Отчет по лабораторной работе №7 на тему:

«Изучение процессов POSIX»

План работы

Изучить теорию из присоединенного файла и написать на языке C программу launch для Linux, которая принимает в первом параметре командной строки имя файла программы для запуска, а во втором параметре командной строки имя файла стандартного вывода для запущенной программы.

launch <prog> <file>

Программа launch должна запустить программу в дочернем процессе и перенаправить стандартный вывод запущенной программы в файл , дождаться завершения дочернего процесса и вывести на экран PID и код возврата завершенного процесса.

Ход работы

Для выполнения данной лабораторной работы используется виртуальная машина Oracle Virtual Box, Ubuntu и язык программирования C.

Для начала перейдем в папку os, в которой хранятся все лабораторные работы. Внутри нее при помощи nano создадим файл launch.c и запишем в нем скрипт (Рисунок 1), (Рисунок 2).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <prog> <file>\n", argv[0]);
        return 1;
    }

    char *prog = argv[1];
    char *file = argv[2];

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    }

    if (pid == 0) {
        int fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IWGPR | S_IROTH);
        if (fd < 0) {
            perror("Failed to open file");
            exit(1);
        }

        if(dup2(fd, STDOUT_FILENO) < 0) {
            perror("Failed to redirect stdout");
            close(fd);
            exit(1);
        }
        close(fd);
        execlp(prog, prog, NULL);
        perror("Failed to execute program");
        exit(1);
    } else {
        int status;
        pid_t child_pid = wait(&status);
```

Рисунок 1 – Первая часть кода в launch.c

```c
        if(child_pid < 0) {
            perror("wait failed")
            return 1;
        }
        printf("Child process PID: %d\n", child_pid);

        if(WIFEXITED(status)) {
            printf("Exit code: %d\n", WEXITSTATUS(status));
        } else {
            printf("Child process did not terminate normally\n");
        }
    }
    return 0;
}
```

Рисунок 2 – Вторая часть кода в launch.c

На (Рисунок 3) находятся команды:

1. nano – открытие и создание файла launch.c
2. gcc -o launch launch.c – компиляция launch.c
3. nano test.c – создание тестового скрипта, он просто выводит «Hello, World!»
4. компиляция тестовой программы
5. выполнение нашей программы на тестовой

В результате вы получаем PID и код возврата завершенного процесса. При помощи cat выводим содержимое текстового файла с выходными данными, полученными при помощи нашего скрипта.

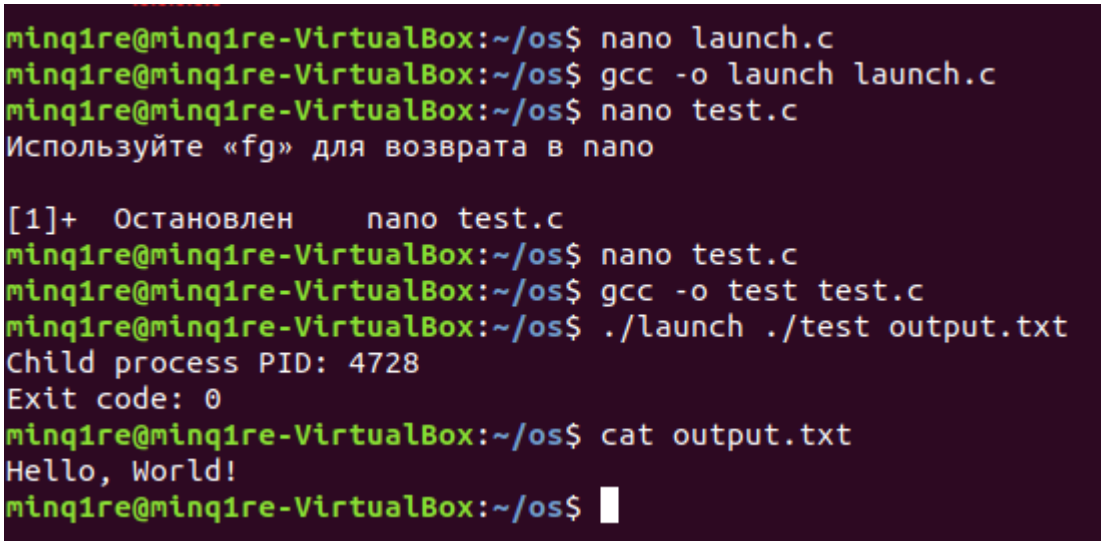

Рисунок 3 – Проверка скрипта

Вывод

Используя язык программирования C, я разработал программу launch, которая управляет процессами и потоками ввода-вывода в операционной системе Linux.

Программа launch получает в качестве аргументов командной строки два параметра:

- Имя исполняемого файла, который необходимо запустить
- Имя файла, в который будет перенаправлен стандартный вывод запущенного процесса

Приложение launch выполняет следующие действия:

- Создает дочерний процесс с помощью системного вызова fork().
- Перенаправляет стандартный вывод дочернего процесса в указанный файл с помощью системного вызова dup2().
- Ожидает завершения дочернего процесса с помощью системного вызова wait().
- Выводит на экран идентификатор процесса (PID) и код возврата завершившегося дочернего процесса.

Разработка этой программы позволила мне закрепить следующие навыки работы в среде Linux:

- Создание новых процессов с помощью fork()
- Управление файловыми дескрипторами с помощью dup2()
- Замена текущей программы другой программой с помощью execlp()

В целом, эта задача помогла мне улучшить понимание работы с процессами, файлами и параметрами командной строки в операционной системе Linux.

Отчет по лабораторной работе №8 на тему:

«Изучение процессов POSIX»

План работы

Изучить теорию из присоединенного файла и написать программу для Linux, которая выполняет следующие действия:

1. Создает дочерний процесс, который каждые 3 секунды посылает сигнал SIGUSR1 родительскому процессу
2. Создает в текущем каталоге именованный канал с именем reqests и открывает его для чтения
3. Создает сигнальные дескрипторы для обработки сигналов SIGINT и SIGUSR1
4. Используя poll ожидает в цикле на дескрипторах сигналов и канала
5. При получении данных из именованного канала выводит данные на экран как текст
6. При получении сигнала SIGUSR1 выводит имя полученного сигнала на экран
7. При получении сигнала SIGINT завершает цикл ожидания
8. После завершения цикла ожидания посылает сигнал SIGTERM дочернему процессу
9. Ждет завершение дочернего процесса и выводит на экран его код завершения
10. Удаляет ранее созданный именованный канал reqests и завершается

Ход работы

Для выполнения данной лабораторной работы используется виртуальная машина Oracle Virtual Box, Ubuntu и язык программирования C.

Для начала перейдем в папку os, в которой хранятся все лабораторные работы. Внутри нее при помощи nano создадим файл signal_pipe.c и запишем в нем скрипт (Рисунки 4-6).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <poll.h>
#include <errno.h>
#include <string.h>

#define FIFO_NAME "reqests"
#define BUFFER_SIZE 256

volatile sig_atomic_t keep_running = 1;

void handle_sigusr1(int signo) {
    printf("Received signal: SIGUSR1\n");
}

void handle_sigint(int signo) {
    printf("Received signal: SIGINT, exiting...\n");
    keep_running = 0;
}

int main() {
    if (access(FIFO_NAME, F_OK) == 0) {
        printf("Named pipe '%s' already exists. Deleting...\n", FIFO_NAME);
        unlink(FIFO_NAME);
    }

    if (mkfifo(FIFO_NAME, 0666) == -1) {
        perror("Failed to create named pipe");
        return 1;
    }

    printf("Named pipe '%s' created.\n", FIFO_NAME);

    pid_t child_pid = fork();

    if (child_pid < 0) {
        perror("Fork failed");
```

Рисунок 4 – Первая часть кода на C

```
    unlink(FIFO_NAME);
    return 1;
}

if (child_pid == 0) {
    while(1) {
        sleep(3);
        kill(getppid(), SIGUSR1);
    }
    exit(0);
} else {
    int fifo_fd = open(FIFO_NAME, O_RDONLY | O_NONBLOCK);
    if (fifo_fd == -1) {
        perror("Failed to open named pipe");
        kill(child_pid, SIGTERM);
        unlink(FIFO_NAME);
        return 1;
    }

    struct sigaction sa_usr1, sa_int;
    sa_usr1.sa_handler = handle_sigusr1;
    sa_usr1.sa_flags = 0;
    sigemptyset(&sa_usr1.sa_mask);
    sigaction(SIGUSR1, &sa_usr1, NULL);

    sa_int.sa_handler = handle_sigint;
    sa_int.sa_flags = 0;
    sigemptyset(&sa_int.sa_mask);
    sigaction(SIGINT, &sa_int, NULL);

    struct pollfd fds[1];
    fds[0].fd = fifo_fd;
    fds[0].events = POLLIN;

    printf("Waiting for events...\n");

    while (keep_running) {
        int ret = poll(fds, 1, -1);
        if (ret == -1) {
            if (errno == EINTR) {
                continue;
            }
```

Рисунок 5 – Вторая часть кода на C

```
            perror("Poll failed");
            break;
        }

        if (fds[0].revents & POLLIN) {
            char buffer[BUFFER_SIZE];
            ssize_t bytes_read = read(fds[0].fd, buffer, sizeof(buffer));
            if (bytes_read > 0) {
                buffer[bytes_read] = '\0';
                printf("Received from pipe: %s\n", buffer);
            }
        }
    }

    kill(child_pid, SIGTERM);
    int status;
    waitpid(child_pid, &status, 0);

    if (WIFEXITED(status)) {
        printf("Child process exited with code: %d\n", WEXITSTATUS(status));
    } else {
        printf("Child process terminated abnormally.\n");
    }

    close(fifo_fd);
    unlink(FIFO_NAME);
    printf("Named pipe '%s' deleted.\n", FIFO_NAME);

    return 0;
}
}
```

Рисунок 6 – Третья часть кода на C

После того, как сохранили изменения в файле signal_pipe.c компилируем его и выполняем (Рисунок 7).

```
minq1re@minq1re-VirtualBox:~/os$ nano signal_pipe.c
minq1re@minq1re-VirtualBox:~/os$
minq1re@minq1re-VirtualBox:~/os$ gcc -o signal_pipe signal_pipe.c
```

Рисунок 7 – Компиляция кода

Запишем данные в канал с помощью другого параллельно открытого терминала (Рисунок 8).

При запуске программы выводится:

```
Named pipe 'reqests' created.
Waiting for events...
```

Каждые 3 секунды будет выводиться:

```
Received signal: SIGUSR1
```

При записи данных в канал:

```
echo "Hello World" > reqests
```

Вывод в основном терминале:

```
Received from pipe: Hello World
```

При завершении программы с помощью Ctrl+C:

```
Received signal: SIGINT, exiting...
Child process exited with code: 0
Named pipe 'reqests' deleted.
```

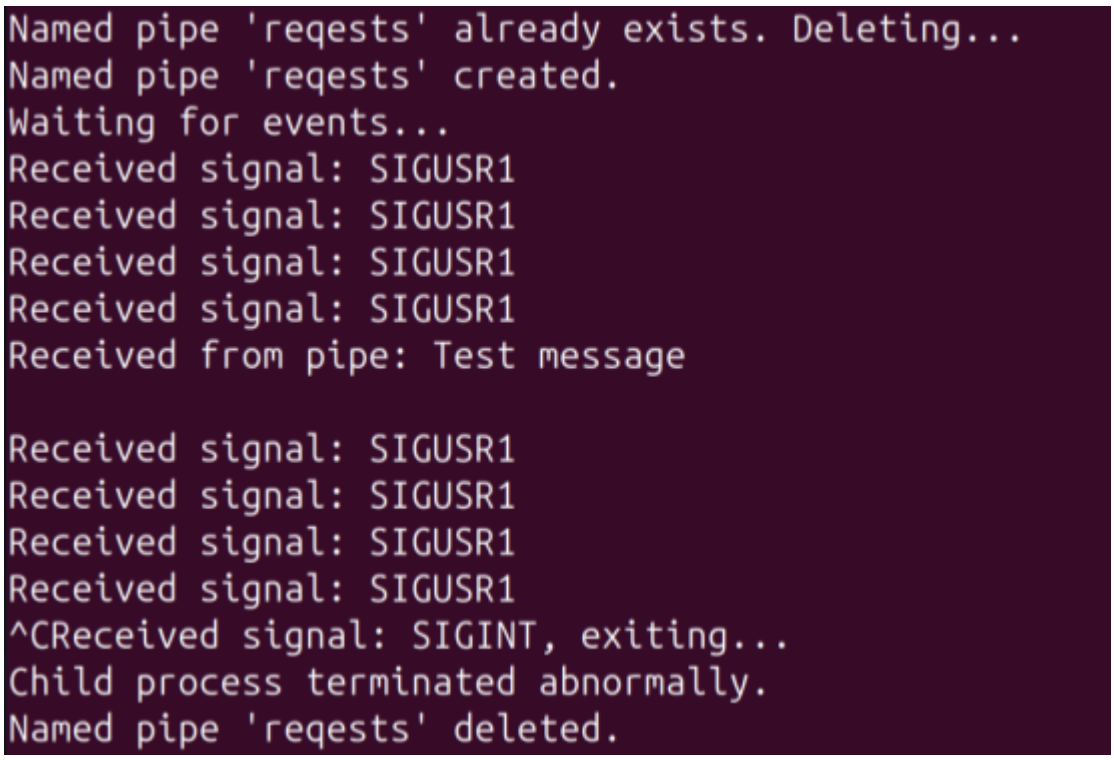

Рисунок 8 – Выполнение и тестирование скрипта

Вывод

Этот проект позволил мне освоить несколько важных аспектов программирования под Linux: управление процессами и сигналами, межпроцессное взаимодействие и обработку событий. В частности, я научился:

1. Создавать и управлять процессами: используя fork() для создания дочерних процессов и sigaction() для обработки сигналов, обеспечивая надёжное управление их жизненным циклом.

2. Реализовывать межпроцессное взаимодействие: с помощью именованных каналов для обмена данными между процессами.
3. Обрабатывать множественные события: эффективно используя poll() для одновременного мониторинга событий в именованном канале и реагирования на поступающие сигналы.
4. Обеспечивать надёжное завершение программы: грамотно завершая все дочерние процессы и удаляя созданные временные файлы, гарантируя чистоту системы после завершения работы программы.

Отчет по лабораторной работе №9 на тему:

«Изучение процессов POSIX»

План работы

Исправить (добавить семафор) в приведенную программу, чтобы исключить конфликт родительского и дочернего процессов при выводе текста на экран. Каждая строчка должна содержать или 10 символов 'P' или 10 символов 'C', символы не должны смешиваться в одной строке.

Ход работы

Для выполнения данной лабораторной работы используется виртуальная машина Oracle Virtual Box, Ubuntu и язык программирования C, а также программа, данная в задании.

Создадим файл process_sync.c при помощи nano, вставим в него код и добавим семафор. (Рисунки 9-10)

Семафоры:

- sem_parent – для сигнала от родительского процесса к дочернему о завершении работы
- sem_child – для сигнала от дочернего процесса к родительскому

Цикл ожидания:

- sem_wait – ожидание сигнала
- sem_post – отправка сигнала

Используется для исключения конфликта при выводе.

```c
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/wait.h>

int main() {
    // создаем два именованных семафора
    sem_t *sem_parent = sem_open("/sem_parent", O_CREAT, 0666, 1);
    sem_t *sem_child = sem_open("/sem_child", O_CREAT, 0666, 0);

    if (sem_parent == SEM_FAILED || sem_child == SEM_FAILED) {
        perror("sem_open");
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) { // дочерний процесс
        while(1) {
            sem_wait(sem_child);
            for (int i = 0; i < 10; i++) {
                putchar('C');
                fflush(stdout);
                usleep(10000); // 10 ms
            }
            putchar('\n');
            sem_post(sem_parent); // сигнал родителю
        }
    }

    // Родительский процесс
    while(1) {
        sem_wait(sem_parent); // ждем сигнала от дочернего процесса
        for (int i = 0; i < 10; i++) {
            putchar('P');
            fflush(stdout);
            usleep(10000); // 10 ms
        }
```

Рисунок 9 – Первая часть скрипта на C

```c
            putchar('\n');
            sem_post(sem_parent); // сигнал родителю
        }
    }

    // Родительский процесс
    while(1) {
        sem_wait(sem_parent); // ждем сигнала от дочернего процесса
        for (int i = 0; i < 10; i++) {
            putchar('P');
            fflush(stdout);
            usleep(10000); // 10 ms
        }
        putchar('\n');
        sem_post(sem_child); // сигал дочернему процессу
    }

    //  Код очистки никогда не выполняется, так как программа бесконечна
    //  Закрытие семафоров

    sem_close(sem_parent);
    sem_close(sem_child);

    // Удаление семафоров
    sem_unlink("/sem_parent");
    sem_unlink("/sem_child");

    return 0;
}
```

Рисунок 10 – Вторая часть скрипта на C

Скомпилируем программу и протестируем ее. Программа останавливается при использовании комбинации клавиш Ctrl+C (Рисунок 11).

Рисунок 11 - Компиляция программы и тестирование

Вывод

Этот проект позволил мне освоить использование POSIX-семафоров для синхронизации доступа к ресурсам между родительским и дочерним процессами. Благодаря применению семафоров, вывод программы стал корректным и последовательным, предотвращая наложение сообщений от родительского и дочернего процессов и гарантируя, что выводимые строки содержат символы только одного типа. Это решение исключило конфликты при одновременном доступе к стандартному выводу.