Interprocess Communication Assignment Report

How To Run

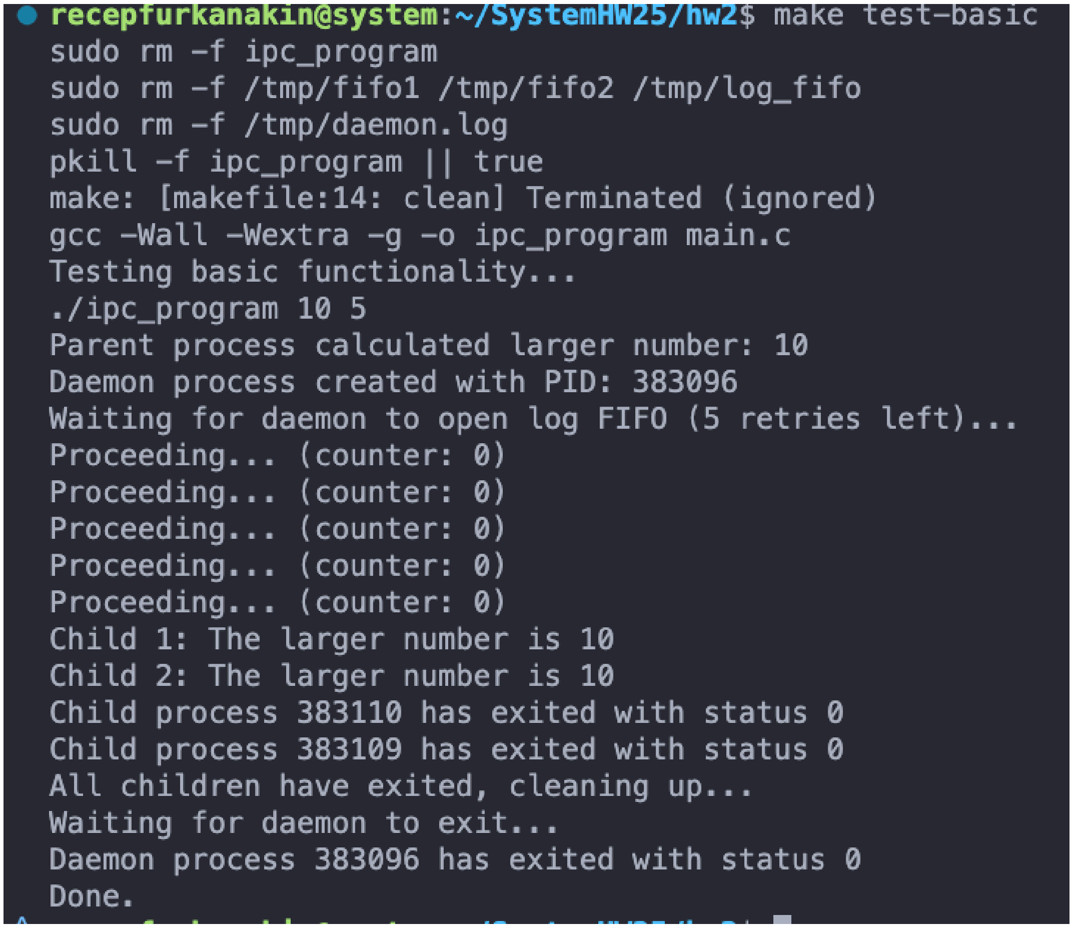
```
make #For compiling the program

make test #For running comprehensive test

make test-basic #For running basic test

make test-memory #For running memory leak test with valgrind
```

Basic Run Screenshot



```
recepfurkanakin@system:~/SystemHW25/hw2$ make test-basic
sudo rm -f ipc_program
sudo rm -f /tmp/fifo1 /tmp/fifo2 /tmp/log_fifo
sudo rm -f /tmp/daemon.log
pkill -f ipc_program || true
make: [makefile:14: clean] Terminated (ignored)
gcc -Wall -Wextra -g -o ipc_program main.c
Testing basic functionality...
./ipc_program 10 5
Parent process calculated larger number: 10
Daemon process created with PID: 383096
Waiting for daemon to open log FIFO (5 retries left)...
Proceeding... (counter: 0)
Proceeding... (counter: 0)
Proceeding... (counter: 0)
Proceeding... (counter: 0)
Proceeding... (counter: 0)
Child 1: The larger number is 10
Child 2: The larger number is 10
Child process 383110 has exited with status 0
Child process 383109 has exited with status 0
All children have exited, cleaning up...
Waiting for daemon to exit...
Daemon process 383096 has exited with status 0
Done.
```

Communication Mechanisms

The system uses three main IPC (Inter-Process Communication) mechanisms:

1. **Named Pipes (FIFOs)**: For data transfer between processes
   - FIFO1: Parent → Child1 (sends two integers)
   - FIFO2: Child1 → Child2 (sends the larger integer)
   - LOG_FIFO: All processes → Daemon (for logging)
2. **Signals**: For process coordination and cleanup
   - SIGCHLD: Notifies parent when children terminate
   - SIGTERM/SIGINT: For graceful daemon termination
   - SIGHUP: For daemon reconfiguration (not fully implemented)
3. **Files**: For persistent logging
   - DAEMON_LOG: Output file for daemon logging

Program Flow

1. **Initialization**:
   - Parent process parses two integers from command line arguments
   - Creates necessary FIFOs and initializes signal handlers
   - Forks a daemon process for logging
2. **Process Creation**:
   - Parent creates two child processes via fork()
   - Each child has specific responsibilities in the pipeline
3. **Data Flow**:
   - Parent writes two integers to FIFO1
   - Child1 reads these integers, finds the maximum, and writes it to FIFO2
   - Child2 reads the maximum from FIFO2 and displays it
4. **Termination and Cleanup**:
   - Parent monitors child termination via SIGCHLD handler
   - When all children exit, parent sends SIGTERM to daemon
   - Parent removes all FIFOs and releases resources

Key Technical Features

1. **Signal Handling**:
   - Uses sigaction() for reliable signal handling
   - SIGCHLD handler to detect child termination
   - Custom handlers for daemon termination
2. **Non-blocking I/O**:
   - Daemon uses non-blocking reads to avoid hanging

    - Allows for responsive signal handling

3. **Robust Error Handling**:
    - Graceful handling of fork failures
    - FIFO creation error detection
    - Proper resource cleanup on errors
4. **Timeout Mechanisms**:
    - Alarm signal for daemon safety
    - Retry mechanisms for FIFO opening
    - Forced termination (SIGKILL) as last resort
5. **Synchronization**:
    - Sleep statements to ensure proper sequencing
    - Signal-based coordination

Bonus Parts

Zombie Process Protection

The code prevents zombie processes using several mechanisms:

1. **SIGCHLD Signal Handler**:
    - The most important zombie prevention mechanism is the `sigchld_handler` function
    - This handler is triggered whenever a child process terminates
    - It uses `waitpid(-1, &status, WNOHANG)` in a loop to reap all terminated children without blocking
    - By calling `waitpid()`, it collects the exit status and removes the process from the system table

```
void sigchld_handler(int sig __attribute__((unused)))
{
    pid_t pid;
    int status;

    // Use waitpid with WNOHANG to avoid blocking
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
    {
        // Skip counting the daemon to avoid early termination
        if (pid == daemon_pid)
        {
            printf("Daemon process %d has exited with status %d\\n", pid, WEXITSTATUS(status));
            daemon_pid = 0; // Mark daemon as handled
            continue;
        }

        printf("Child process %d has exited with status %d\\n", pid, WEXITSTATUS(status));
        child_counter += 2;
    }
}
```

1. **Signal Handler Registration**:
    - The handler is registered using `sigaction()` which is more reliable than `signal()`
    - Uses `SA_RESTART` flag to restart interrupted system calls
    - Uses `SA_NOCLDSTOP` to only trigger the handler when processes terminate, not when they stop
2. **Explicit Process Cleanup**:
    - In error cases, processes are killed explicitly with `kill()` signals
    - For the daemon process, there's a tiered termination approach (SIGTERM first, then SIGKILL)

Exit Status Reporting

The code reports exit statuses of all processes in several ways:

1. **Child Process Exit Status Reporting**:
    - In the SIGCHLD handler, `WEXITSTATUS(status)` extracts the exit code from the status value
    - This is printed along with the process ID: `printf("Child process %d has exited with status %d\\n", pid, WEXITSTATUS(status));`
2. **Daemon Process Exit Status**:
    - The daemon's exit status is specially handled and reported: `printf("Daemon process %d has exited with status %d\\n", pid, WEXITSTATUS(status));`
3. **Logging**:
    - Exit events are also written to the log FIFO, which the daemon process reads and adds timestamps to
    - The daemon itself logs its termination: `fprintf(stderr, "%s Daemon exiting cleanly\\n", time_str);`
4. **Graceful Termination**:
    - The parent process monitors the daemon's termination with this loop:

    ```
    for (int i = 0; i < 3; i++) {
        if (waitpid(daemon_pid, NULL, WNOHANG) == daemon_pid ||
            (kill(daemon_pid, 0) < 0 && errno == ESRCH)) {
            printf("Daemon has exited\\n");
            daemon_pid = 0;
            break;
        }
        sleep(1);
    }
    ```
5. **Timeout Mechanisms**:
    - The daemon has a 60-second alarm as a safety mechanism
    - The parent uses a 3-second timeout when waiting for the daemon to exit

These mechanisms ensure that no child process becomes a zombie process and that all process exit statuses are properly reported to the user and logged appropriately.

Author: Recep Furkan Akın

Date: April 8, 2025

Department of Computer Science and Engineering

System Programming - CSE 344