CSE344 -- HOMEWORK #2

08/04/2025

Cemal BOLAT

STUDENT NO: 210104004010

Introduction:

This project focuses on building a communication system between two processes using Inter-Process Communication (IPC) via FIFOs (named pipes), enhancing the system with a daemon process responsible for background operations. The program begins by accepting two integer values as command-line arguments and initializes the communication channels required for process interaction.

Once initialized, the parent process creates two child processes using the fork() system call. The first child reads the two numbers from a FIFO, determines the larger value, and writes the result into another FIFO. The second child then reads this result and prints it to the screen. Throughout this operation, the parent process monitors the execution by printing periodic status messages and handling signals from child processes using a SIGCHLD signal handler. This handler manages the termination of child processes and keeps track of their completion through a counter mechanism.

A daemon process is also implemented to handle background logging, monitor process status, and ensure stability through signal handling (such as SIGUSR1, SIGTERM, and SIGHUP). It logs essential details such as process start and end times, PIDs, and any runtime errors, ensuring the system can be monitored and debugged effectively. The daemon also includes timeout logic and non-blocking FIFO operations to avoid deadlocks and detect unresponsive processes.

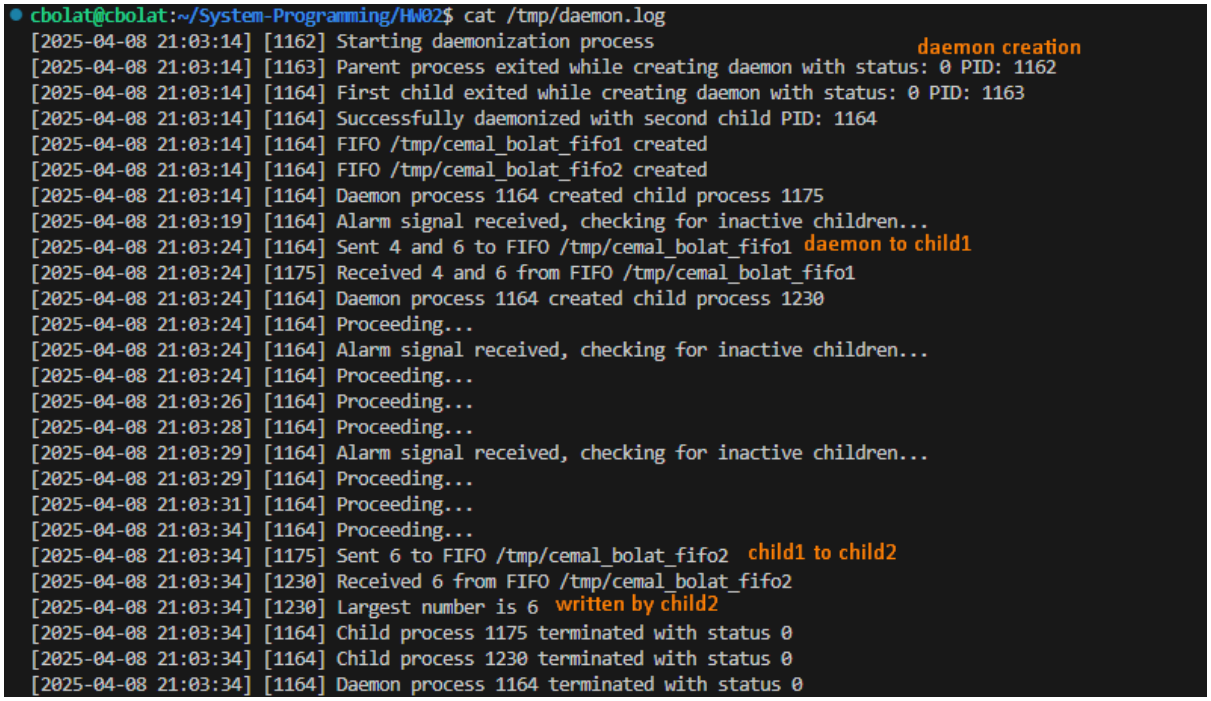
This system demonstrates a complete cycle of process creation, communication, monitoring, and termination with an emphasis on error handling, background service management, and resource cleanup. Each component was implemented with a strong focus on modularity and reliability, ensuring the processes interact smoothly and terminate gracefully under different conditions.

TESTING (all scenerios created by me):

TESTING WITHOUT PROBLEM

```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ valgrind ./ipc -5 67
==2460== Memcheck, a memory error detector
==2460== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2460== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==2460== Command: ./ipc -5 67
==2460==
==2460==
==2460== HEAP SUMMARY:
==2460==     in use at exit: 0 bytes in 0 blocks
==2460==   total heap usage: 10 allocs, 10 frees, 5,869 bytes allocated
==2460==
==2460== All heap blocks were freed -- no leaks are possible
==2460==
==2460== For lists of detected and suppressed errors, rerun with: -s
==2460== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ ==2461==
==2461== HEAP SUMMARY:
==2461==     in use at exit: 0 bytes in 0 blocks
==2461==   total heap usage: 11 allocs, 11 frees, 5,884 bytes allocated
==2461==
==2461== All heap blocks were freed -- no leaks are possible
==2461==
==2461== For lists of detected and suppressed errors, rerun with: -s
==2461== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ ==2463==
==2463== HEAP SUMMARY:
==2463==     in use at exit: 0 bytes in 0 blocks
==2463==   total heap usage: 17 allocs, 17 frees, 5,974 bytes allocated
==2463==
==2463== All heap blocks were freed -- no leaks are possible
==2463==
==2463== For lists of detected and suppressed errors, rerun with: -s
==2463== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==2474==
==2474== HEAP SUMMARY:
==2474==     in use at exit: 0 bytes in 0 blocks
==2474==   total heap usage: 22 allocs, 22 frees, 6,049 bytes allocated
==2474==
==2474== All heap blocks were freed -- no leaks are possible
==2474==
==2474== For lists of detected and suppressed errors, rerun with: -s
==2474== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==2462==
==2462== HEAP SUMMARY:
==2462==     in use at exit: 0 bytes in 0 blocks
==2462==   total heap usage: 35 allocs, 35 frees, 6,244 bytes allocated
==2462==
==2462== All heap blocks were freed -- no leaks are possible
==2462==
==2462== For lists of detected and suppressed errors, rerun with: -s
==2462== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$
```

Valgrind output is clear there is no leaks on no purposes

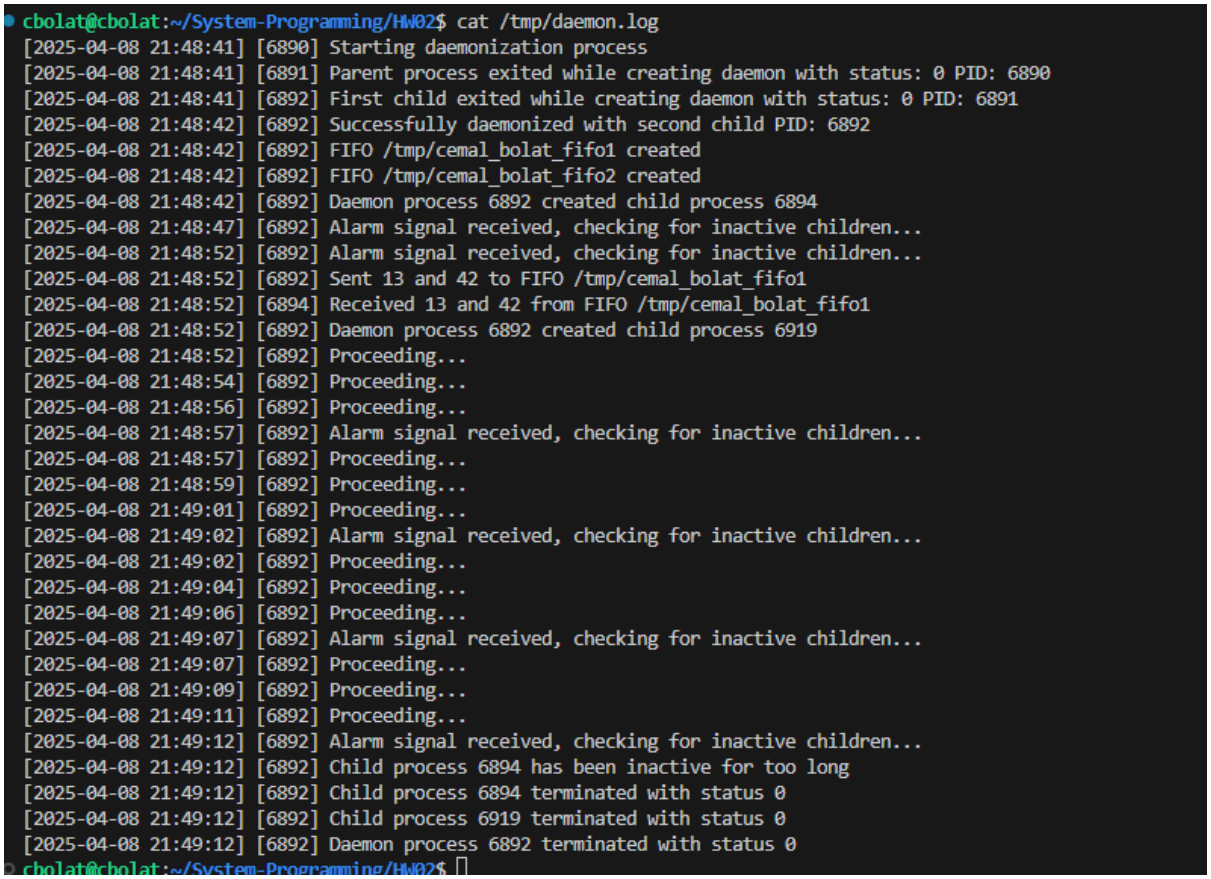


LOG FILE AFTER EXECUTION

TESTING WHILE CHILD 1 IS INACTIVE AFTER CHILD 2 IS CREATED

```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ valgrind ./ipc 56 78
==2562== Memcheck, a memory error detector
==2562== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2562== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==2562== Command: ./ipc 56 78
==2562==
==2562==
==2562== HEAP SUMMARY:
==2562==     in use at exit: 0 bytes in 0 blocks
==2562==   total heap usage: 10 allocs, 10 frees, 5,869 bytes allocated
==2562==
==2562== All heap blocks were freed -- no leaks are possible
==2562==
==2562== For lists of detected and suppressed errors, rerun with: -s
==2562== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ ==2563==
==2563== HEAP SUMMARY:
==2563==     in use at exit: 0 bytes in 0 blocks
==2563==   total heap usage: 11 allocs, 11 frees, 5,884 bytes allocated
==2563==
==2563== All heap blocks were freed -- no leaks are possible
==2563==
==2563== For lists of detected and suppressed errors, rerun with: -s
==2563== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ ==2564==
==2564== HEAP SUMMARY:
==2564==     in use at exit: 0 bytes in 0 blocks
==2564==   total heap usage: 27 allocs, 27 frees, 6,124 bytes allocated
==2564==
==2564== All heap blocks were freed -- no leaks are possible
==2564==
==2564== For lists of detected and suppressed errors, rerun with: -s
==2564== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==2565==
==2565== HEAP SUMMARY:
==2565==     in use at exit: 0 bytes in 0 blocks
==2565==   total heap usage: 16 allocs, 16 frees, 5,959 bytes allocated
==2565==
==2565== All heap blocks were freed -- no leaks are possible
==2565==
==2565== For lists of detected and suppressed errors, rerun with: -s
==2565== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==2566==
==2566== HEAP SUMMARY:
==2566==     in use at exit: 0 bytes in 0 blocks
==2566==   total heap usage: 20 allocs, 20 frees, 6,019 bytes allocated
==2566==
==2566== All heap blocks were freed -- no leaks are possible
==2566==
==2566== For lists of detected and suppressed errors, rerun with: -s
==2566== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$
```

Valgrind output is clear there is no leaks on no purposes



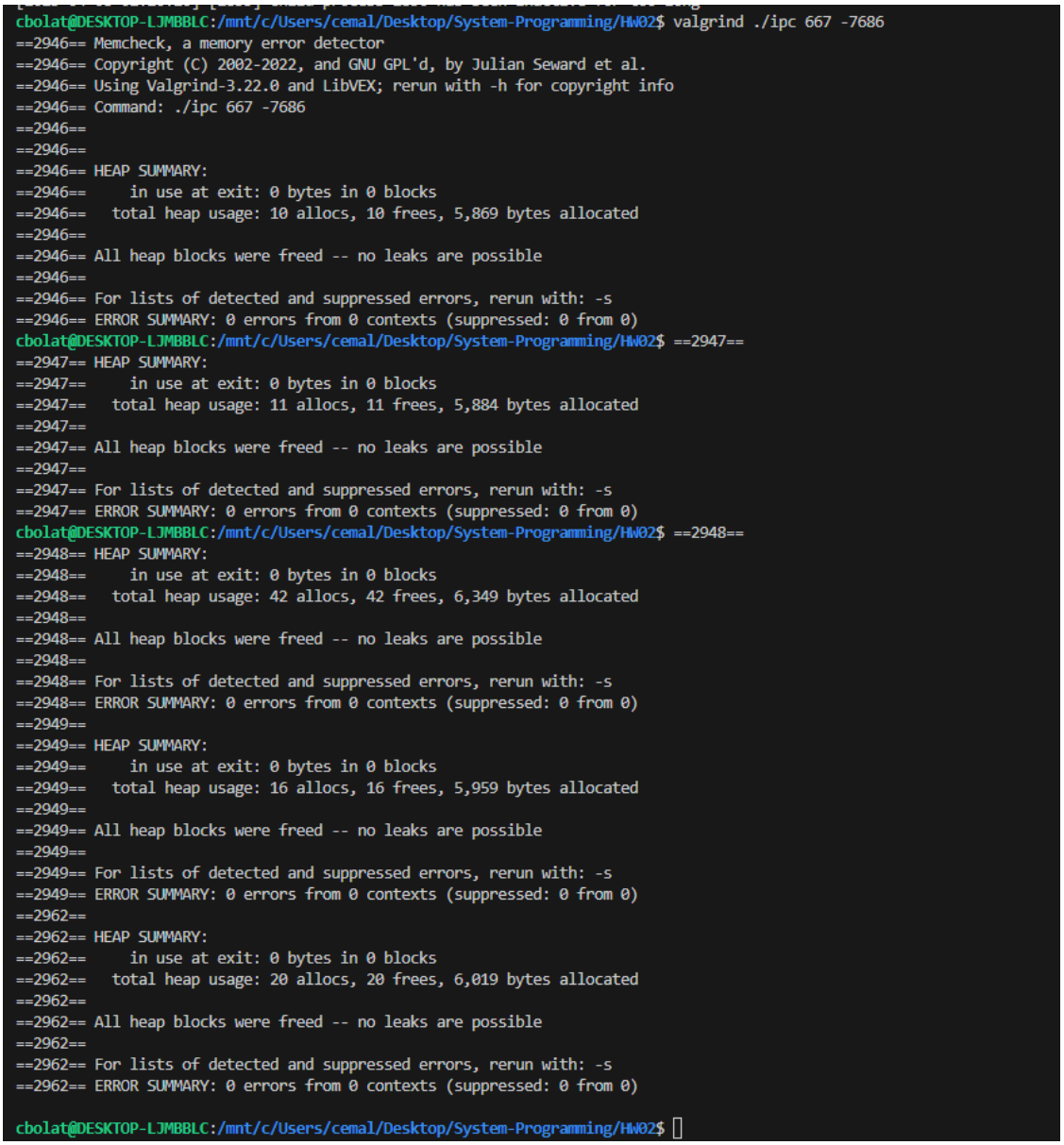
LOG FILE AFTER EXECUTION

TESTING WHILE CHILD 1 IS INACTIVE BEFORE CHILD 2 IS CREATED

```
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ valgrind ./ipc 123 -856
==2777== Memcheck, a memory error detector
==2777== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2777== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==2777== Command: ./ipc 123 -856
==2777==
==2777==
==2777== HEAP SUMMARY:
==2777==     in use at exit: 0 bytes in 0 blocks
==2777==   total heap usage: 10 allocs, 10 frees, 5,869 bytes allocated
==2777==
==2777== All heap blocks were freed -- no leaks are possible
==2777==
==2777== For lists of detected and suppressed errors, rerun with: -s
==2777== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ ==2778==
==2778== HEAP SUMMARY:
==2778==     in use at exit: 0 bytes in 0 blocks
==2778==   total heap usage: 11 allocs, 11 frees, 5,884 bytes allocated
==2778==
==2778== All heap blocks were freed -- no leaks are possible
==2778==
==2778== For lists of detected and suppressed errors, rerun with: -s
==2778== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ ==2779==
==2779== HEAP SUMMARY:
==2779==     in use at exit: 0 bytes in 0 blocks
==2779==   total heap usage: 22 allocs, 22 frees, 6,049 bytes allocated
==2779==
==2779== All heap blocks were freed -- no leaks are possible
==2779==
==2779== For lists of detected and suppressed errors, rerun with: -s
==2779== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==2780==
==2780== HEAP SUMMARY:
==2780==     in use at exit: 0 bytes in 0 blocks
==2780==   total heap usage: 15 allocs, 15 frees, 5,944 bytes allocated
==2780==
==2780== All heap blocks were freed -- no leaks are possible
==2780==
==2780== For lists of detected and suppressed errors, rerun with: -s
==2780== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$
```

```
cbolat@cbolat:~/System-Programming/HW02$ cat /tmp/daemon.log
[2025-04-08 21:49:39] [7018] Starting daemonization process
[2025-04-08 21:49:39] [7019] Parent process exited while creating daemon with status: 0 PID: 7018
[2025-04-08 21:49:39] [7020] First child exited while creating daemon with status: 0 PID: 7019
[2025-04-08 21:49:40] [7020] Successfully daemonized with second child PID: 7020
[2025-04-08 21:49:40] [7020] FIFO /tmp/cemal_bolat_fifo1 created
[2025-04-08 21:49:40] [7020] FIFO /tmp/cemal_bolat_fifo2 created
[2025-04-08 21:49:40] [7020] Daemon process 7020 created child process 7022
[2025-04-08 21:49:45] [7020] Alarm signal received, checking for inactive children...
[2025-04-08 21:49:50] [7020] Alarm signal received, checking for inactive children...
[2025-04-08 21:49:56] [7020] Alarm signal received, checking for inactive children...
[2025-04-08 21:50:01] [7020] Alarm signal received, checking for inactive children...
[2025-04-08 21:50:06] [7020] Alarm signal received, checking for inactive children...
[2025-04-08 21:50:06] [7020] Child process 7022 has been inactive for too long
[2025-04-08 21:50:06] [7020] Child process 7022 terminated with status 0
[2025-04-08 21:50:06] [7020] Daemon process 7020 terminated with status 0
cbolat@cbolat:~/System-Programming/HW02$
```

TESTING WHILE CHILD 2 IS INACTIVE AFTER BOTH CHILD CREATED



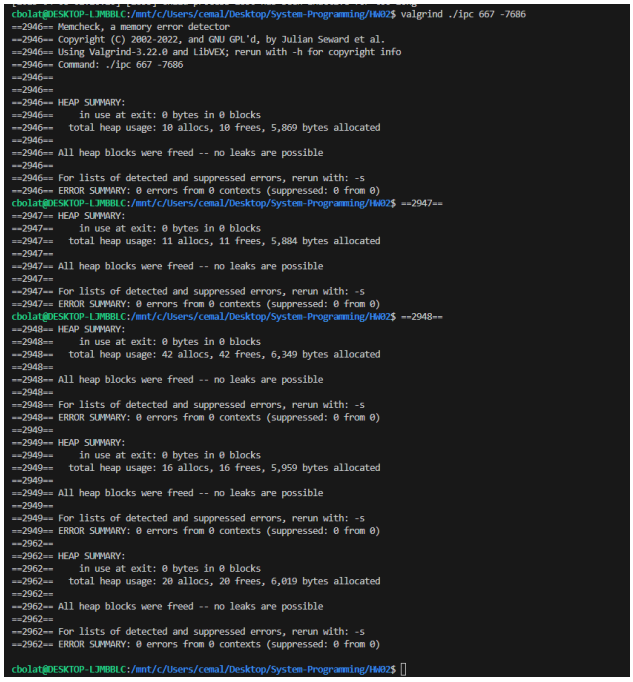
```
cbolat@cbolat:~/System-Programming/HW02$ cat /tmp/daemon.log
[2025-04-08 21:51:10] [7170] Starting daemonization process
[2025-04-08 21:51:10] [7171] Parent process exited while creating daemon with status: 0 PID: 7170
[2025-04-08 21:51:10] [7172] First child exited while creating daemon with status: 0 PID: 7171
[2025-04-08 21:51:10] [7172] Successfully daemonized with second child PID: 7172
[2025-04-08 21:51:10] [7172] FIFO /tmp/cemal_bolat_fifo1 created
[2025-04-08 21:51:10] [7172] FIFO /tmp/cemal_bolat_fifo2 created
[2025-04-08 21:51:10] [7172] Daemon process 7172 created child process 7174
[2025-04-08 21:51:15] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:20] [7174] Received 13 and 42 from FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:51:20] [7172] Sent 13 and 42 to FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:51:21] [7172] Daemon process 7172 created child process 7206
[2025-04-08 21:51:21] [7172] Proceeding...
[2025-04-08 21:51:21] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:21] [7172] Proceeding...
[2025-04-08 21:51:23] [7172] Proceeding...
[2025-04-08 21:51:25] [7172] Proceeding...
[2025-04-08 21:51:26] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:26] [7172] Proceeding...
[2025-04-08 21:51:28] [7172] Proceeding...
[2025-04-08 21:51:30] [7172] Proceeding...
[2025-04-08 21:51:31] [7174] Sent 42 to FIFO /tmp/cemal_bolat_fifo2
[2025-04-08 21:51:31] [7206] Received 42 from FIFO /tmp/cemal_bolat_fifo2
[2025-04-08 21:51:31] [7172] Child process 7174 terminated with status 0 exited by himself
[2025-04-08 21:51:31] [7172] Proceeding...
[2025-04-08 21:51:31] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:31] [7172] Proceeding...
[2025-04-08 21:51:33] [7172] Proceeding...
[2025-04-08 21:51:35] [7172] Proceeding...
[2025-04-08 21:51:36] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:36] [7172] Proceeding...
[2025-04-08 21:51:38] [7172] Proceeding...
[2025-04-08 21:51:40] [7172] Proceeding...
[2025-04-08 21:51:41] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:41] [7172] Proceeding...
[2025-04-08 21:51:43] [7172] Proceeding...
[2025-04-08 21:51:45] [7172] Proceeding...
[2025-04-08 21:51:46] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:46] [7172] Proceeding...
[2025-04-08 21:51:48] [7172] Proceeding...
[2025-04-08 21:51:50] [7172] Proceeding...
[2025-04-08 21:51:51] [7172] Alarm signal received, checking for inactive children...
[2025-04-08 21:51:51] [7172] Child process 7206 has been inactive for too long
[2025-04-08 21:51:51] [7172] Child process 7206 terminated with status 0 closed by daemon
[2025-04-08 21:51:51] [7172] Daemon process 7172 terminated with status 0
cbolat@cbolat:~/System-Programming/HW02$
```

```
/home/cbolat/.hushlogin file.
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$ ps aux | grep ipc
cbolat      3066  0.0  0.0   4088  1916 pts/2    S+   02:33   0:00 grep --color=auto ipc
cbolat@DESKTOP-LJMBBLC:/mnt/c/Users/cemal/Desktop/System-Programming/HW02$
```

After all execution there is no left process or anything clean exit is achieved.

SENDING SIGHUP SIGNAL WHILE PROCEEDING

```
cbolat@cbolat:~/System-Programming/HW02$ cat /tmp/daemon.log
[2025-04-08 21:58:21] [8478] Starting daemonization process
[2025-04-08 21:58:21] [8479] Parent process exited while creating daemon with status: 0 PID: 8478
[2025-04-08 21:58:21] [8480] First child exited while creating daemon with status: 0 PID: 1
[2025-04-08 21:58:22] [8480] Successfully daemonized with second child PID: 8480
[2025-04-08 21:58:22] [8480] FIFO /tmp/cemal_bolat_fifo1 created
[2025-04-08 21:58:22] [8480] FIFO /tmp/cemal_bolat_fifo2 created
[2025-04-08 21:58:22] [8480] Daemon process 8480 created child process 8482
[2025-04-08 21:58:27] [8480] Alarm signal received, checking for inactive children...
[2025-04-08 21:58:32] [8480] Alarm signal received, checking for inactive children...
[2025-04-08 21:58:32] [8480] Sent 45 and 76 to FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:58:32] [8480] Daemon process 8480 created child process 8501
[2025-04-08 21:58:32] [8480] Proceeding...
[2025-04-08 21:58:32] [8482] Received 45 and 76 from FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:58:34] [8480] Proceeding...
[2025-04-08 21:58:35] [8480] Received SIGHUP signal, terminating daemon...
[2025-04-08 21:58:35] [8480] Child process 8482 terminated with status 0
[2025-04-08 21:58:35] [8480] Child process 8501 terminated with status 0
[2025-04-08 21:58:35] [8480] All child processes terminated
[2025-04-08 21:58:35] [8480] Daemon process terminated
```



SENDING SIGTERM SIGNAL WHILE PROCEEDING

```
cbolat@cbolat:~/System-Programming/HW02$ cat /tmp/daemon.log
[2025-04-08 21:59:18] [8568] Starting daemonization process
[2025-04-08 21:59:18] [8569] Parent process exited while creating daemon with status: 0 PID: 8568
[2025-04-08 21:59:18] [8570] First child exited while creating daemon with status: 0 PID: 1
[2025-04-08 21:59:19] [8570] Successfully daemonized with second child PID: 8570
[2025-04-08 21:59:19] [8570] FIFO /tmp/cemal_bolat_fifo1 created
[2025-04-08 21:59:19] [8570] FIFO /tmp/cemal_bolat_fifo2 created
[2025-04-08 21:59:19] [8570] Daemon process 8570 created child process 8572
[2025-04-08 21:59:24] [8570] Alarm signal received, checking for inactive children...
[2025-04-08 21:59:28] [8570] Received SIGTERM signal, terminating daemon...
[2025-04-08 21:59:28] [8570] Child process 8572 terminated with status 0
[2025-04-08 21:59:28] [8570] All child processes terminated
[2025-04-08 21:59:28] [8570] Daemon process terminated
cbolat@cbolat:~/System-Programming/HW02$
```

SENDING SIGUSR1 SIGNAL WHILE PROCEEDING

```
cbolat@cbolat:~/System-Programming/HW02$ cat /tmp/daemon.log
[2025-04-08 21:54:23] [7727] Starting daemonization process
[2025-04-08 21:54:23] [7728] Parent process exited while creating daemon with status: 0 PID: 7727
[2025-04-08 21:54:23] [7729] First child exited while creating daemon with status: 0 PID: 7728
[2025-04-08 21:54:23] [7729] Successfully daemonized with second child PID: 7729
[2025-04-08 21:54:23] [7729] FIFO /tmp/cemal_bolat_fifo1 created
[2025-04-08 21:54:23] [7729] FIFO /tmp/cemal_bolat_fifo2 created
[2025-04-08 21:54:23] [7729] Daemon process 7729 created child process 7731
[2025-04-08 21:54:28] [7729] Alarm signal received, checking for inactive children...
[2025-04-08 21:54:33] [7729] Sent 13 and 42 to FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:54:33] [7731] Received 13 and 42 from FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:54:33] [7729] Daemon process 7729 created child process 7782
[2025-04-08 21:54:33] [7729] Proceeding...
[2025-04-08 21:54:33] [7729] Alarm signal received, checking for inactive children...
[2025-04-08 21:54:33] [7729] Proceeding...
[2025-04-08 21:54:35] [7729] Proceeding...
[2025-04-08 21:54:37] [7729] Received SIGUSR1 signal, reconfiguring daemon...
[2025-04-08 21:54:37] [7729] Proceeding...
[2025-04-08 21:54:39] [7729] Alarm signal received, checking for inactive children...
[2025-04-08 21:54:39] [7729] Proceeding...
[2025-04-08 21:54:41] [7729] Proceeding...
[2025-04-08 21:54:43] [7729] Proceeding...
[2025-04-08 21:54:43] [7731] Sent 42 to FIFO /tmp/cemal_bolat_fifo2
[2025-04-08 21:54:43] [7782] Received 42 from FIFO /tmp/cemal_bolat_fifo2
[2025-04-08 21:54:43] [7729] Child process 7731 terminated with status 0
[2025-04-08 21:54:43] [7729] Proceeding...
[2025-04-08 21:54:44] [7729] Alarm signal received, checking for inactive children...
[2025-04-08 21:54:44] [7729] Proceeding...
[2025-04-08 21:54:46] [7729] Proceeding...
[2025-04-08 21:54:48] [7729] Proceeding...
[2025-04-08 21:54:49] [7729] Alarm signal received, checking for inactive children...
[2025-04-08 21:54:49] [7729] Proceeding...
[2025-04-08 21:54:51] [7729] Proceeding...
[2025-04-08 21:54:53] [7729] Proceeding...
[2025-04-08 21:54:54] [7729] Alarm signal received, checking for inactive children...
[2025-04-08 21:54:54] [7729] Proceeding...
[2025-04-08 21:54:56] [7729] Proceeding...
[2025-04-08 21:54:57] [7729] Received SIGUSR1 signal, reconfiguring daemon...
[2025-04-08 21:54:57] [7729] Proceeding...
[2025-04-08 21:54:58] [7729] Received SIGUSR1 signal, reconfiguring daemon...
[2025-04-08 21:54:58] [7729] Proceeding...
[2025-04-08 21:54:59] [7729] Alarm signal received, checking for inactive children...
[2025-04-08 21:54:59] [7729] Child process 7782 has been inactive for too long
[2025-04-08 21:54:59] [7729] Child process 7782 terminated with status 0
[2025-04-08 21:54:59] [7729] Daemon process 7729 terminated with status 0
cbolat@cbolat:~/System-Programming/HW02$
```

```
cbolat@cbolat:~/System-Programming/HW02$ valgrind ./ipc 23 54
==4832== Memcheck, a memory error detector
==4832== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4832== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==4832== Command: ./ipc 23 54
==4832==
==4832==
==4832== HEAP SUMMARY:
==4832==     in use at exit: 0 bytes in 0 blocks
==4832==   total heap usage: 10 allocs, 10 frees, 5,869 bytes allocated
==4832==
==4832== All heap blocks were freed -- no leaks are possible
==4832==
==4832== For lists of detected and suppressed errors, rerun with: -s
==4832== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@cbolat:~/System-Programming/HW02$ ==4833==
==4833== HEAP SUMMARY:
==4833==     in use at exit: 0 bytes in 0 blocks
==4833==   total heap usage: 11 allocs, 11 frees, 5,884 bytes allocated
==4833==
==4833== All heap blocks were freed -- no leaks are possible
==4833==
==4833== For lists of detected and suppressed errors, rerun with: -s
==4833== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4836==
==4836== HEAP SUMMARY:
==4836==     in use at exit: 0 bytes in 0 blocks
==4836==   total heap usage: 17 allocs, 17 frees, 5,974 bytes allocated
==4836==
==4836== All heap blocks were freed -- no leaks are possible
==4836==
==4836== For lists of detected and suppressed errors, rerun with: -s
==4836== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4879==
==4879== HEAP SUMMARY:
==4879==     in use at exit: 0 bytes in 0 blocks
==4879==   total heap usage: 23 allocs, 23 frees, 6,064 bytes allocated
==4879==
==4879== All heap blocks were freed -- no leaks are possible
==4879==
==4879== For lists of detected and suppressed errors, rerun with: -s
==4879== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4834==
==4834== HEAP SUMMARY:
==4834==     in use at exit: 0 bytes in 0 blocks
==4834==   total heap usage: 34 allocs, 34 frees, 6,229 bytes allocated
==4834==
==4834== All heap blocks were freed -- no leaks are possible
==4834==
==4834== For lists of detected and suppressed errors, rerun with: -s
==4834== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

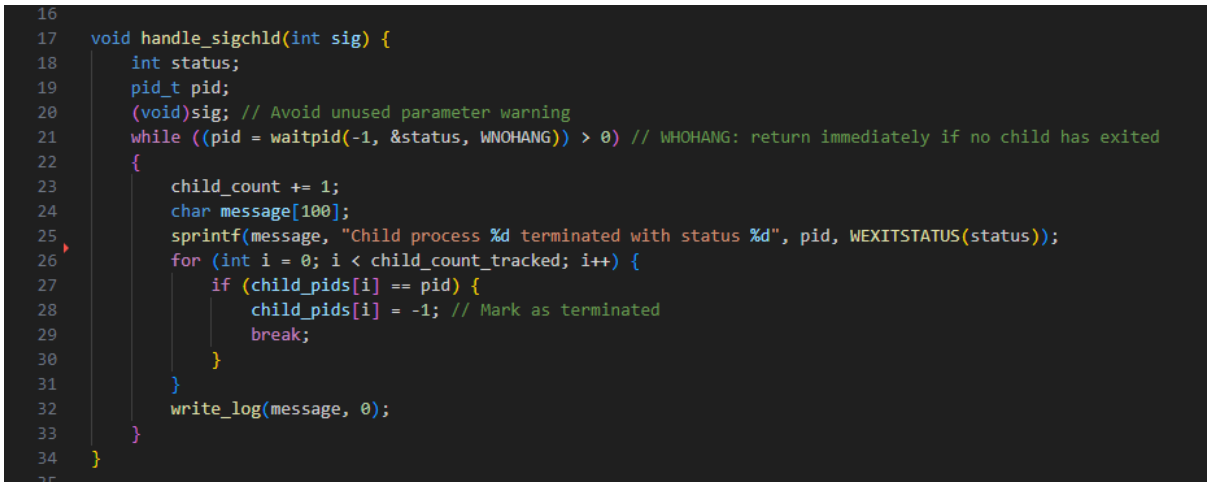
TESTING CHILD 1 IS INACTIVE AFTER CHILD 2 EXITTED

```
cbolat@cbolat:~/System-Programming/HW02$ cat /tmp/daemon.log
[2025-04-08 21:45:14] [6507] Starting daemonization process
[2025-04-08 21:45:14] [6511] Parent process exited while creating daemon with status: 0 PID: 6507
[2025-04-08 21:45:14] [6512] First child exited while creating daemon with status: 0 PID: 6511
[2025-04-08 21:45:14] [6512] Successfully daemonized with second child PID: 6512
[2025-04-08 21:45:14] [6512] FIFO /tmp/cemal_bolat_fifo1 created
[2025-04-08 21:45:14] [6512] FIFO /tmp/cemal_bolat_fifo2 created
[2025-04-08 21:45:14] [6512] Daemon process 6512 created child process 6514
[2025-04-08 21:45:19] [6512] Alarm signal received, checking for inactive children...
[2025-04-08 21:45:24] [6512] Alarm signal received, checking for inactive children...
[2025-04-08 21:45:24] [6512] Sent 4 and -86 to FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:45:24] [6514] Received 4 and -86 from FIFO /tmp/cemal_bolat_fifo1
[2025-04-08 21:45:24] [6512] Daemon process 6512 created child process 6539
[2025-04-08 21:45:24] [6512] Proceeding...
[2025-04-08 21:45:26] [6512] Proceeding...
[2025-04-08 21:45:28] [6512] Proceeding...
[2025-04-08 21:45:29] [6512] Alarm signal received, checking for inactive children...
[2025-04-08 21:45:29] [6512] Proceeding...
[2025-04-08 21:45:31] [6512] Proceeding...
[2025-04-08 21:45:33] [6512] Proceeding...
[2025-04-08 21:45:34] [6512] Alarm signal received, checking for inactive children...
[2025-04-08 21:45:34] [6512] Proceeding...
[2025-04-08 21:45:34] [6514] Sent 4 to FIFO /tmp/cemal_bolat_fifo2
[2025-04-08 21:45:34] [6539] Received 4 from FIFO /tmp/cemal_bolat_fifo2
[2025-04-08 21:45:34] [6539] Largest number is 4
[2025-04-08 21:45:34] [6512] Child process 6539 terminated with status 0
[2025-04-08 21:45:34] [6512] Proceeding...
[2025-04-08 21:45:36] [6512] Proceeding...
[2025-04-08 21:45:38] [6512] Proceeding...
[2025-04-08 21:45:39] [6512] Alarm signal received, checking for inactive children...
[2025-04-08 21:45:39] [6512] Proceeding...
[2025-04-08 21:45:41] [6512] Proceeding...
[2025-04-08 21:45:43] [6512] Proceeding...
[2025-04-08 21:45:44] [6512] Alarm signal received, checking for inactive children...
[2025-04-08 21:45:44] [6512] Child process 6514 has been inactive for too long
[2025-04-08 21:45:44] [6512] Child process 6514 terminated with status 0
[2025-04-08 21:45:44] [6512] Daemon process 6512 terminated with status 0
cbolat@cbolat:~/System-Programming/HW02$ cat /tmp/daemon.log
```

```
cbolat@cbolat:~/System-Programming/HW02$ valgrind ./ipc 23 54
==4832== Memcheck, a memory error detector
==4832== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4832== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==4832== Command: ./ipc 23 54
==4832==
==4832==
==4832== HEAP SUMMARY:
==4832==     in use at exit: 0 bytes in 0 blocks
==4832==   total heap usage: 10 allocs, 10 frees, 5,869 bytes allocated
==4832==
==4832== All heap blocks were freed -- no leaks are possible
==4832==
==4832== For lists of detected and suppressed errors, rerun with: -s
==4832== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cbolat@cbolat:~/System-Programming/HW02$ ==4833==
==4833== HEAP SUMMARY:
==4833==     in use at exit: 0 bytes in 0 blocks
==4833==   total heap usage: 11 allocs, 11 frees, 5,884 bytes allocated
==4833==
==4833== All heap blocks were freed -- no leaks are possible
==4833==
==4833== For lists of detected and suppressed errors, rerun with: -s
==4833== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4836==
==4836== HEAP SUMMARY:
==4836==     in use at exit: 0 bytes in 0 blocks
==4836==   total heap usage: 17 allocs, 17 frees, 5,974 bytes allocated
==4836==
==4836== All heap blocks were freed -- no leaks are possible
==4836==
==4836== For lists of detected and suppressed errors, rerun with: -s
==4836== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4879==
==4879== HEAP SUMMARY:
==4879==     in use at exit: 0 bytes in 0 blocks
==4879==   total heap usage: 23 allocs, 23 frees, 6,064 bytes allocated
==4879==
==4879== All heap blocks were freed -- no leaks are possible
==4879==
==4879== For lists of detected and suppressed errors, rerun with: -s
==4879== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4834==
==4834== HEAP SUMMARY:
==4834==     in use at exit: 0 bytes in 0 blocks
==4834==   total heap usage: 34 allocs, 34 frees, 6,229 bytes allocated
==4834==
==4834== All heap blocks were freed -- no leaks are possible
==4834==
==4834== For lists of detected and suppressed errors, rerun with: -s
==4834== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

ALL PROCESS TERMINATION STATUS IS LOGGED AS SEEN FROM OUTPUTS

ZOMBIE PROTECTION IS DONE WITH SIG_CHLD handler

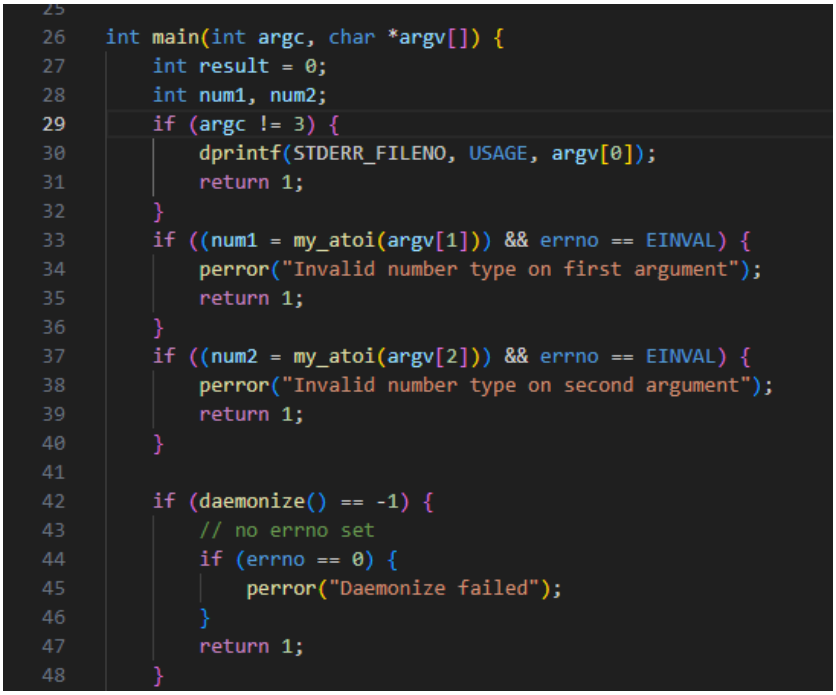


```
16
17  void handle_sigchld(int sig) {
18      int status;
19      pid_t pid;
20      (void)sig; // Avoid unused parameter warning
21      while ((pid = waitpid(-1, &status, WNOHANG)) > 0) // WHOHANG: return immediately if no child has exited
22      {
23          child_count += 1;
24          char message[100];
25          sprintf(message, "Child process %d terminated with status %d", pid, WEXITSTATUS(status));
26          for (int i = 0; i < child_count_tracked; i++) {
27              if (child_pids[i] == pid) {
28                  child_pids[i] = -1; // Mark as terminated
29                  break;
30              }
31          }
32          write_log(message, 0);
33      }
34  }
```

Project Development Steps:

The development of the project was carried out step by step, focusing on modularity, fault tolerance, and adherence to system programming principles. The main function served as the entry point and orchestrated the overall control flow of the application.

The program begins by checking the validity of the command-line arguments. It expects two integers, and these inputs are parsed using a custom my_atoi function, which also handles input validation and sets appropriate error flags in case of incorrect formats. If the input is invalid, a corresponding error message is displayed, and the program terminates gracefully.

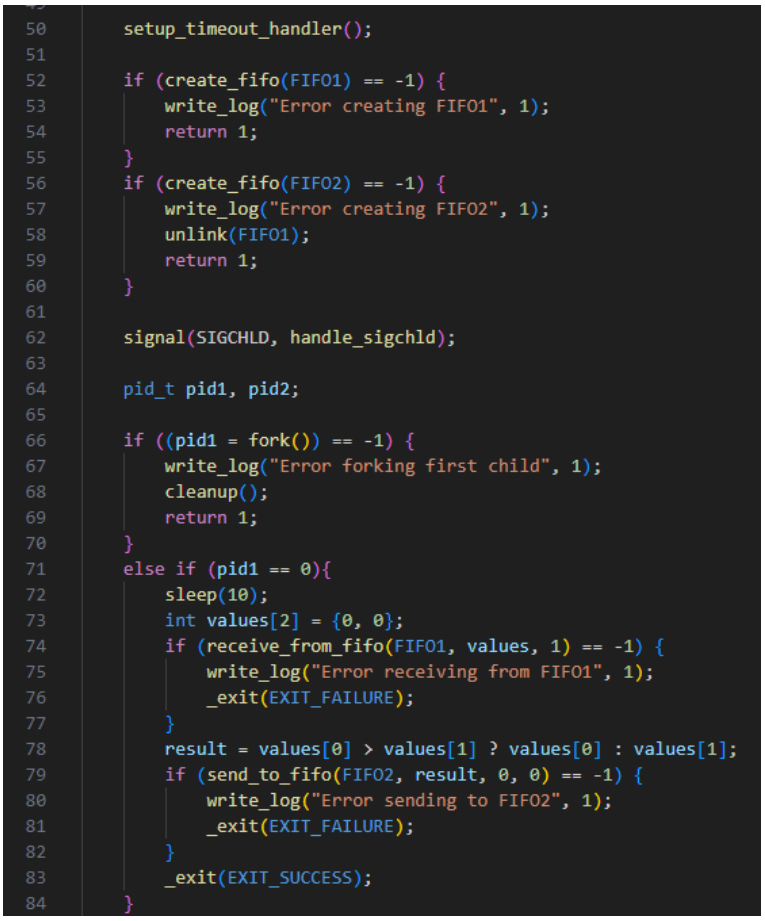


```
25
26  int main(int argc, char *argv[]) {
27      int result = 0;
28      int num1, num2;
29      if (argc != 3) {
30          dprintf(STDERR_FILENO, USAGE, argv[0]);
31          return 1;
32      }
33      if ((num1 = my_atoi(argv[1])) && errno == EINVAL) {
34          perror("Invalid number type on first argument");
35          return 1;
36      }
37      if ((num2 = my_atoi(argv[2])) && errno == EINVAL) {
38          perror("Invalid number type on second argument");
39          return 1;
40      }
41
42      if (daemonize() == -1) {
43          // no errno set
44          if (errno == 0) {
45              perror("Daemonize failed");
46          }
47          return 1;
48      }
```

Once valid input is ensured, the process attempts to transform itself into a daemon using the daemonize() function. This step detaches the process from the controlling terminal and allows it to run in the background. If daemonization fails, the program reports the error and exits.

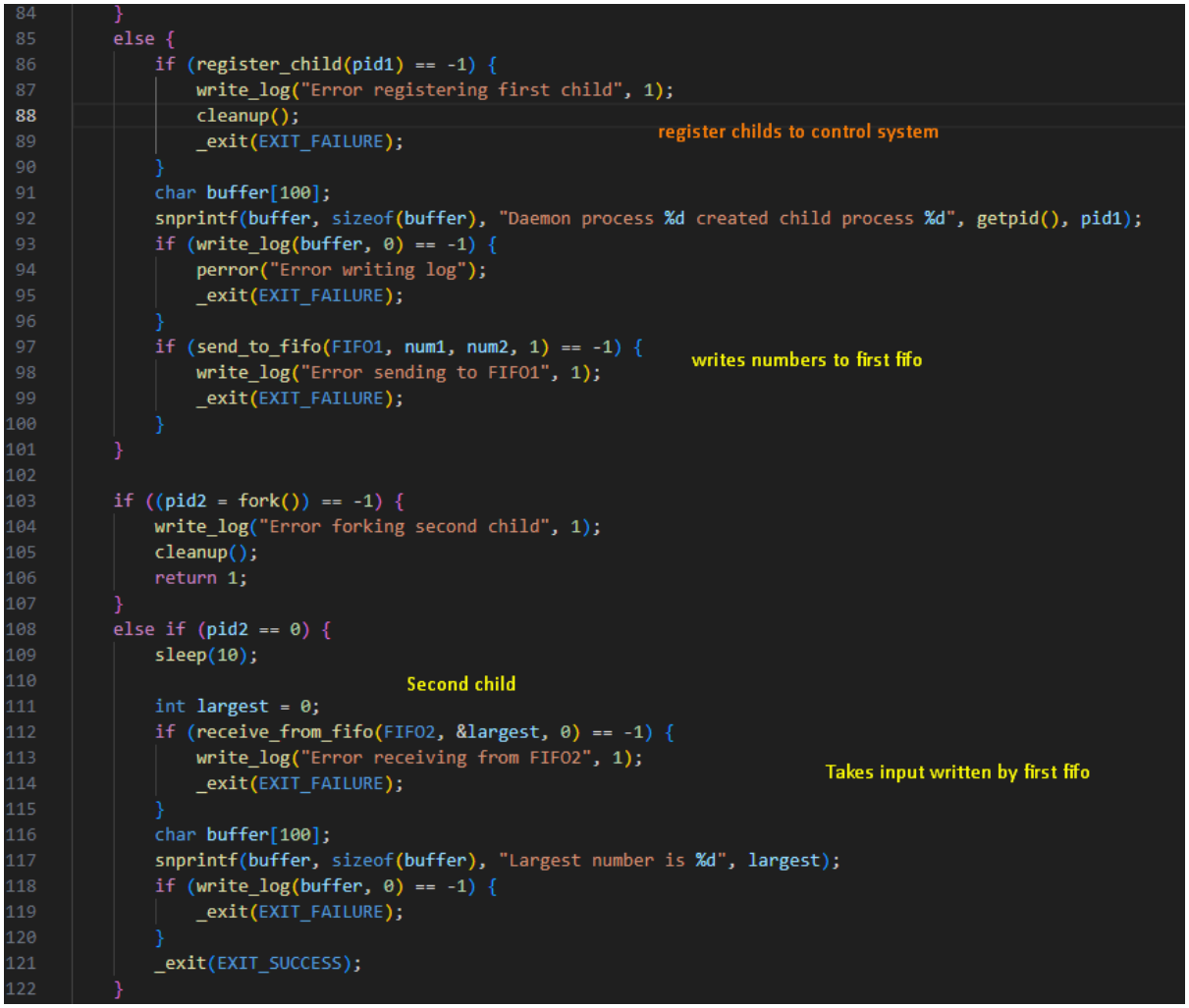
The daemon then sets up a timeout signal handler to prevent indefinite execution and proceeds to create two named pipes (FIFOs) for inter-process communication. These FIFOs (FIFO1 and FIFO2) are used to transmit data between the child processes. If either FIFO fails to be created, the daemon logs the error and performs cleanup to avoid resource leakage.

Following the IPC setup, two child processes are forked. The first child is responsible for reading the two integers from FIFO1, comparing them, and writing the larger one to FIFO2. The second child then reads the result from FIFO2 and logs the outcome, indicating which number was larger. Both child processes are registered and tracked by the parent daemon using a register_child function to monitor their lifecycle.



```
50      setup_timeout_handler();
51
52      if (create_fifo(FIFO1) == -1) {
53          write_log("Error creating FIFO1", 1);
54          return 1;
55      }
56      if (create_fifo(FIFO2) == -1) {
57          write_log("Error creating FIFO2", 1);
58          unlink(FIFO1);
59          return 1;
60      }
61
62      signal(SIGCHLD, handle_sigchld);
63
64      pid_t pid1, pid2;
65
66      if ((pid1 = fork()) == -1) {
67          write_log("Error forking first child", 1);
68          cleanup();
69          return 1;
70      }
71      else if (pid1 == 0){
72          sleep(10);
73          int values[2] = {0, 0};
74          if (receive_from_fifo(FIFO1, values, 1) == -1) {
75              write_log("Error receiving from FIFO1", 1);
76              _exit(EXIT_FAILURE);
77          }
78          result = values[0] > values[1] ? values[0] : values[1];
79          if (send_to_fifo(FIFO2, result, 0, 0) == -1) {
80              write_log("Error sending to FIFO2", 1);
81              _exit(EXIT_FAILURE);
82          }
83          _exit(EXIT_SUCCESS);
84      }
```

Daemon sets the timeout mechanism and create first child to FIFO1




```
    }
    else {
        if (register_child(pid1) == -1) {
            write_log("Error registering first child", 1);
            cleanup();
            _exit(EXIT_FAILURE);
        }
        char buffer[100];
        snprintf(buffer, sizeof(buffer), "Daemon process %d created child process %d", getpid(), pid1);
        if (write_log(buffer, 0) == -1) {
            perror("Error writing log");
            _exit(EXIT_FAILURE);
        }
        if (send_to_fifo(FIFO1, num1, num2, 1) == -1) {
            write_log("Error sending to FIFO1", 1);
            _exit(EXIT_FAILURE);
        }
    }

    if ((pid2 = fork()) == -1) {
        write_log("Error forking second child", 1);
        cleanup();
        return 1;
    }
    else if (pid2 == 0) {
        sleep(10);
                                Second child
        int largest = 0;
        if (receive_from_fifo(FIFO2, &largest, 0) == -1) {
            write_log("Error receiving from FIFO2", 1);
            _exit(EXIT_FAILURE);
        }
        char buffer[100];
        snprintf(buffer, sizeof(buffer), "Largest number is %d", largest);
        if (write_log(buffer, 0) == -1) {
            _exit(EXIT_FAILURE);
        }
        _exit(EXIT_SUCCESS);
    }
```

FIFO2 reads largest number and logs to “/tmp/deamon.log” file

The parent (daemon) process continues running in the background, logging the status periodically with a “Proceeding…” message until both child processes have terminated. Once this condition is met, it performs a final cleanup routine by unlinking the FIFOs and logging its termination status.

```
    else {
        if (register_child(pid2) == -1) {
            write_log("Error registering second child", 1);
            cleanup();
            _exit(EXIT_FAILURE);
        }
        char buffer[100];
        snprintf(buffer, sizeof(buffer), "Daemon process %d created child process %d", getpid(), pid2);
        if (write_log(buffer, 0) == -1) {
            perror("Error writing log");
            _exit(EXIT_FAILURE);
        }
        while (child_count < 2){
            write_log("Proceeding...", 0);

            sleep(2); // Sleep for 2 seconds
        }
        cleanup();

        char buffer1[100];
        snprintf(buffer1, sizeof(buffer1), "Daemon process %d terminated with status 0", getpid());
        if (write_log(buffer1, 0) == -1) {
            perror("Error writing log");
        }
        _exit(EXIT_SUCCESS);
    }
    return 0;
}
```

This structured approach ensured the development remained focused on reliability, signal handling, and smooth communication between processes. Each stage of the project builds upon the previous one, leading to a robust system capable of managing its resources and gracefully recovering from errors.

FIFO OPERATIONS

1. int create_fifo(const char *fifo)

This function is responsible for **creating a named pipe (FIFO)**, which is used for inter-process communication between the parent and child processes.

- First, it checks whether the FIFO already exists using access().
- If it does exist, it removes the existing FIFO using unlink() to prevent conflicts.
- If the FIFO does **not exist**, it checks whether the error is something **other than "file not found"**, and if so, returns an error.
- It resets errno to zero and tries to create a new FIFO with mkfifo() and permission 0666, which allows both reading and writing for all users.
- If creation is successful, it logs a message stating that the FIFO was created.

This function ensures that FIFO creation is **clean, non-blocking, and properly logged**, which is essential for reliable IPC setup.

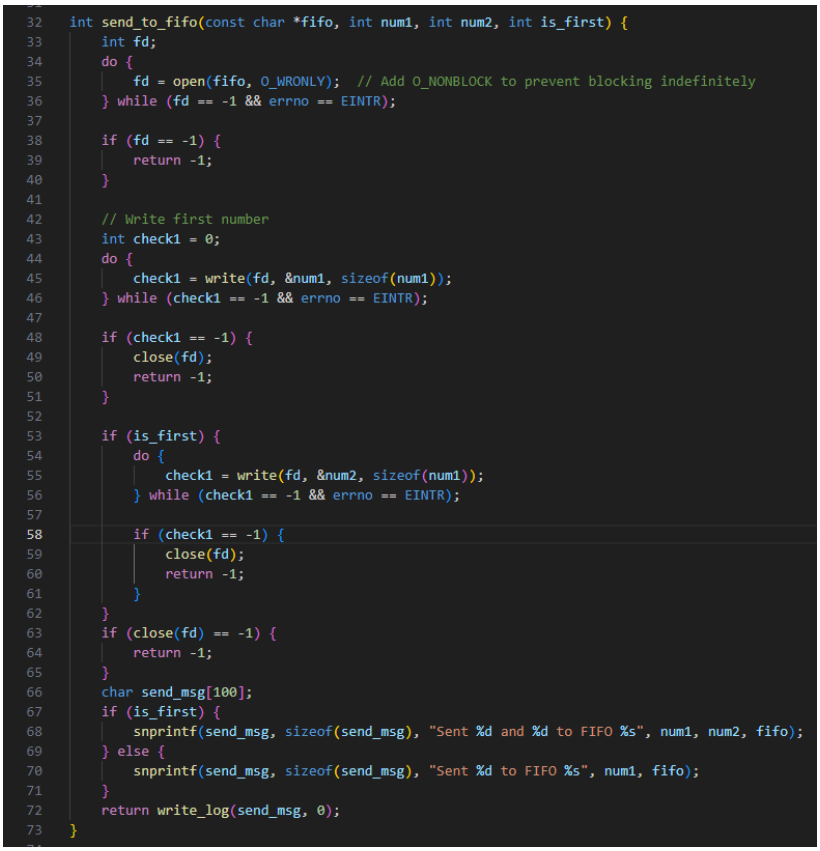
```
13  int create_fifo(const char *fifo) {
14      // If fifo exists unlink it first
15      if (access(fifo, F_OK) != -1) { // FIFO exists
16          if (unlink(fifo) == -1) {
17              return -1;
18          }
19      } else if (errno != ENOENT) { // Error other than "no such file or directory"
20          return -1;
21      }
22      errno = 0; // Reset errno before creating FIFO
23      if (mkfifo(fifo, 0666) == -1) {
24          return -1;
25      }
26      char create_msg[100];
27      snprintf(create_msg, sizeof(create_msg), "FIFO %s created", fifo);
28      return write_log(create_msg, 0);
29
30  }
```

2. int send_to_fifo(const char *fifo, int num1, int num2, int is_first)

This function handles **sending data (integers)** to the FIFO.

- It tries to open the FIFO for writing. If the open call is interrupted by a signal (EINTR), it retries until it succeeds or fails with another error.
- If it fails to open, it returns an error.
- Once the FIFO is opened successfully, it writes the first integer using write(). Again, it retries if interrupted.
- If is_first is set to 1, it indicates that this is the first stage of communication, and therefore it writes both num1 and num2.
- After writing, it closes the FIFO descriptor and logs a message indicating which values were sent and to which FIFO.

This function ensures that writing to the FIFO is done **safely**, with proper retry logic and logging, avoiding deadlocks or silent failures.



```c
int send_to_fifo(const char *fifo, int num1, int num2, int is_first) {
    int fd;
    do {
        fd = open(fifo, O_WRONLY);  // Add O_NONBLOCK to prevent blocking indefinitely
    } while (fd == -1 && errno == EINTR);

    if (fd == -1) {
        return -1;
    }

    // Write first number
    int check1 = 0;
    do {
        check1 = write(fd, &num1, sizeof(num1));
    } while (check1 == -1 && errno == EINTR);

    if (check1 == -1) {
        close(fd);
        return -1;
    }

    if (is_first) {
        do {
            check1 = write(fd, &num2, sizeof(num1));
        } while (check1 == -1 && errno == EINTR);

        if (check1 == -1) {
            close(fd);
            return -1;
        }
    }
    if (close(fd) == -1) {
        return -1;
    }
    char send_msg[100];
    if (is_first) {
        snprintf(send_msg, sizeof(send_msg), "Sent %d and %d to FIFO %s", num1, num2, fifo);
    } else {
        snprintf(send_msg, sizeof(send_msg), "Sent %d to FIFO %s", num1, fifo);
    }
    return write_log(send_msg, 0);
}
```

3. int receive_from_fifo(const char *fifo, int *values, int is_first)

This function handles **reading values** from the FIFO into a buffer.

- It attempts to open the FIFO in **non-blocking read mode**. Like before, it retries if the open call is interrupted.
- Once the FIFO is open, it begins reading integers. The number of integers it expects depends on the is_first flag:
  - If is_first is 1, it reads **two integers**.
  - Otherwise, it reads **one integer**.
- If the FIFO is empty (EAGAIN), it waits briefly (usleep(1000)) before retrying.
- If reading is successful, the read values are stored in the provided array (values), and the FIFO is closed.
- A log message is generated indicating what was received from the FIFO.

This function is robust against interruptions and empty FIFO buffers, ensuring that data is **read reliably and without data loss**.

FUNCTION IN NEXT PAGE

```c
int receive_from_fifo(const char *fifo, int *values, int is_first) {
    int fd;
    do {
        // Wait for FIFO to be ready
        fd = open(fifo, O_RDONLY | O_NONBLOCK);
    } while (fd == -1 && errno == EINTR);

    if (fd == -1) {
        return -1;
    }
    // Read byte by byte
    int num;
    int i = 0;
    int num_to_read = is_first ? 2 : 1;
    do {
        int read_byte = read(fd, &num, sizeof(int));
        if (read_byte == -1) {
            if (errno == EAGAIN) {
                // FIFO is empty wait for a while and try again
                usleep(1000);
                errno = 0; // Reset errno before retrying
                continue;
            }
            if (errno == EINTR) {
                // Interrupted by a signal, retry
                continue;
            }
            int err = errno;
            if (close(fd) == -1) {
                errno = err;
                return -1;
            }
            return -1;
        }
        else if (read_byte == 0){
            continue; // No data read, continue looping timeout mechanism around
        }
        values[i] = num;
        i++;
    } while (i < num_to_read);
    if (close(fd) == -1) {
        return -1;
    }
    char receive_msg[100];
    if (is_first) {snprintf(receive_msg, sizeof(receive_msg), "Received %d and %d from FIFO %s", values[0], values[1], fifo);}
    else { snprintf(receive_msg, sizeof(receive_msg), "Received %d from FIFO %s", values[0], fifo);}
    return write_log(receive_msg, 0);
}
```

Daemonizing Operations

The `daemonize` function is responsible for transforming a running process into a daemon—a background process that runs independently of a terminal and user session. It starts by opening a log file where the daemon's output and errors will be written. If the log file cannot be opened, the function logs an error message and returns `-1`. After that, it sets up signal handlers for specific signals: `SIGUSR1`, `SIGTERM`, and `SIGHUP`, ensuring that the daemon can handle these events appropriately.

The process then performs the first `fork()`, creating a child process. The parent process immediately exits to detach from the terminal, while the child process continues. This fork ensures that the daemon is no longer associated with the terminal session that launched it. The child process proceeds by calling `setsid()`, which creates a new session and makes the process the session leader. This step is crucial to ensure the daemon is completely detached from the terminal and not tied to any user session. Afterward, a second `fork()` is performed to further ensure that the daemon cannot reacquire a controlling terminal. The first child exits, and the second child continues, thus solidifying the daemon’s independence.

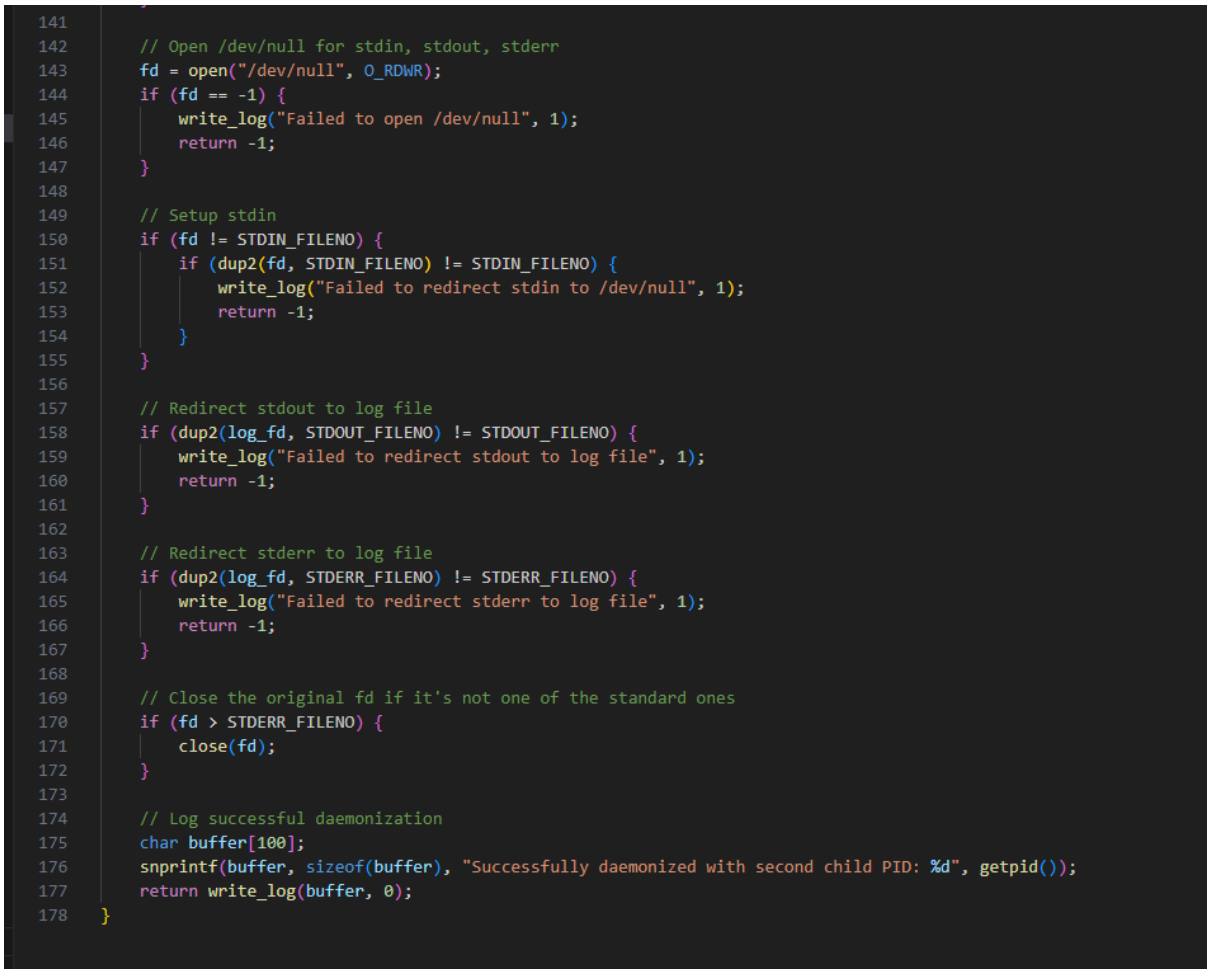
```c
int daemonize(void) {
    pid_t pid;
    int maxfd, fd;

    // Open log file // if exist clear it
    log_fd = open(LOG_FILE, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (log_fd == -1) {
        perror("Failed to open log file");
        return -1;
    }

    // Set up signal handlers
    signal(SIGUSR1, handle_sigusr1);
    signal(SIGTERM, handle_sigterm_and_signup);
    signal(SIGHUP, handle_sigterm_and_signup);

    // Log start of daemonization
    write_log("Starting daemonization process", 0);

    // First fork: detach from controlling terminal
    pid = fork();
    if (pid < 0) {
        write_log("First fork failed in daemonize()", 1);
        return -1;
    }
    if (pid > 0) {
        // Parent exits
        exit(EXIT_SUCCESS);
    } else {
        // print exit status of the parent
        char buffer[100];
        // Get parent process ID & exit status
        snprintf(buffer, sizeof(buffer), "Parent process exited while creating daemon with status: 0 PID: %d", getppid());
        write_log(buffer, 0);
    }

    // Child continues: become session leader
    if (setsid() == -1) {
        write_log("setsid failed in daemonize()", 1);
        return -1;
    }
```

Next, the function sets the file creation mask to `0` using `umask(0)`, giving the daemon full control over the permissions of any files it creates. This step is necessary for the proper functioning of the daemon, especially when creating log files or other resources. The daemon then changes its working directory to the root directory (`/`). This is done to avoid locking any directory, which could prevent filesystem unmounting. The working directory change ensures that the daemon can safely operate without affecting system directories.

```
    // Second fork: prevent acquiring a controlling terminal
    pid = fork();
    if (pid < 0) {
        write_log("Second fork failed in daemonize()", 1);
        return -1;
    }
    if (pid > 0) {
        // First child exits
        exit(EXIT_SUCCESS);
    }
    else {
        // print exit status of the first child
        char buffer[100];
        // Get first child process ID & exit status
        snprintf(buffer, sizeof(buffer), "First child exited while creating daemon with status: 0 PID: %d", getppid());
        write_log(buffer, 0);
    }
    // Final daemon process continues

    // Set file creation mask
    umask(0);

    // Change current directory to root
    if (chdir("/") == -1) {
        write_log("chdir failed in daemonize()", 1);
        return -1;
    }

    // Close all open file descriptors except log
    maxfd = sysconf(_SC_OPEN_MAX);
    if (maxfd == -1) {
        write_log("sysconf failed in daemonize()", 1);
        maxfd = 1024; // Fallback if sysconf fails
    }

    for (fd = 0; fd < maxfd; fd++) {
        if (fd != log_fd) {
            close(fd);
        }
    }
```

After setting up the directory, the function proceeds to close all open file descriptors, except for the log file. This includes the standard input, output, and error streams. Closing unnecessary file descriptors prevents resource leaks and ensures the daemon doesn't interact with the terminal. The function then redirects the standard input to `/dev/null` (to discard any input), and both standard output and error are redirected to the log file. This step ensures the daemon no longer expects user input and that its output and errors are logged for later review.

Finally, the function performs any final cleanup by closing any remaining file descriptors, ensuring that the daemon operates without unnecessary resources being held. It logs a success message indicating that the daemon has been successfully created and returns, completing the daemonization process. The result is a fully operational daemon that runs independently of user sessions and terminal control.



```
141
142     // Open /dev/null for stdin, stdout, stderr
143     fd = open("/dev/null", O_RDWR);
144     if (fd == -1) {
145         write_log("Failed to open /dev/null", 1);
146         return -1;
147     }
148
149     // Setup stdin
150     if (fd != STDIN_FILENO) {
151         if (dup2(fd, STDIN_FILENO) != STDIN_FILENO) {
152             write_log("Failed to redirect stdin to /dev/null", 1);
153             return -1;
154         }
155     }
156
157     // Redirect stdout to log file
158     if (dup2(log_fd, STDOUT_FILENO) != STDOUT_FILENO) {
159         write_log("Failed to redirect stdout to log file", 1);
160         return -1;
161     }
162
163     // Redirect stderr to log file
164     if (dup2(log_fd, STDERR_FILENO) != STDERR_FILENO) {
165         write_log("Failed to redirect stderr to log file", 1);
166         return -1;
167     }
168
169     // Close the original fd if it's not one of the standard ones
170     if (fd > STDERR_FILENO) {
171         close(fd);
172     }
173
174     // Log successful daemonization
175     char buffer[100];
176     snprintf(buffer, sizeof(buffer), "Successfully daemonized with second child PID: %d", getpid());
177     return write_log(buffer, 0);
178 }
```

Timeout mechanism

1. int register_child(pid_t pid)

This function is used to **register a newly created child process**.

- If the number of registered children is less than 2:
  - The child PID is stored.
  - The current timestamp (time(NULL)) is saved as the child's **last active time**.
  - The count of registered children is incremented.
  - Returns 0 on success.
- If the maximum number of children (2) has been reached:
  - Logs an error and returns -1.

This ensures the daemon is tracking each child's creation and activity timestamp properly.

```
11  int child_pids[2]; // Array to store child process IDs
12  time_t child_last_active[2]; // Array to store last active time of child processes
13  int child_count_tracked = 0; // Variable to track the number of child processes
14
15  int register_child(pid_t pid){
16      if (child_count_tracked < 2){
17          child_pids[child_count_tracked] = pid;
18          time_t now = time(NULL);
19          if (now == -1) {
20              write_log("Failed to get current time", 1);
21              return -1; // Failure
22          }
23          child_last_active[child_count_tracked] = now;
24          child_count_tracked++;
25          return 0; // Success
26      } else {
27          write_log("Maximum number of child processes reached", 1);
28          return -1; // Failure
29      }
30  }
```

2. int check_inactive_children()

This function checks whether **any child process has been inactive for too long**, based on a predefined timeout constant (CHILD_TIMEOUT).

- It fetches the **current time**.
- Iterates through the registered child processes:
    - For each, it checks the time elapsed since the last active timestamp using difftime().
    - If the child has exceeded the timeout:
        - A log is created marking the child as inactive.
        - Inactive count is incremented.
- If **any** inactive child is detected:
    - All child processes are terminated using kill(pid, SIGUSR1).
    - Logs are generated for each termination.
    - A message is logged that the daemon will terminate as well.
    - Calls cleanup() and exits the process with _exit(EXIT_FAILURE).

This provides an automatic **watchdog mechanism**, ensuring that unresponsive child processes are detected and handled cleanly.

FUNCTION IN NEXT PAGE

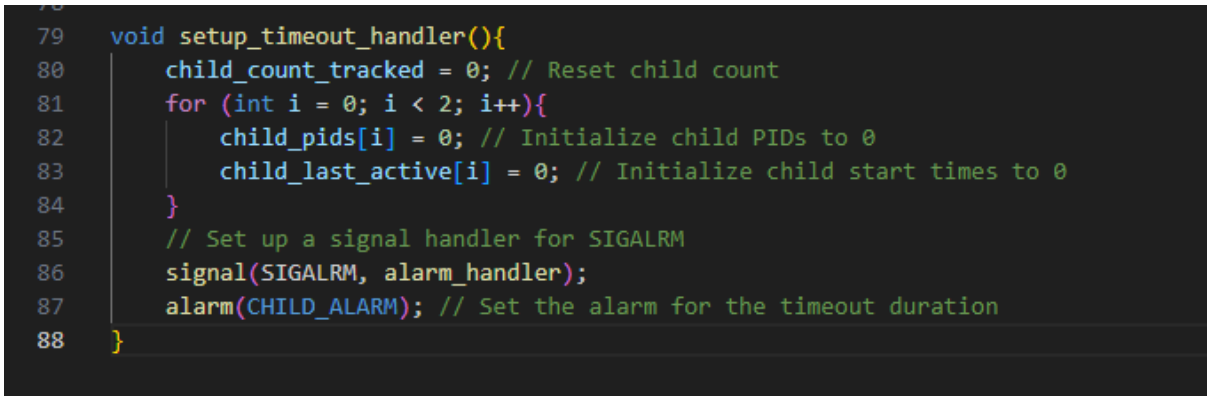
```c
int check_inactive_children(){
    time_t now = time(NULL);
    int inactive_count = 0;
    if (now == -1) {
        write_log("Failed to get current time", 1);
        return -1; // Failure
    }
    for (int i = 0; i < child_count_tracked; i++){
        if (child_pids[i] != 0 && child_pids[i] != -1 && difftime(now, child_last_active[i]) > CHILD_TIMEOUT){
            char message[100];
            snprintf(message, sizeof(message), "Child process %d has been inactive for too long", child_pids[i]);
            if (write_log(message, 0) == -1) {
                return -1; // Failure
            }
            inactive_count++;
        }
    }
    if (inactive_count != 0){
        // Kill all child processes..
        for (int i = 0; i < child_count_tracked; i++){
            if (child_pids[i] != 0 && child_pids[i] != -1){
                if (kill(child_pids[i], SIGUSR1) == -1){
                    if (errno == ESRCH){ // No such process
                        continue; // Process already terminated
                    } else {
                        // Error occurred while trying to kill the process
                        char error_message[100];
                        snprintf(error_message, sizeof(error_message), "Failed to terminate child process %d", child_pids[i]);
                        if (write_log(error_message, 1) == -1) {
                            return -1; // Failure
                        }
                    }
                } else {
                }
            }
        }

        sleep(5); // Wait for 5 seconds before checking again
        char buffer1[100];
        snprintf(buffer1, sizeof(buffer1), "Daemon process %d terminated with status 0", getpid());
        if (write_log(buffer1, 0) == -1) {
            perror("Error writing log");
        }
        _exit(EXIT_FAILURE); // Exit the daemon process
    }
    return 0; // Success
}
```

4. void setup_timeout_handler()

This function sets up the **timeout system and signal handler**.

- It resets all tracking variables to their initial state.
- Sets the signal handler for SIGALRM to a function called alarm_handler (assumed to be defined elsewhere).
- Sets an alarm using alarm(CHILD_ALARM), which will trigger the SIGALRM signal after a set number of seconds.

This initializes the entire timeout-checking mechanism, letting the daemon **periodically validate child activity** and take action if needed.

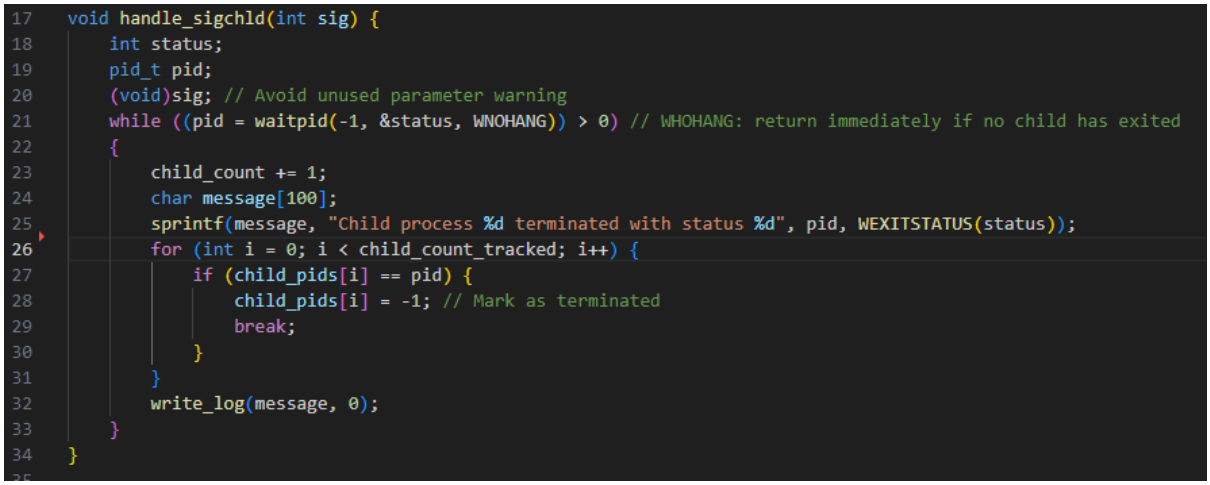


```
79  void setup_timeout_handler(){
80      child_count_tracked = 0; // Reset child count
81      for (int i = 0; i < 2; i++){
82          child_pids[i] = 0; // Initialize child PIDs to 0
83          child_last_active[i] = 0; // Initialize child start times to 0
84      }
85      // Set up a signal handler for SIGALRM
86      signal(SIGALRM, alarm_handler);
87      alarm(CHILD_ALARM); // Set the alarm for the timeout duration
88  }
```

SIGNALS:

1. handle_sigchld(int sig):

- **Purpose**: Handles the SIGCHLD signal, which is sent when a child process terminates.
- **How It Works**: The function uses waitpid() in non-blocking mode (WNOHANG) to check for any child processes that have exited. When a child process terminates, the function retrieves the child's exit status and logs the termination details. This helps keep track of the daemon's child processes and ensures proper cleanup and monitoring of child process states.

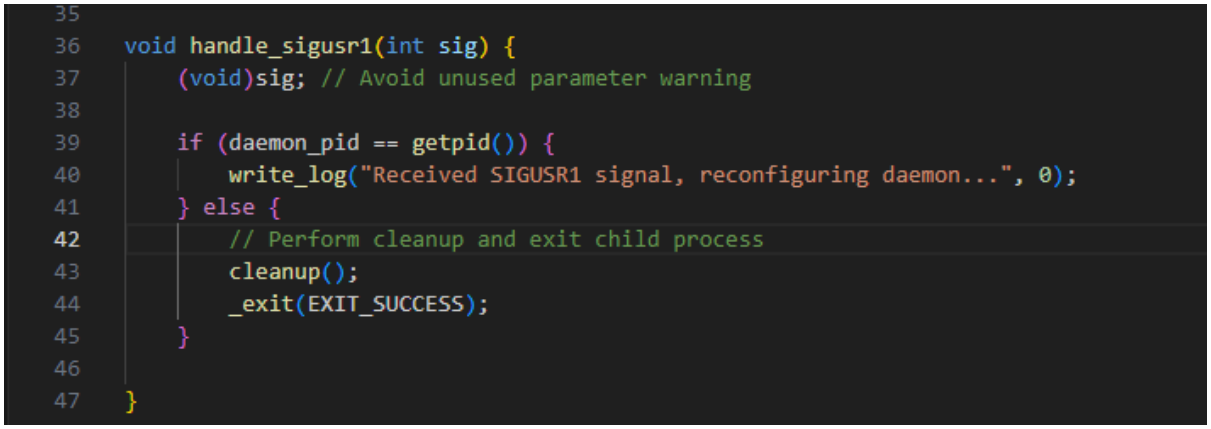


```
17  void handle_sigchld(int sig) {
18      int status;
19      pid_t pid;
20      (void)sig; // Avoid unused parameter warning
21      while ((pid = waitpid(-1, &status, WNOHANG)) > 0) // WHOHANG: return immediately if no child has exited
22      {
23          child_count += 1;
24          char message[100];
25          sprintf(message, "Child process %d terminated with status %d", pid, WEXITSTATUS(status));
26          for (int i = 0; i < child_count_tracked; i++) {
27              if (child_pids[i] == pid) {
28                  child_pids[i] = -1; // Mark as terminated
29                  break;
30              }
31          }
32          write_log(message, 0);
33      }
34  }
```

2. handle_sigusr1 (int sig):

This function handles the SIGUSR1 signal.

- If the current process is the main daemon, it logs a message about reconfiguration.
- If it's a child process, it cleans up and exits**.**



```
35
36  void handle_sigusr1(int sig) {
37      (void)sig; // Avoid unused parameter warning
38
39      if (daemon_pid == getpid()) {
40          write_log("Received SIGUSR1 signal, reconfiguring daemon...", 0);
41      } else {
42          // Perform cleanup and exit child process
43          cleanup();
44          _exit(EXIT_SUCCESS);
45      }
46
47  }
```

3. void handle_sigterm_and_signup(int sig)

This function handles SIGTERM or SIGHUP:

- If SIGTERM:
  - The **main process** logs a termination message.
  - A **child process** logs, cleans up, and exits.
- If SIGHUP:
  - The daemon logs and prepares for restart.

Then, it:

- Sends SIGUSR1 to all children.
- If they don’t exit, it sends SIGKILL.
- Waits for 5 seconds.
- Logs everything, cleans up, and exits.

4. void alarm_handler **(int sig)**

This handles the SIGALRM signal (from alarm()).

- Logs a message.
- Calls check_inactive_children() to see if any children are inactive.
- Sets another alarm to repeat the check later.

FUNCTION IN NEXT PAGE

```c
void handle_sigterm_and_signup(int sig) {
    // Perform cleanup and exit
    if (sig == SIGTERM) {
        if (daemon_pid == getpid()) {
            write_log("Received SIGTERM signal, terminating daemon...", 0);
        } else {
            char message[100];
            sprintf(message, "Received SIGTERM signal, terminating child process %d...", getpid());
            write_log(message, 0);
            cleanup();
            _exit(EXIT_SUCCESS);
        }
    } else if (sig == SIGHUP) {
        write_log("Received SIGHUP signal, terminating daemon...", 0);
    }
    for (int i = 0; i < child_count_tracked; i++) {
        if (child_pids[i] > 0) {
            if (kill(child_pids[i], SIGUSR1) == -1) {
                write_log("Sending SIGUSR1 to child that didn't terminate gracefully", 1);
                if (kill(child_pids[i], SIGKILL) == -1) {
                    write_log("Failed to kill child process", 1);
                } else {
                    write_log("Child process killed successfully", 0);
                }
            } else {
                child_pids[i] = -1; // Mark as terminated
            }
        }
    }

    sleep(5); // Wait for children to terminate

    write_log("All child processes terminated", 0);
    write_log("Daemon process terminated", 0);
    cleanup();


    _exit(EXIT_SUCCESS);
}

void alarm_handler(int sig) {
    (void)sig; // Avoid unused parameter warning
    write_log("Alarm signal received, checking for inactive children...", 0);
    if (check_inactive_children() == -1) {
        write_log("Error checking for inactive children", 1);
    }
    alarm(CHILD_ALARM); // Reset the alarm
}
```