Lab 2: Interprocess Communications with Pipes and Java Threads

Liam Geraghty - 300356748
Shane Stock - XXXXXXXXX
CSI3131 - Operating Systems
2025-06-08

1 Introduction and Objectives

The objective of this lab is to explore interprocess communication (IPC) using UNIX/Linux pipes and to learn about Java threads and thread pools. The specific goals include:

- Understanding IPC mechanisms through the use of pipes in a C program.
- Gaining experience with process management in a Linux environment.
- Learning how to implement and manage threads in Java.
- Comparing the performance of individual threads versus a thread pool.

2 Methodology

The lab was conducted in a Linux virtual machine environment. The following steps were taken to achieve the objectives:

1. **Modify C Program**: Enhanced the `mon.c` program to create `mon2.c` for monitoring processes using pipes.
2. **Compilation**: Compiled the modified `mon2.c` program using gcc.
3. **Execution**: Ran the `mon2` program with the `calcloop` argument and observed the filtered output.

4. **Signal Handling**: Experimented with sending `SIGSTOP` and `SIGCONT` signals to the `calcloop` process.

5. **Java Setup**: Compiled the provided Java files for generating the Mandelbrot set.

6. **Execution of Mandelbrot**: Executed the `MandelBrot` application with various parameters.

7. **Thread Implementation**: Modified the Java code to use threads for rendering the Mandelbrot set.

8. **Thread Pool Implementation**: Further modified the code to utilize a thread pool with Executors.

3 Presentation and Analysis of Results

3.1 C Program Execution

The execution of the `mon2` program yielded filtered output for the `calcloop` process (Fig. 3). The following changes were made to the original `mon.c`:

1. Creation of `kill_process` function to gracefully terminate a process given a pid (line 9).

2. Creation of pipe `fd` for communication between `procmon` and `filter` (line 29).

3. Fork a new process to run `filter` (line 71).

4. Pipe standard output of `procmon` through `fd` (line 61) to standard input of `filter` (line 79) using `dup2()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

void kill_process(pid_t pid) {
       if (kill(pid, SIGTERM) == -1) {
```

```
                perror("Failed to kill process");
        }
}

/* the program execution starts here */
int main(int argc, char **argv)
{
    char   *program;
    pid_t  pid, procmon_pid, filter_pid;
    int fd[2];

    if (argc != 2) {
        printf("Usage: mon fileName\n where fileName is an
            executable file.\n");
        exit(-1);
    } else {
        program = argv[1];

        // Create pipe to pass messages
        if (pipe(fd) == -1) {
                fprintf(stderr, "Pipe failed");
                return -1;
        }

        // Fork a new process for program
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork failed\n");
                return 1;
        }
        else if (pid == 0) {
                // Execute program
                execl(program, program, (char*) NULL);
                perror("execl failed\n");
                exit(EXIT_FAILURE);
        }

        sleep(1);

        // Get string of program's pid to pass as cmd arg
        char pid_str[10];
        sprintf(pid_str, "%d", pid);
```

```
// Fork new process for procmon
procmon_pid = fork();
if(procmon_pid < 0) {
        fprintf(stderr, "Fork failed");
        kill_process(pid);
        return 1;
} else if (procmon_pid == 0) {
        close(fd[0]); // Close read end of pipe
        dup2(fd[1], STDOUT_FILENO);  // Redirect output
            of process to write end of pipe
        close(fd[1]); // Close write end of pipe

        // Execute procmon with pid_str as arg
        execl("./procmon", "procmon", pid_str, (char
            *)NULL);
        perror("execl for procmon failed\n");
        exit(EXIT_FAILURE);
}

// Fork new process for filter
filter_pid = fork();
if (filter_pid < 0) {
        fprintf(stderr, "Fork failed\n");
        kill_process(pid);
        kill_process(procmon_pid);
        return 1;
} else if (filter_pid == 0) {
        close(fd[1]); // Close write end of pipe
        dup2(fd[0], STDIN_FILENO);   // Redirect read end
            of pipe to process input
        close(fd[0]); // Close read end of pipe

        // Execute filter
        execl("./filter", "filter", (char *)NULL);
        perror("execl for filter failed\n");
        exit(EXIT_FAILURE);
}

// Close both ends of pipe
close(fd[1]);
close(fd[0]);

sleep(20);
```

```
            kill_process(pid);
            sleep(2);
            kill_process(procmon_pid);
            kill_process(filter_pid);
            waitpid(pid, NULL, 0);
            waitpid(procmon_pid, NULL, 0);
            waitpid(filter_pid, NULL, 0);
        }

        return 0;
    }
```

3.2 Signal Handling

We successfully sent signals to the `calcloop` process, observing the effects on its output. Screenshots of the commands and their effects are shown in Figure 4.

3.3 Java Threads Implementation

The modified Java code effectively utilized threads for rendering the Mandelbrot set. The updated display is shown in Figure 4.

3.4 Thread Pool Implementation

The implementation of a thread pool improved performance by managing thread resources more efficiently. The impact can be seen in Figure 5.

4 Discussion and Conclusion

This lab provided valuable insights into interprocess communication and threading in Java. Key learnings include:

- The effectiveness of using pipes for IPC in C programs.
- The importance of process management and signal handling in a Linux environment.
- Practical experience in Java threading and the advantages of using thread pools for resource management.

Challenges Encountered

We expected that sending the `SIGCONT` signal would resume the process, but looking at the output of `mon2` in `test.log`, it seems that it never continued.

Screenshots and Evidence

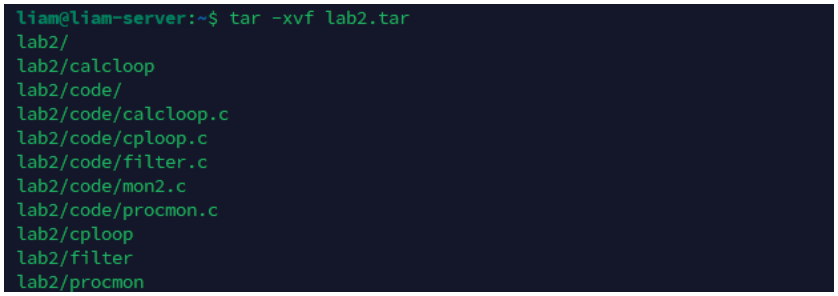

Figure 1: Extraction of `lab2.tar`.

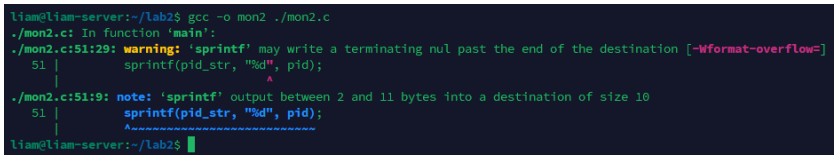

Figure 2: Compilation of `mon2.c`.

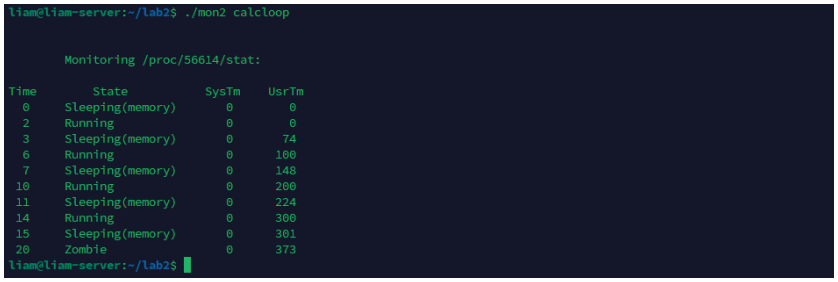

Figure 3: Execution of `mon2`.

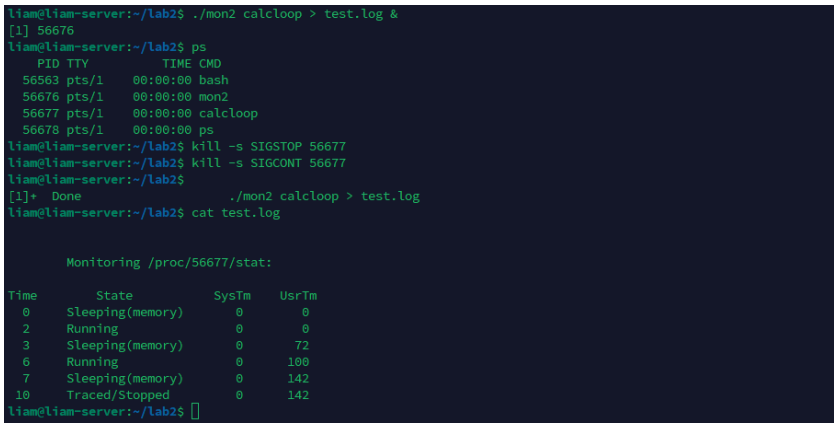

Figure 4: Sending signals to `calcloop`.

Figure 5: Compilation of Java files for Mandelbrot.

Figure 6: Execution of `MandelBrot` with parameters.