Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЕТ

к лабораторной работе №4

на тему

**УПРАВЛЕНИЕ ПРОЦЕССАМИ И ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ**

Студент И. В. Бобко

Преподаватель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146752068)

[2 Теоретические сведения 4](#_Toc146752069)

[3 Результат выполнения 6](#_Toc146752070)

[Заключение 6](#_Toc146752071)

[Список использованных источников 7](#_Toc146752072)

[Приложение А (обязательное) Листинг кода 8](#_Toc146752073)

1. **ЦЕЛЬ РАБОТЫ**

Целью данной работы является изучение основных особенностей подсистемы управления процессами и средств взаимодействия процессов в Unix. Практическое проектирование, реализация и отладка программных комплексов из нескольких взаимодействующих процессов.

Написать программу, реализующую самовосстанавливающийся процесс, который при получении сигнала, стандартно вызывающего завершение, создает свою копию, которая продолжает выполняться с прерванного места, и лишь после этого завершается, избегая таким образом безусловного «уничтожения» неперехватываемым сигналом. В качестве демонстрации «живости» процесса и его выполнения можно использовать произвольные действия, повторяющиеся периодически и дающие заметный результат. Для консольных приложений это может быть, например, счетчик, значение которого обновляется с заданной частотой и записывается в файл.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

В языке программирования C и операционной системе Linux для обмена информацией между процессами и обработки событий используются сигналы. Сигналы являются асинхронными уведомлениями, отправляемыми одним процессом другому процессу или самому себе [1].

В C сигналы реализуются с помощью функций из заголовочного файла signal.h. Основные функции, используемые для работы со сигналами, включают:

* signal. Эта функция позволяет установить обработчик сигнала для определенного типа сигнала. Прототип функции выглядит следующим образом: void (\*signal(int signum, void (\*handler)(int)))(int). Здесь signum - номер сигнала, а handler - указатель на функцию, которая будет вызываться при получении сигнала.
* kill. Эта функция используется для отправки сигнала другому процессу. Прототип функции выглядит следующим образом: int kill(pid\_t pid, int sig). Здесь pid - идентификатор процесса, которому отправляется сигнал, а sig - номер сигнала.
* raise. Эта функция отправляет сигнал текущему процессу. Прототип функции выглядит следующим образом: int raise(int sig). Здесь sig - номер сигнала.
* sigaction. Эта функция позволяет установить обработчик сигнала с более сложным поведением, чем функция signal(). Прототип функции выглядит следующим образом: int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact). Здесь signum - номер сигнала, act - указатель на структуру struct sigaction, содержащую информацию о поведении обработчика сигнала, и oldact - указатель на структуру, в которую будет сохранен предыдущий обработчик сигнала [2].

Linux также предоставляет ряд стандартных сигналов, таких как SIGINT (сигнал прерывания), который генерируется при нажатии комбинации клавиш Ctrl+C, и SIGTERM (сигнал завершения), который обычно используется для грациозного завершения процесса.

При работе со сигналами важно учитывать асинхронный характер их обработки. Рекомендуется обрабатывать сигналы как можно быстрее и выполнять в обработчиках только самые необходимые операции, так как во время обработки сигнала некоторые функции могут давать непредсказуемые результаты.

1. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

В результате лабораторной работы был написана программа, реализующая самовосстанавливающийся процесс, который при получении сигнала, стандартно вызывающего завершение, создает свою копию, которая продолжает выполняться с прерванного места, и лишь после этого завершается. В качестве демонстрации «живости» процесса и его выполнения использован счётчик (рисунок 1).

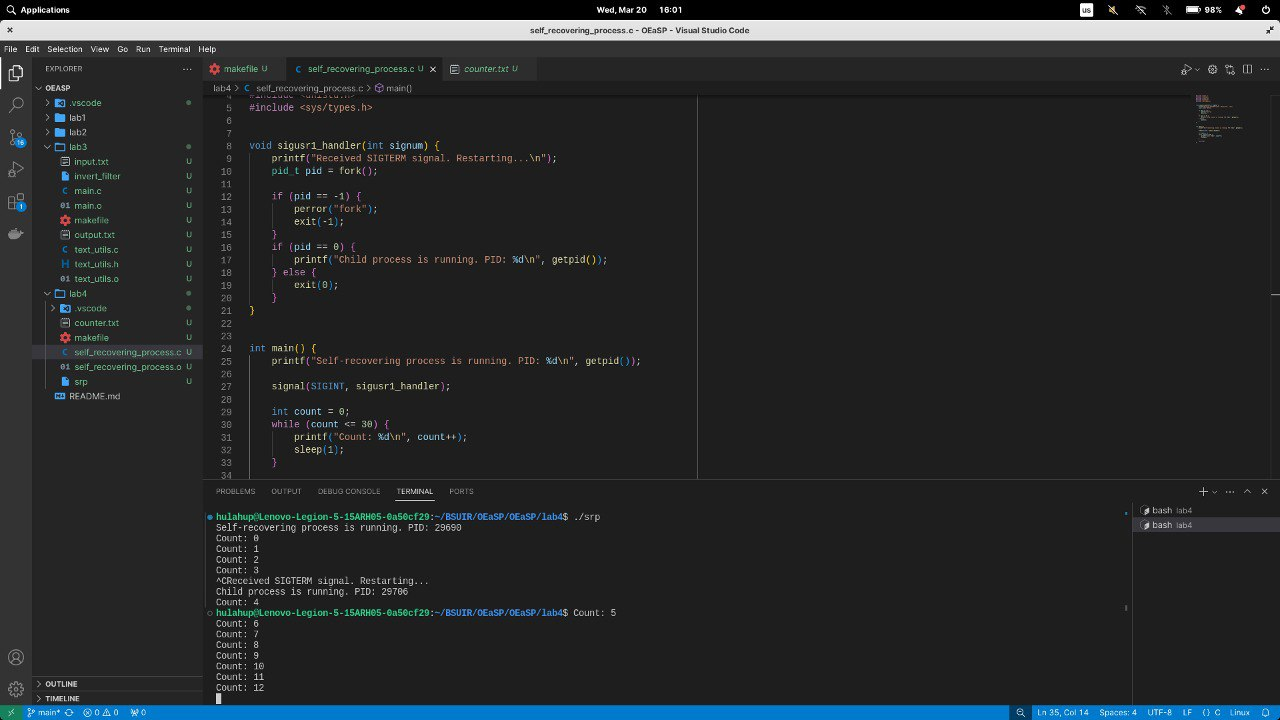


Рисунок 1 – Результаты выполнения программы

# ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы были изучены основные особенностей подсистемы управления процессами и средств взаимодействия процессов в Unix. Практическое проектирование, реализация и отладка программных комплексов из нескольких взаимодействующих процессов. Также была разработана программа, реализующая процесс, который при получении сигнала, стандартно вызывающего завершение, создает свою копию, которая продолжает выполняться с прерванного места, и лишь после этого завершается. избегая таким образом безусловного «уничтожения» неперехватываемым сигналом.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Понятие о сигналах [Электронный ресурс]. – Режим доступа: https://www.opennet.ru/docs/RUS/linux\_parallel/node10.html – Дата доступа: 10.03.2024.
2. Signals [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/cpp/c-runtime-library/reference/signal?view=msv – Дата доступа: 11.03.2024.

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – Файл self\_recovery.c

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

void sigusr1\_handler(int signum) {

printf("Received SIGUSR1 signal. Restarting...\n");

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(1);

} else if (pid == 0) {

sleep(5); // Пример работы

printf("Child process: Work complete.\n");

exit(0);

} else {

return;

}

}

int main() {

signal(SIGUSR1, sigusr1\_handler);

printf("Self-recovering process is running...\n");

int count = 0;

while (1) {

printf("Count: %d\n", count++);

sleep(1);

}

return 0;

}

Листинг 2 – Файл makefile

CC = gcc

CFLAGS = -Wall

all: self\_recovery

self\_recovery: self\_recovery.c

$(CC) $(CFLAGS) -o self\_recovery self\_recovery.c

clean:

rm -f self\_recovery