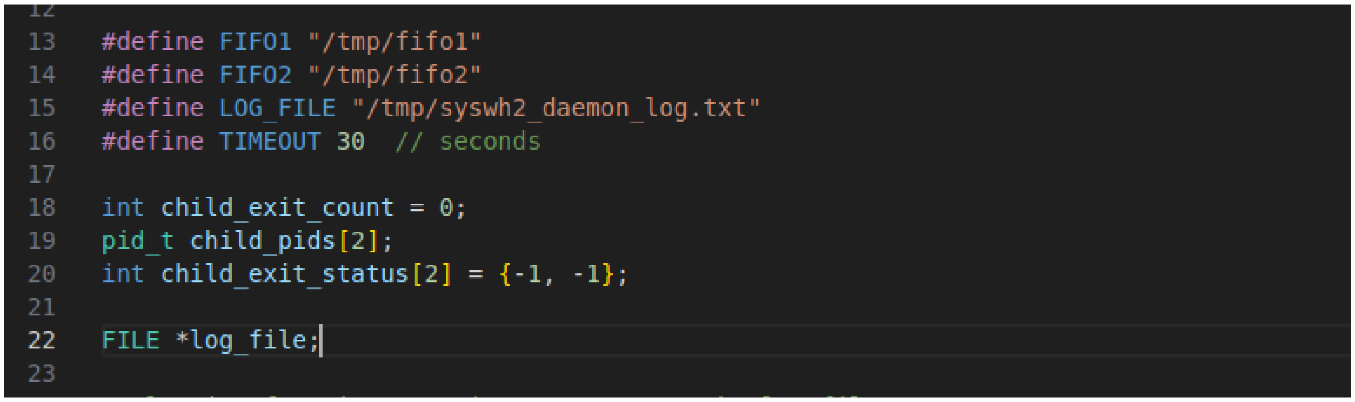
First child process sleeps for 5 seconds to allow the parent to create the second child and to write the input numbers to the first fifo. After 5 second sleep, the first fifo reads the numbers with using usleep to avoid busy waiting, it's technically still a busy wait since we check the read_bytes variable repeatedly, but it's better than checking it in a while loop without sleep. After the first child reads the numbers, it closes the first fifo and determines the maximum of two numbers, and writes it into the second fifo and closes the second fifo and exits.

Second child process sleeps for 10 seconds to allow the first child process and parent child process to write and read to first and second fifo's to prevent race conditions between parent and child processes. After 10 seconds, opens the second fifo to read the maximum number while using usleep to avoid busy waiting (that I mentioned in the first child process), then closes the second fifo, then prints and logs the maximum number message and exits.

The daemonizer function (become_daemon()) first forks the program, then sets the SIGTERM and SIGHUP signals' handlers, then becomes the leader of the session with setsid(), then forks again and terminates the session leader program to make daemon program as a daemon, then sets the signal handlers for SIGTERM and SIGHUP, then forks the program again to prevent daemon process to killed by a terminal signal, then parent process exits, then re-sets the SIGCHILD handler to ensure that the SIGCHILD handler still has file descriptor to log exited childs, then changes working directory to the root, then clears the file mode creation mask with umask(0), then it closes the inherited file descriptors EXCEPT the log file descriptor, then redirects stdin, stdout, stderr to log file descriptor.

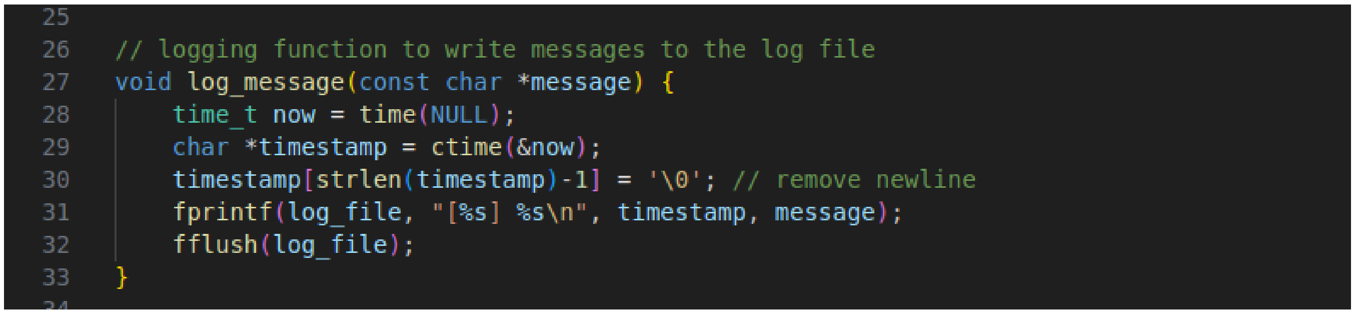
2) Code Explanation

2.0) global variables: FIFO1, FIFO2, LOG_FILE and TIMEOUT are defined variables so that I can use and change them dynamically in my functions.
int child_exit_count is the variable to check the exited childs count for the while loop's condition in the parent process where I wait for both childs to exit or kill them after the timeout.
pid_t child_pids[2] is the pid array for my child processes that so that I can kill the child processes if they don't exit properly with a kill signal and I can use this array to log messages with their PID's into the log file.
int child_exit_status[2] is the int array for the child processes exit statuses to log the exit status of the child processes into the log file.
FILE *log_file is the file pointer for the log file to log actions happening in processes.



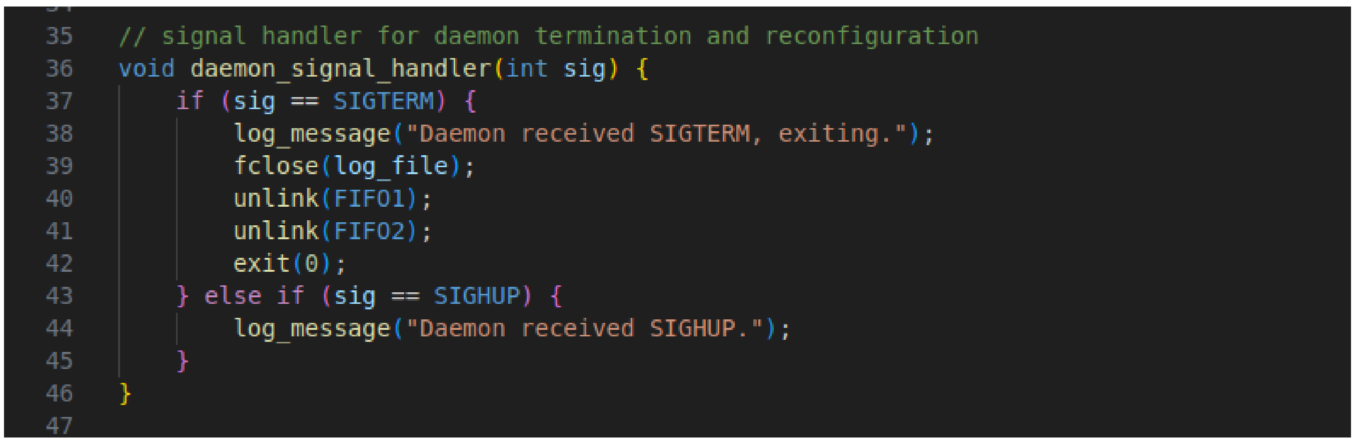
```c
#define FIFO1 "/tmp/fifo1"
#define FIFO2 "/tmp/fifo2"
#define LOG_FILE "/tmp/syswh2_daemon_log.txt"
#define TIMEOUT 30  // seconds

int child_exit_count = 0;
pid_t child_pids[2];
int child_exit_status[2] = {-1, -1};

FILE *log_file;
```

2.1) void log_message(const char *message): Logger function to write fifo read write operations and proceeding message into the "/tmp/syshw2_daemon_log.txt" log file, function adds timestamp, removes the user entered newline, then writes the logs into the log file.



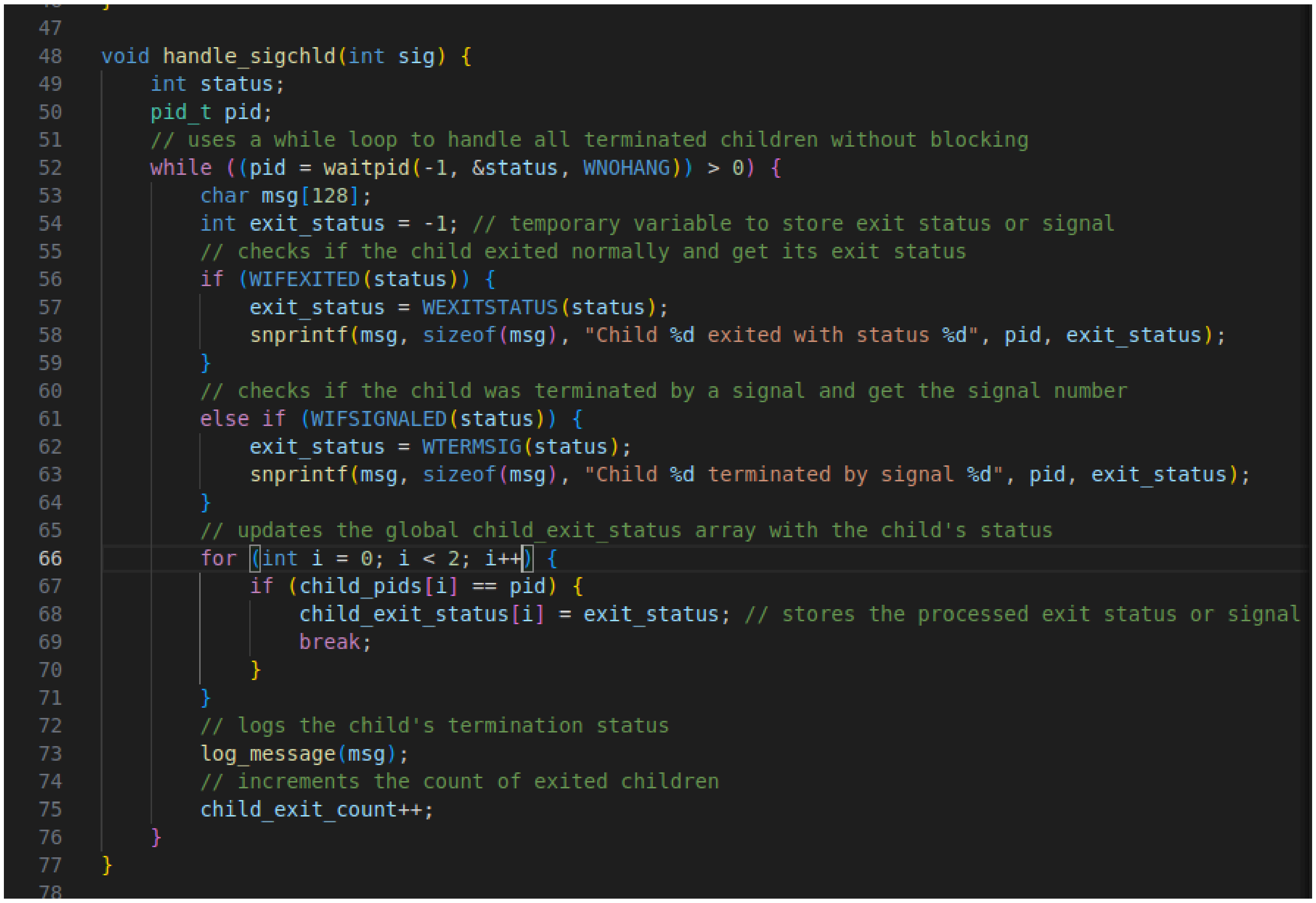
```c
// logging function to write messages to the log file
void log_message(const char *message) {
    time_t now = time(NULL);
    char *timestamp = ctime(&now);
    timestamp[strlen(timestamp)-1] = '\0'; // remove newline
    fprintf(log_file, "[%s] %s\n", timestamp, message);
    fflush(log_file);
}
```

2.2) void daemon_signal_handler(int sig): Signal handler for SIGTERM and SIGHUP signals for daemonized program. If the input signal is SIGTERM, logs signal received message, closes log file descriptor, unlinks fifos, then exits the program. If the input signal is SIGHUP, logs signal received message.



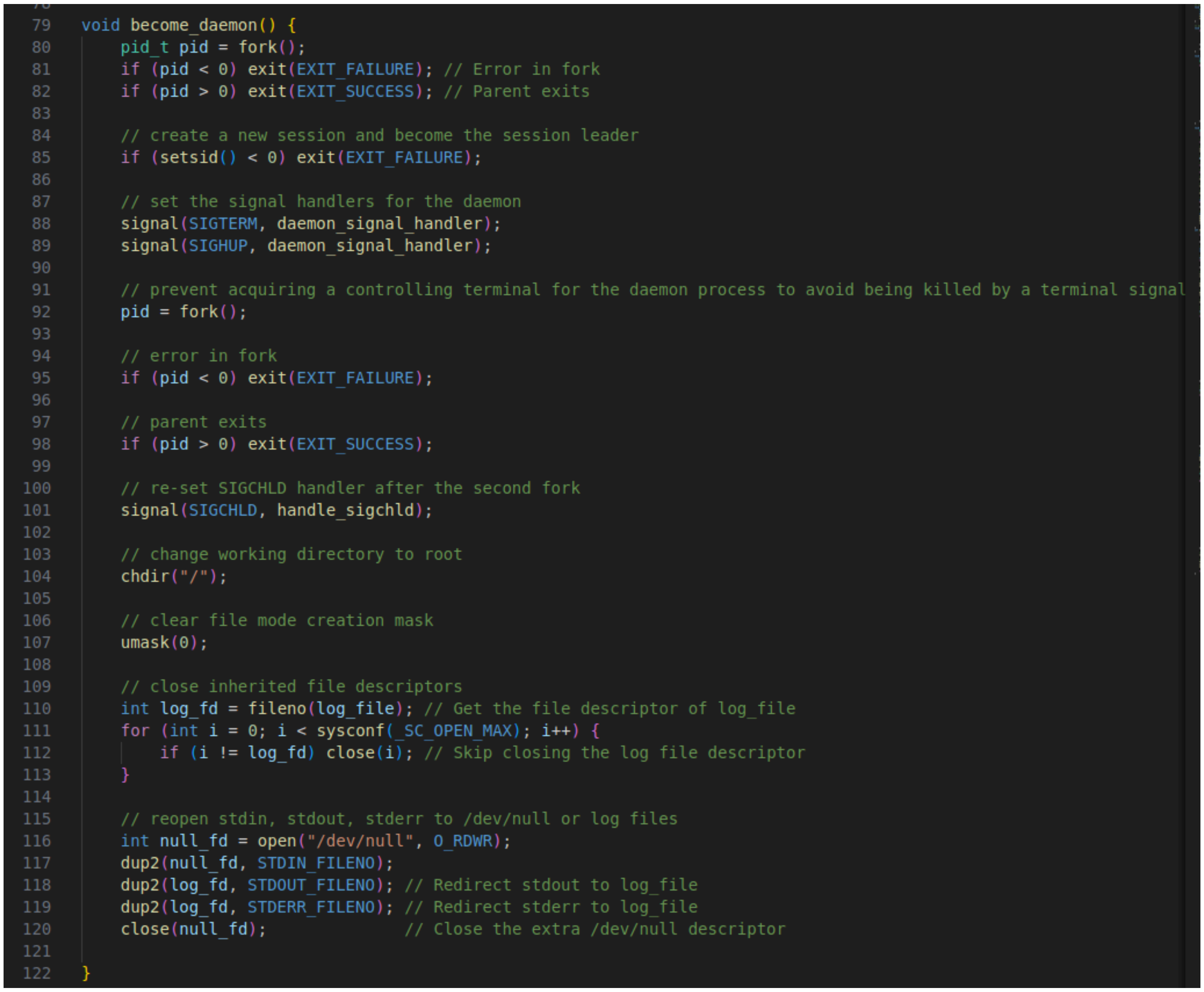
```c
// signal handler for daemon termination and reconfiguration
void daemon_signal_handler(int sig) {
    if (sig == SIGTERM) {
        log_message("Daemon received SIGTERM, exiting.");
        fclose(log_file);
        unlink(FIFO1);
        unlink(FIFO2);
        exit(0);
    } else if (sig == SIGHUP) {
        log_message("Daemon received SIGHUP.");
    }
}
```

2.3) void handle_sigchld(int sig): Waits both terminated childs without blocking with using WNOHANG flag, checks if the childs exited normally with an exit status, if they are, gets the pid and exit status of the child, logs the child pid and exit status to the log file, if the child was terminated by a signal and not exited normally, gets the signal number and logs the child pid and terminate signal to the log file, after logging the exit status or signal to log file, function updates the global child_exit_status array and increases the child_exit_count as we asked to implement to check in the main process in a loop to check if 2 child processes are exited and checking their count in the main.



```c
void handle_sigchld(int sig) {
    int status;
    pid_t pid;
    // uses a while loop to handle all terminated children without blocking
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        char msg[128];
        int exit_status = -1; // temporary variable to store exit status or signal
        // checks if the child exited normally and get its exit status
        if (WIFEXITED(status)) {
            exit_status = WEXITSTATUS(status);
            snprintf(msg, sizeof(msg), "Child %d exited with status %d", pid, exit_status);
        }
        // checks if the child was terminated by a signal and get the signal number
        else if (WIFSIGNALED(status)) {
            exit_status = WTERMSIG(status);
            snprintf(msg, sizeof(msg), "Child %d terminated by signal %d", pid, exit_status);
        }
        // updates the global child_exit_status array with the child's status
        for (int i = 0; i < 2; i++) {
            if (child_pids[i] == pid) {
                child_exit_status[i] = exit_status; // stores the processed exit status or signal
                break;
            }
        }
        // logs the child's termination status
        log_message(msg);
        // increments the count of exited children
        child_exit_count++;
    }
}
```

2.4) void become_daemon(): This function is mostly taken by the slides uploaded to our teams page (week05.pdf, page 38, 39, 40). Function first forks the program, exits the parent program and child continues, then calls setsid() to start a new session, then sets SIGTERM and SIGHUP signal handlers, then forks the program again and exiting the parent to prevent any association with the controlling terminal, then sets the SIGCHILD signal handler, changes the working directory to root, clears the file mode creation mask with umask(0), then closes the inherited file descriptors *except the log file descriptor so that daemon still have one file descriptor to use for the logging operations, then redirects the stdin to /dev/null (this part is from the slide, then this descriptor will be closed but still I want to keep it), then redirects stdout and stderr to log file descriptor, then closes the null file descriptor because we don't need it.



```c
void become_daemon() {
    pid_t pid = fork();
    if (pid < 0) exit(EXIT_FAILURE); // Error in fork
    if (pid > 0) exit(EXIT_SUCCESS); // Parent exits

    // create a new session and become the session leader
    if (setsid() < 0) exit(EXIT_FAILURE);

    // set the signal handlers for the daemon
    signal(SIGTERM, daemon_signal_handler);
    signal(SIGHUP, daemon_signal_handler);

    // prevent acquiring a controlling terminal for the daemon process to avoid being killed by a terminal signal
    pid = fork();

    // error in fork
    if (pid < 0) exit(EXIT_FAILURE);

    // parent exits
    if (pid > 0) exit(EXIT_SUCCESS);

    // re-set SIGCHLD handler after the second fork
    signal(SIGCHLD, handle_sigchld);

    // change working directory to root
    chdir("/");

    // clear file mode creation mask
    umask(0);

    // close inherited file descriptors
    int log_fd = fileno(log_file); // Get the file descriptor of log_file
    for (int i = 0; i < sysconf(_SC_OPEN_MAX); i++) {
        if (i != log_fd) close(i); // Skip closing the log file descriptor
    }

    // reopen stdin, stdout, stderr to /dev/null or log files
    int null_fd = open("/dev/null", O_RDWR);
    dup2(null_fd, STDIN_FILENO);
    dup2(log_fd, STDOUT_FILENO); // Redirect stdout to log_file
    dup2(log_fd, STDERR_FILENO); // Redirect stderr to log_file
    close(null_fd);              // Close the extra /dev/null descriptor

}
```

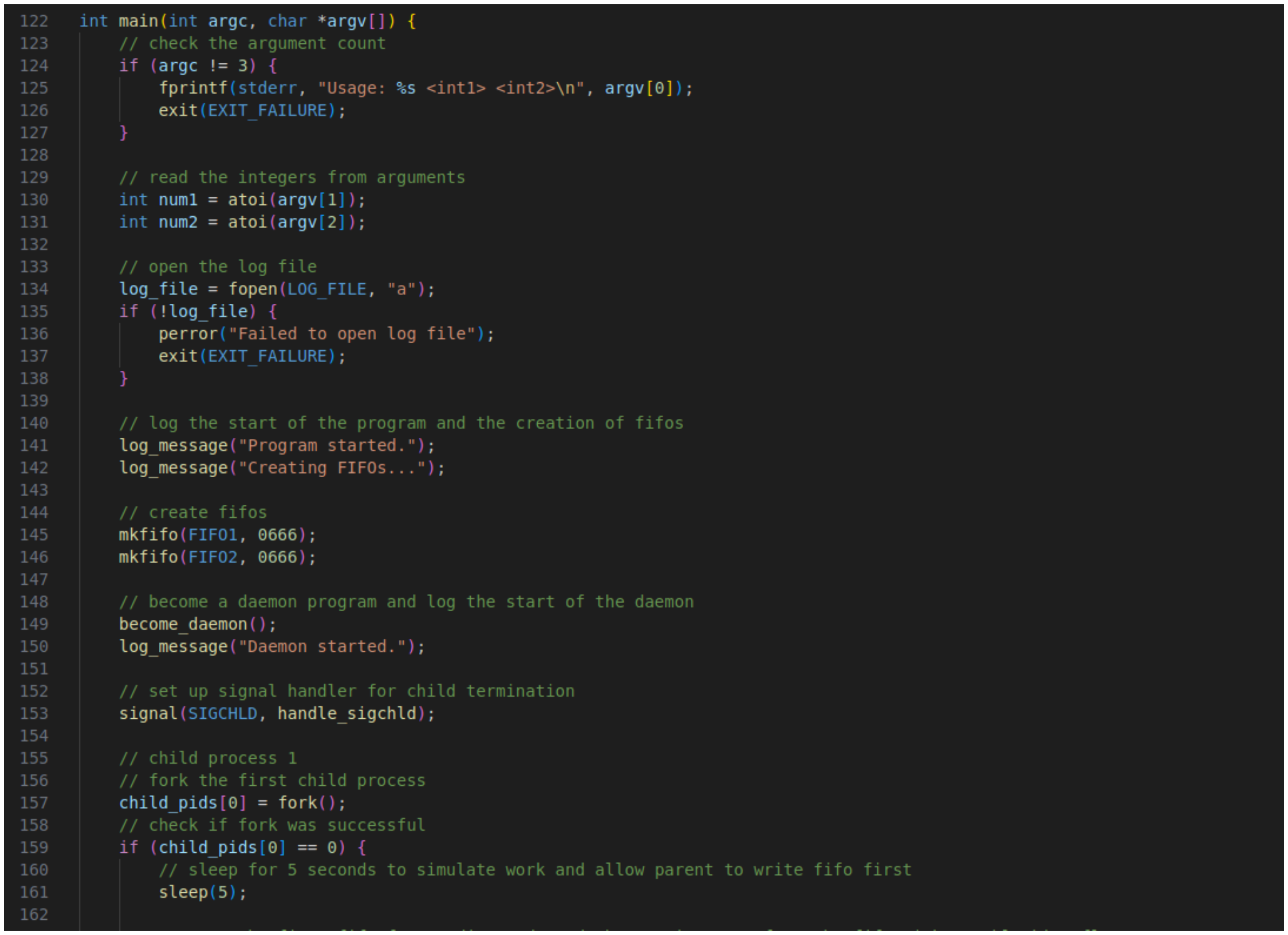
2.5) int main(int argc, char *argv[]): Main function first checks the argument count to check the count of the input numbers, then assigns them num1 and num2, opens the log file descriptor in append mode, then prints the program start messages and creates fifos. Then program became a daemon program.

I have tried to create child processes before the program daemonized but I couldn't catch their SIGCHILD signals in the daemonized parent process, so that's why I have daemonized the program before creating the child processes as we allowed to do which has announced in the course teams page.

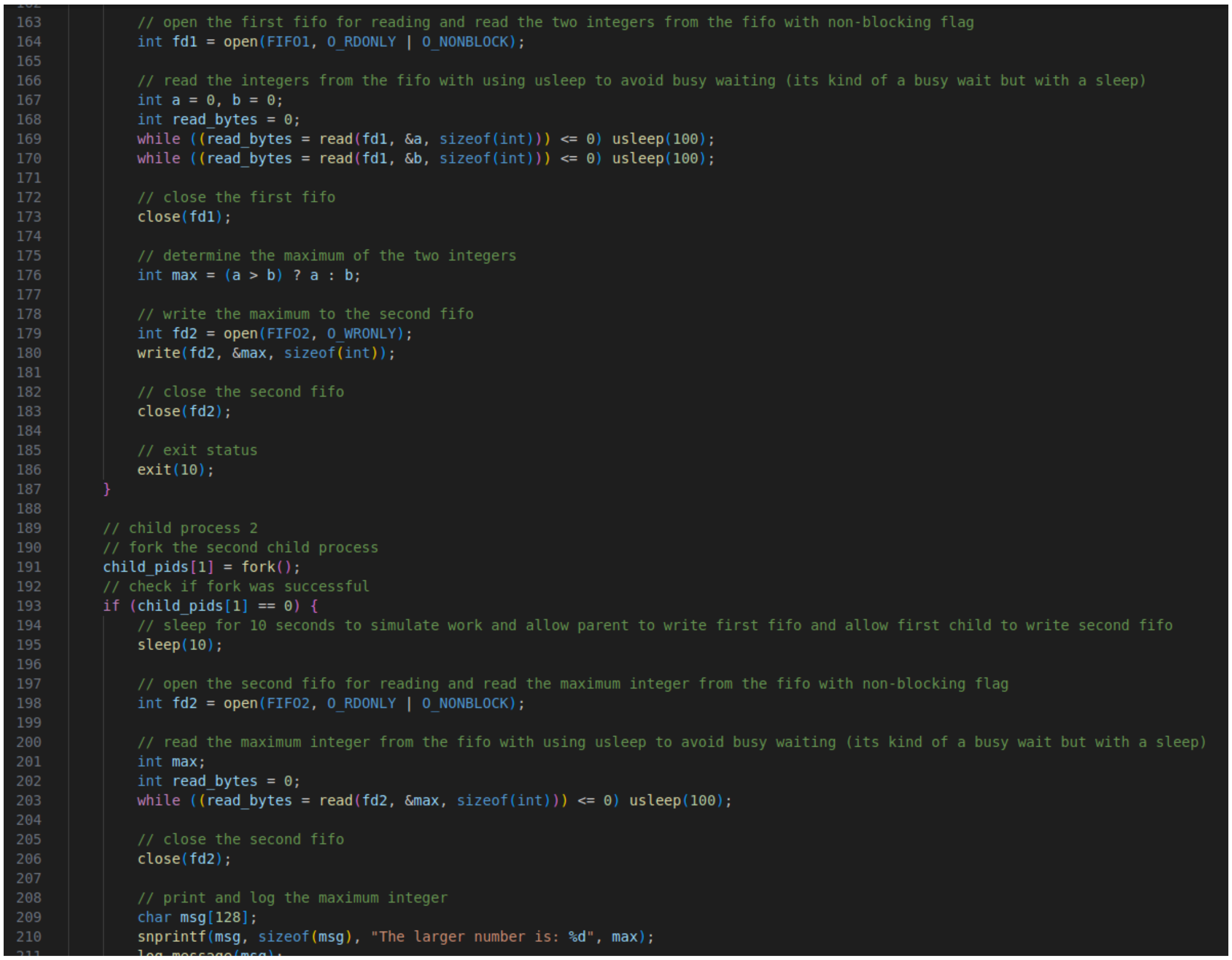
// first child
Then program creates the first child process with forking the program, first child sleeps for 5 seconds to let parent process create the second child and write numbers to fifo1, after sleep(5), child process 1 opens the first fifo in non-blocking mode as we asked in the document, reads the numbers in a while loop with using usleep to avoid busy waiting, then closes the pipe, determines the maximum number between the two input numbers, opens the second fifo and writes the maximum number in it, closes the second fifo and exits with the status 10. I have implemented exit(10), so that I can show in the log file I can catch the child processes exit statuses and log them in file.

// second child
Then program creates the second child process, second child process sleeps for 10 seconds to let parent process write the first pipe, first child process to sleep 5 seconds and reads the first pipe and writes the maximum number to the second fifo. After sleep(10), second child process opens the second fifo, reads the maximum number in a while loop with using usleep to avoid busy waiting, then closes the second fifo, logs the "larger number is: %d" message to the log file, and exits with status 20. I have implemented exit(20), so that I can show in the log file I can catch the child processes exit statuses and log them in file.

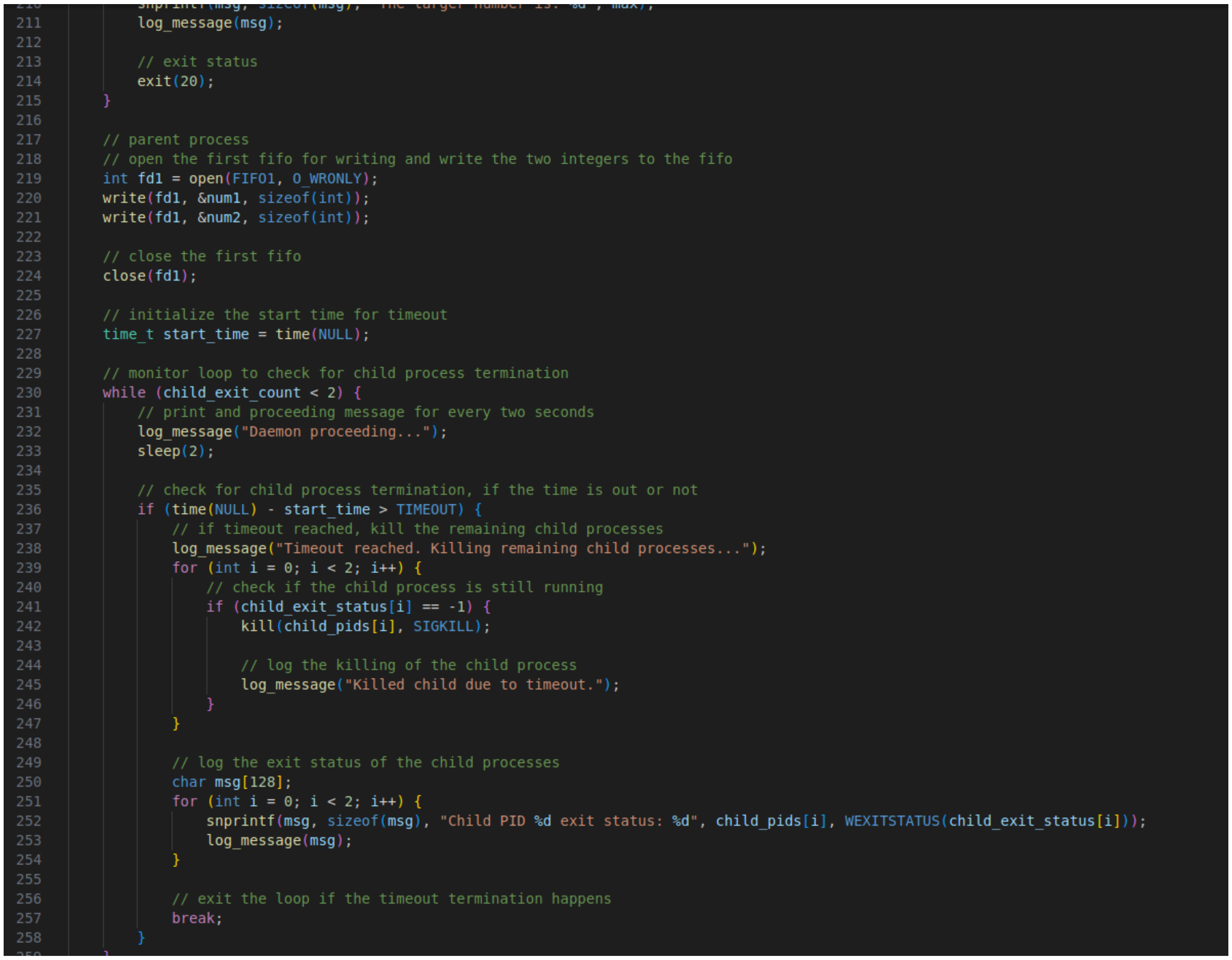
// parent process (daemonized at the beginning)
After creating child processes, the parent process opens the first fifo and writes the num1 and num2, closes the first fifo and initializes the start time for timeout logic. Then enters in a while loop to check the exited child counts are reached to 2 or the defined timeout has over and prints “Daemon proceeding…” into the log file in every 2 seconds with using sleep(2). If the timeout has overed, related timeout message has logged to log file and both childs are killed with a signal SIGKILL with using kill() function and logs the “Killed child due to timeout” message and prints the childs PID’s and exit statuses to the log file. If childs are normally exited with their own exit statuses, that's handled in the handle_sigchld() function and logged to file, this one is about the processes that reaches the timeout and exits with a signal. Whether with their own exit commands and statuses or due to timeout exits, after both childs are exits, program logs the “Daemon shutting down” message, closes the log file descriptor, unlinks the first and second fifo and terminates the program.



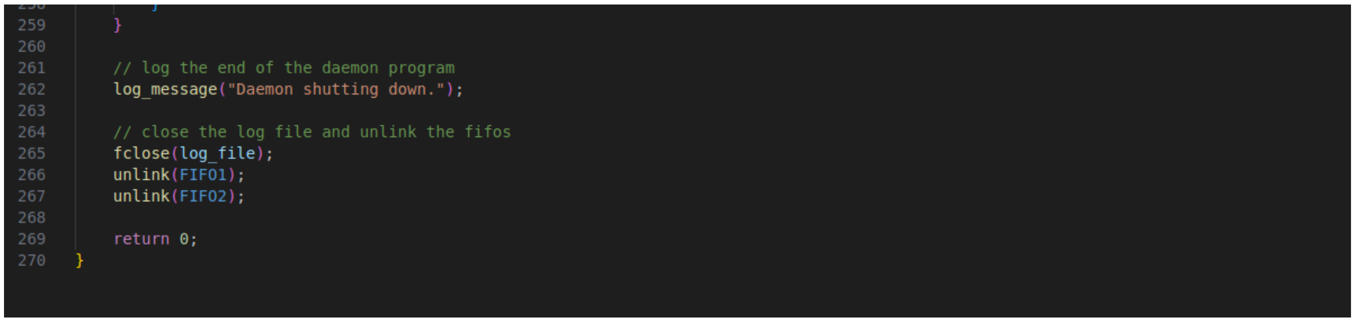
```c
int main(int argc, char *argv[]) {
    // check the argument count
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <int1> <int2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // read the integers from arguments
    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);

    // open the log file
    log_file = fopen(LOG_FILE, "a");
    if (!log_file) {
        perror("Failed to open log file");
        exit(EXIT_FAILURE);
    }

    // log the start of the program and the creation of fifos
    log_message("Program started.");
    log_message("Creating FIFOs...");

    // create fifos
    mkfifo(FIFO1, 0666);
    mkfifo(FIFO2, 0666);

    // become a daemon program and log the start of the daemon
    become_daemon();
    log_message("Daemon started.");

    // set up signal handler for child termination
    signal(SIGCHLD, handle_sigchld);

    // child process 1
    // fork the first child process
    child_pids[0] = fork();
    // check if fork was successful
    if (child_pids[0] == 0) {
        // sleep for 5 seconds to simulate work and allow parent to write fifo first
        sleep(5);

```



```c
        // open the first fifo for reading and read the two integers from the fifo with non-blocking flag
        int fd1 = open(FIFO1, O_RDONLY | O_NONBLOCK);

        // read the integers from the fifo with using usleep to avoid busy waiting (its kind of a busy wait but with a sleep)
        int a = 0, b = 0;
        int read_bytes = 0;
        while ((read_bytes = read(fd1, &a, sizeof(int))) <= 0) usleep(100);
        while ((read_bytes = read(fd1, &b, sizeof(int))) <= 0) usleep(100);

        // close the first fifo
        close(fd1);

        // determine the maximum of the two integers
        int max = (a > b) ? a : b;

        // write the maximum to the second fifo
        int fd2 = open(FIFO2, O_WRONLY);
        write(fd2, &max, sizeof(int));

        // close the second fifo
        close(fd2);

        // exit status
        exit(10);
    }

    // child process 2
    // fork the second child process
    child_pids[1] = fork();
    // check if fork was successful
    if (child_pids[1] == 0) {
        // sleep for 10 seconds to simulate work and allow parent to write first fifo and allow first child to write second fifo
        sleep(10);

        // open the second fifo for reading and read the maximum integer from the fifo with non-blocking flag
        int fd2 = open(FIFO2, O_RDONLY | O_NONBLOCK);

        // read the maximum integer from the fifo with using usleep to avoid busy waiting (its kind of a busy wait but with a sleep)
        int max;
        int read_bytes = 0;
        while ((read_bytes = read(fd2, &max, sizeof(int))) <= 0) usleep(100);

        // close the second fifo
        close(fd2);

        // print and log the maximum integer
        char msg[128];
        snprintf(msg, sizeof(msg), "The larger number is: %d", max);
        log_message(msg);
```



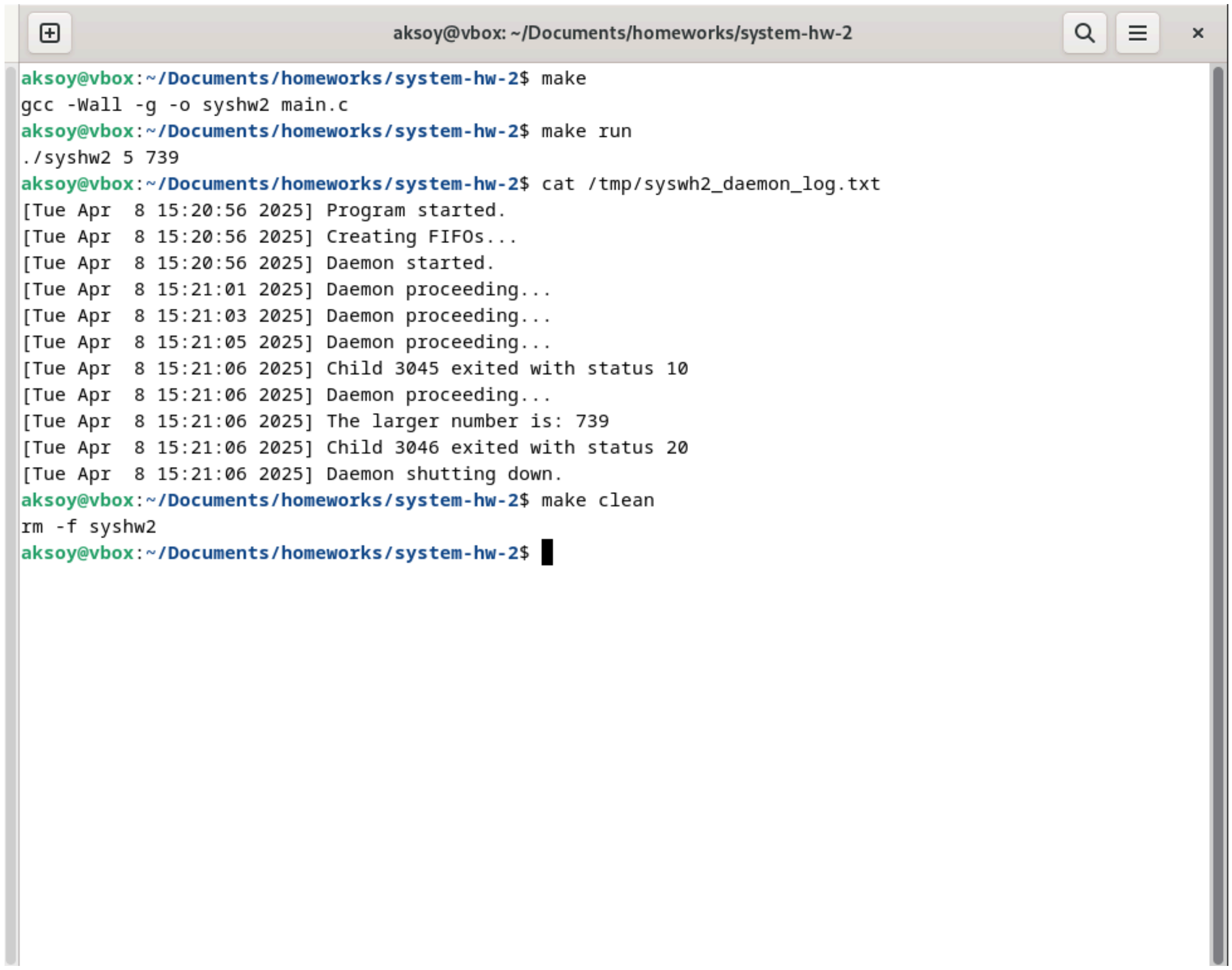
```c
        snprintf(msg, sizeof(msg), "The larger number is: %d", max);
        log_message(msg);

        // exit status
        exit(20);
    }

    // parent process
    // open the first fifo for writing and write the two integers to the fifo
    int fd1 = open(FIFO1, O_WRONLY);
    write(fd1, &num1, sizeof(int));
    write(fd1, &num2, sizeof(int));

    // close the first fifo
    close(fd1);

    // initialize the start time for timeout
    time_t start_time = time(NULL);

    // monitor loop to check for child process termination
    while (child_exit_count < 2) {
        // print and proceeding message for every two seconds
        log_message("Daemon proceeding...");
        sleep(2);

        // check for child process termination, if the time is out or not
        if (time(NULL) - start_time > TIMEOUT) {
            // if timeout reached, kill the remaining child processes
            log_message("Timeout reached. Killing remaining child processes...");
            for (int i = 0; i < 2; i++) {
                // check if the child process is still running
                if (child_exit_status[i] == -1) {
                    kill(child_pids[i], SIGKILL);

                    // log the killing of the child process
                    log_message("Killed child due to timeout.");
                }
            }

            // log the exit status of the child processes
            char msg[128];
            for (int i = 0; i < 2; i++) {
                snprintf(msg, sizeof(msg), "Child PID %d exit status: %d", child_pids[i], WEXITSTATUS(child_exit_status[i]));
                log_message(msg);
            }

            // exit the loop if the timeout termination happens
            break;
        }
```



```c
    }

    // log the end of the daemon program
    log_message("Daemon shutting down.");

    // close the log file and unlink the fifos
    fclose(log_file);
    unlink(FIFO1);
    unlink(FIFO2);

    return 0;
}
```

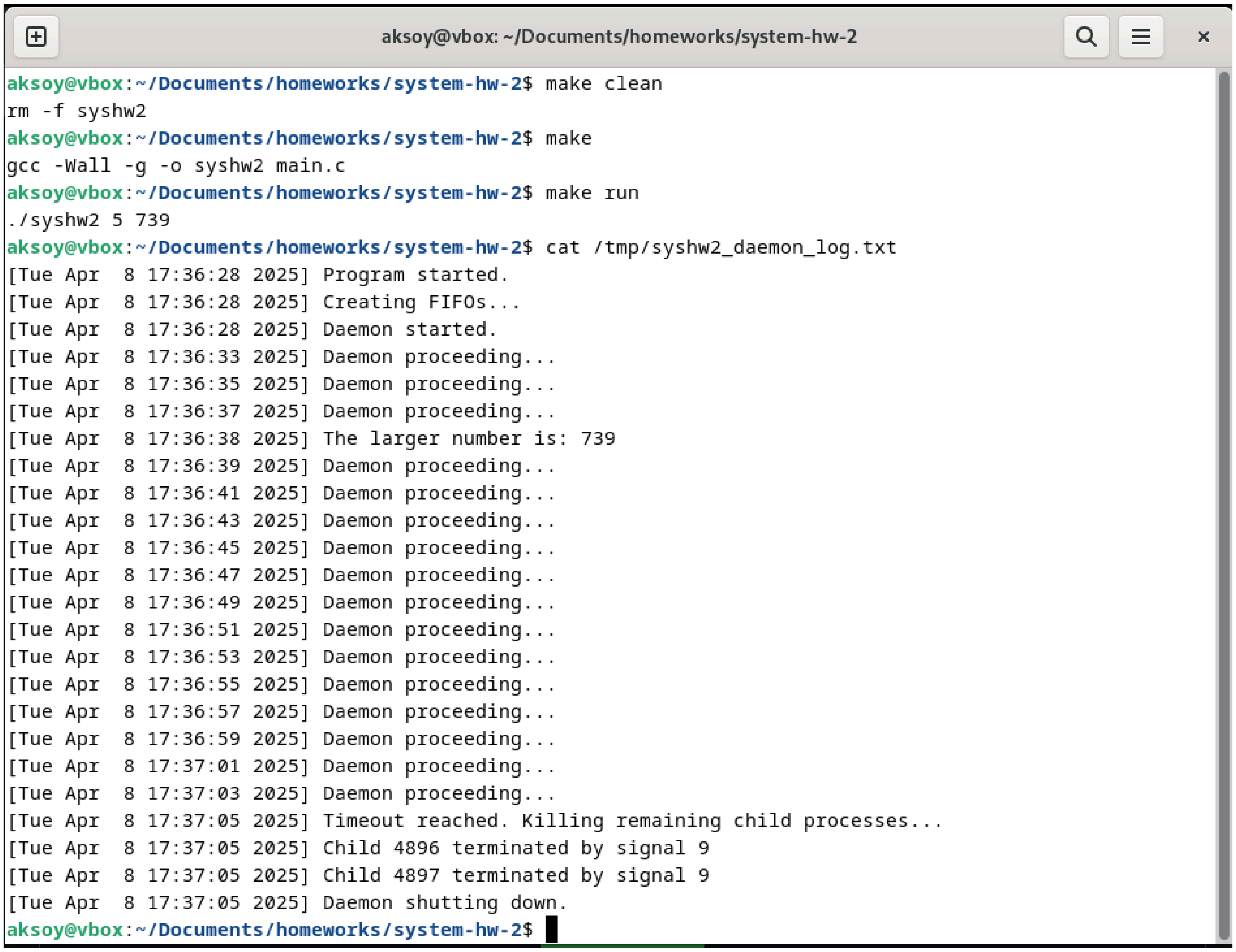
3) Screenshots

Testing program in Debian 12 virtual box with inputs 5 and 739 which I give in the makefile:

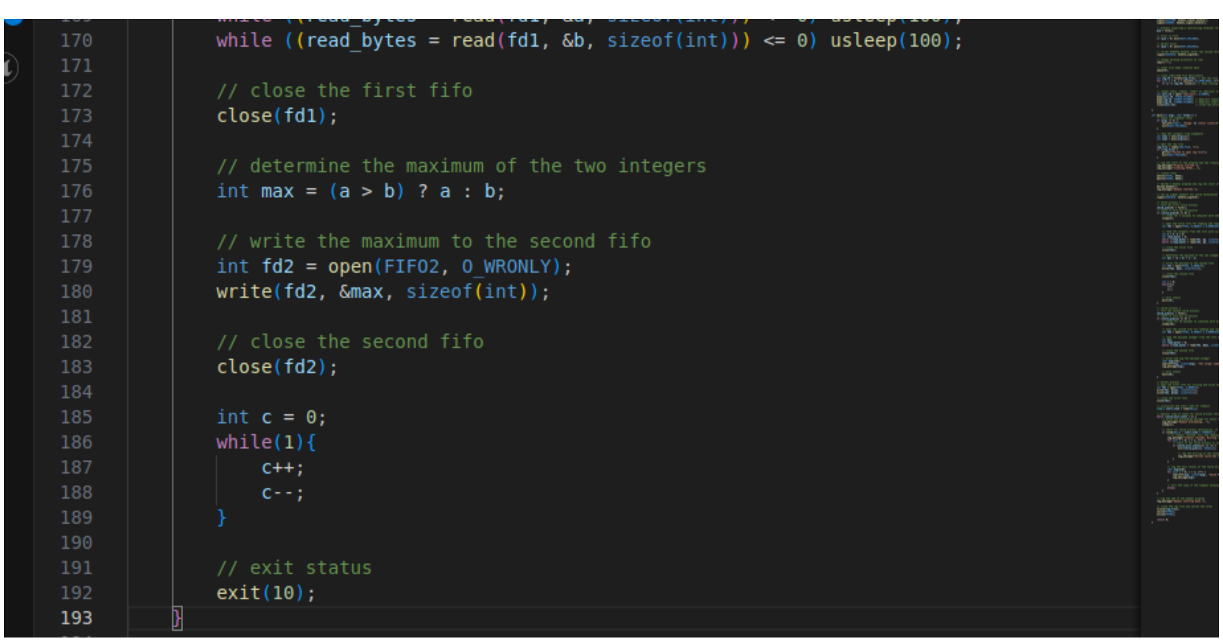


```
aksoy@vbox:~/Documents/homeworks/system-hw-2$ make
gcc -Wall -g -o syshw2 main.c
aksoy@vbox:~/Documents/homeworks/system-hw-2$ make run
./syshw2 5 739
aksoy@vbox:~/Documents/homeworks/system-hw-2$ cat /tmp/syswh2_daemon_log.txt
[Tue Apr  8 15:20:56 2025] Program started.
[Tue Apr  8 15:20:56 2025] Creating FIFOs...
[Tue Apr  8 15:20:56 2025] Daemon started.
[Tue Apr  8 15:21:01 2025] Daemon proceeding...
[Tue Apr  8 15:21:03 2025] Daemon proceeding...
[Tue Apr  8 15:21:05 2025] Daemon proceeding...
[Tue Apr  8 15:21:06 2025] Child 3045 exited with status 10
[Tue Apr  8 15:21:06 2025] Daemon proceeding...
[Tue Apr  8 15:21:06 2025] The larger number is: 739
[Tue Apr  8 15:21:06 2025] Child 3046 exited with status 20
[Tue Apr  8 15:21:06 2025] Daemon shutting down.
aksoy@vbox:~/Documents/homeworks/system-hw-2$ make clean
rm -f syshw2
aksoy@vbox:~/Documents/homeworks/system-hw-2$ 
```

Testing the program’s timeout algorithm with adding dummy while(1) busy wait loops inside of the child processes.



```
aksoy@vbox:~/Documents/homeworks/system-hw-2$ make clean
rm -f syshw2
aksoy@vbox:~/Documents/homeworks/system-hw-2$ make
gcc -Wall -g -o syshw2 main.c
aksoy@vbox:~/Documents/homeworks/system-hw-2$ make run
./syshw2 5 739
aksoy@vbox:~/Documents/homeworks/system-hw-2$ cat /tmp/syshw2_daemon_log.txt
[Tue Apr  8 17:36:28 2025] Program started.
[Tue Apr  8 17:36:28 2025] Creating FIFOs...
[Tue Apr  8 17:36:28 2025] Daemon started.
[Tue Apr  8 17:36:33 2025] Daemon proceeding...
[Tue Apr  8 17:36:35 2025] Daemon proceeding...
[Tue Apr  8 17:36:37 2025] Daemon proceeding...
[Tue Apr  8 17:36:38 2025] The larger number is: 739
[Tue Apr  8 17:36:39 2025] Daemon proceeding...
[Tue Apr  8 17:36:41 2025] Daemon proceeding...
[Tue Apr  8 17:36:43 2025] Daemon proceeding...
[Tue Apr  8 17:36:45 2025] Daemon proceeding...
[Tue Apr  8 17:36:47 2025] Daemon proceeding...
[Tue Apr  8 17:36:49 2025] Daemon proceeding...
[Tue Apr  8 17:36:51 2025] Daemon proceeding...
[Tue Apr  8 17:36:53 2025] Daemon proceeding...
[Tue Apr  8 17:36:55 2025] Daemon proceeding...
[Tue Apr  8 17:36:57 2025] Daemon proceeding...
[Tue Apr  8 17:36:59 2025] Daemon proceeding...
[Tue Apr  8 17:37:01 2025] Daemon proceeding...
[Tue Apr  8 17:37:03 2025] Daemon proceeding...
[Tue Apr  8 17:37:05 2025] Timeout reached. Killing remaining child processes...
[Tue Apr  8 17:37:05 2025] Child 4896 terminated by signal 9
[Tue Apr  8 17:37:05 2025] Child 4897 terminated by signal 9
[Tue Apr  8 17:37:05 2025] Daemon shutting down.
aksoy@vbox:~/Documents/homeworks/system-hw-2$
```



```c
    while ((read_bytes = read(fd1, &b, sizeof(int))) <= 0) usleep(100);

    // close the first fifo
    close(fd1);

    // determine the maximum of the two integers
    int max = (a > b) ? a : b;

    // write the maximum to the second fifo
    int fd2 = open(FIFO2, O_WRONLY);
    write(fd2, &max, sizeof(int));

    // close the second fifo
    close(fd2);

    int c = 0;
    while(1){
        c++;
        c--;
    }

    // exit status
    exit(10);
}
```

4) Conclusion

I have encountered challenges when I try to implement signal handlers before daemonize the parent process, because all file descriptors and previous consoles were closed after the become_daemon() function, so I have changed the program as it was making the program as daemon in the beginning of the program before creating the child processes as we allowed to do which is declared in the course teams page.

I have encountered another challenge to implement the timeout logic where I could not catch the kill signal in my signal handler when I send kill signal to child processes PIDs. I have solved this issue with adding two second sleep to the parent process because it immediately exits the program after terminating the child processes after timeout.