

Министерство образования Республики Беларусь

Учреждение образования  
«Белорусский государственный университет  
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей  
Кафедра ПОИТ

Дисциплина «Архитектура компьютерной техники и  
операционных систем»

ОТЧЁТ  
к лабораторной работе №8

ИСПОЛЬЗОВАНИЕ СИГНАЛОВ В ОС LINUX

Вариант 8

Студент группы №351001  
Ушаков А.Д.  
Преподаватель  
Леванцевич В.А.

Минск 2024

## СОДЕРЖАНИЕ

Введение .....	3
1 Задание к работе .....	5
1.1 Требования к выполнению .....	5
1.2 Индивидуальное задание .....	5
2 Реализация программы .....	6
2.1 Код программы .....	6
2.2 Результат работы программы .....	11
2.2.1 Создание процессов .....	11
2.2.2 Обмен сигналами .....	12
2.2.3 Завершение процессов .....	12
Заключение .....	13
Список использованных источников .....	15

## ВВЕДЕНИЕ

Современные операционные системы предоставляют множество механизмов взаимодействия между процессами, одним из которых является использование сигналов. Сигналы играют ключевую роль в организации межпроцессного взаимодействия и управления процессами в системах семейства UNIX, включая Linux. Они представляют собой асинхронные уведомления, которые используются для передачи управляющих команд от одного процесса к другому или от ядра операционной системы к процессу. Это позволяет процессам реагировать на определённые события или изменения состояния системы, обеспечивая гибкость и динамичность управления.

Основная особенность сигналов заключается в том, что они не переносят данные напрямую, ограничивая их применение как средства общего обмена информацией. Вместе с тем, их использование чрезвычайно эффективно для оповещения процессов о произошедших событиях. Каждый сигнал имеет уникальное числовое значение и мнемоническое имя, которые стандартизированы и задаются в заголовочном файле `<signal.h>`. Это обеспечивает универсальность и удобство использования сигналов в программировании. Например, сигналы `SIGINT`, `SIGTERM` и `SIGKILL` применяются для управления процессами, а сигналы `SIGUSR1` и `SIGUSR2` предоставляют возможность разработчикам задавать собственные сценарии обработки.

Сигналы могут быть обработаны процессами по-разному: игнорированы, обработаны по умолчанию или перехвачены с помощью пользовательских обработчиков. Последний подход является наиболее гибким и позволяет программисту задавать конкретные действия при получении определённого сигнала. Для управления обработкой сигналов используются системные вызовы `signal()` и `sigaction()`, предоставляющие мощный инструмент для настройки поведения процессов. При помощи этих вызовов можно настроить процесс на выполнение специальных функций при поступлении сигналов, что позволяет реализовать сложные сценарии взаимодействия.

Процессы в Linux также могут генерировать сигналы для других процессов с помощью системного вызова `kill()`. Этот вызов позволяет посылать сигналы как отдельным процессам, так и группам процессов, предоставляя разработчику возможность эффективного управления как на уровне отдельных программ, так и на уровне целых систем. Дополнительно, разработчики могут использовать наборы сигналов, определяемые с помощью функций `sigemptyset()` и `sigfillset()`, для гибкого управления группами сигналов и их обработкой.

Современные системы предоставляют не только механизмы отправки и обработки сигналов, но и дополнительные возможности для их управления. Например, использование масок сигналов позволяет временно блокировать обработку определённых сигналов, что может быть полезно для предотвращения нежелательных прерываний в критических участках кода. Маски сигналов задаются при помощи функций, таких как `sigprocmask()` и `sigsetjmp()`, обеспечивая программисту полный контроль над тем, как и когда обрабатываются сигналы. Это особенно важно при разработке многопоточных приложений, где требуется учитывать взаимодействие между потоками и возможность непредсказуемого поступления сигналов.

Значимость изучения механизма сигналов обусловлена их универсальностью и важной ролью в управлении процессами. Отладка, мониторинг, синхронизация и завершение процессов – это лишь некоторые задачи, которые эффективно решаются с использованием сигналов. Данная работа направлена на изучение сигналов как одного из основных инструментов межпроцессного взаимодействия в Linux. Это включает изучение способов их генерации, обработки и применения в различных сценариях, что является важным этапом в освоении программирования для UNIX-подобных систем.

# 1 ЗАДАНИЕ К РАБОТЕ

## 1.1 Требования к выполнению

Создать дерево процессов согласно варианту индивидуального задания.

Процессы непрерывно обмениваются сигналами. Запись вида 1 -> (2,3,4,5) означает, что исходный процесс 0 создаёт дочерний процесс 1, который, в свою очередь, создаёт дочерние процессы 2,3,4,5. Запись вида: 1 -> (2,3,4) SIGUSR1 означает, что процесс 1 посылает дочерним процессам 2,3,4 одновременно (т.е. за один вызов kill ()) сигнал SIGUSR1. После передачи 101-го по счету сигнала SIGUSR родительский процесс посылает сыновьям сигнал и ожидает завершения всех сыновей, после чего завершается. Сыновья, получив сигнал SIGTERM завершают работу с выводом на консоль сообщения вида: «Pid ppid завершил работу после X-го сигнала SIGUSR1 и Y-го сигнала SIGUSR2», где X, Y – количество посланных за все время работы данным сыном сигналов SIGUSR1 и SIGUSR2. Каждый процесс в процессе работы выводит на консоль информацию в следующем виде: N pid ppid послал/получил USR1/USR2 текущее время (мксек), где N-номер сына.

## 1.2 Индивидуальное задание

В варианте 8 необходимо реализовать следующее дерево:

- 1->2, SIGUSR2;
- 2->(3,4,5), SIGUSR1;
- 4->6, SIGUSR1;
- 3->7, SIGUSR1;
- 5->8, SIGUSR1;
- 8->1, SIGUSR2.

## 2 РЕАЛИЗАЦИЯ ПРОГРАММЫ

### 2.1 Код программы

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <memory.h>
#include <stdbool.h>
#include <sys/time.h>
#include <wait.h>

#define PROCESSES_COUNT 9 /* Количество процессов, включая
родительский */
#define SIGN_ITERATION_COUNT 101 /* Количество итераций отправки
сигналов */

/* Структура для хранения данных в разделяемой памяти */
typedef struct {
    /* Массив идентификаторов процессов */
    pid_t pids[PROCESSES_COUNT];
    // Счетчик общих сигналов
    int totalCount;
} SharedMemory;

// Указатель на разделяемую память
SharedMemory *shared_memory;
// Счетчик сигналов SIGUSR1
int sigUser1Count;
// Счетчик сигналов SIGUSR2
int sigUser2Count;
// Функция для получения текущего времени в микросекундах
long long getTimeInMicroseconds() {
    // Структура для хранения времени
    struct timeval tv;

    // Получение текущего времени
    if (gettimeofday(&tv, NULL) == 0) {
        // Преобразование в микросекунды
        return (long long) tv.tv_sec * 1000000 + tv.tv_usec;
    }
}

// Функция обновления состояния при получении сигнала
void updateStates(int sig) {
    // Если сигнал SIGUSR1
    if (sig == SIGUSR1) {
```

```

        // Увеличиваем счетчик SIGUSR1
        sigUser1Count++;
    }
    else { // Если сигнал SIGUSR2
        // Увеличиваем счетчик SIGUSR2
        sigUser2Count++;
    }
    // Увеличиваем общий счетчик сигналов
    shared_memory->totalCount++;
}

// Функция для отправки сигнала от одного процесса к другому
void send_sig(const int sender_index, const int receiver_index, const
int sig) {
    // Проверка количества сигналов
    if (shared_memory->totalCount <= SIGN_ITERATION_COUNT
        // Проверка, что текущий процесс – отправитель
        && getpid() == shared_memory->pids[sender_index]) {

        // Вывод информации о сигнале
        printf("Process %d sends signal %s to process %d\n",
            shared_memory->pids[sender_index],
            (sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2"),
            shared_memory->pids[receiver_index]
        );
        // Отправка сигнала
        kill(shared_memory->pids[receiver_index], sig);
    }
}

// Основная функция работы процессов
void work() {
    // Пока количество сигналов не превышает лимит
    while (shared_memory->totalCount <= SIGN_ITERATION_COUNT) {
        // Отправка сигнала SIGUSR2 от процесса 1 к процессу 2
        send_sig(1, 2, SIGUSR2);
        send_sig(2, 3, SIGUSR1);
        send_sig(2, 4, SIGUSR1);
        send_sig(2, 5, SIGUSR1);
        send_sig(3, 7, SIGUSR1);
        send_sig(4, 6, SIGUSR1);
        send_sig(5, 8, SIGUSR1);
        send_sig(8, 1, SIGUSR2);
    }
}

// Функция создания нового процесса
void createProcess(int process_index) {
    // Создание нового процесса
    pid_t newPid = fork();
    // Дочерний процесс

```

```

    if (newPid == 0) {
        // Сохранение идентификатора процесса
        shared_memory->pids[process_index] = getpid();
        // Вывод информации о создании
        printf("Process %d created\n", getpid());
    }
    // Ошибка при создании процесса
    else if (newPid < 0) {
        // Завершение программы с ошибкой
        exit(1);
    }
}

// Функция создания всех процессов
void createAllProcess() {
    createProcess(1); // Создание процесса 1
    // Создание процесса 2 и т.д.
    if (getpid() == shared_memory->pids[1])
    { createProcess(2); }
    if (getpid() == shared_memory->pids[2])
    { createProcess(3); }
    if (getpid() == shared_memory->pids[2])
    { createProcess(4); }
    if (getpid() == shared_memory->pids[2])
    { createProcess(5); }
    if (getpid() == shared_memory->pids[4])
    { createProcess(6); }
    if (getpid() == shared_memory->pids[3])
    { createProcess(7); }
    if (getpid() == shared_memory->pids[5])
    { createProcess(8); }
}

// Обработчик сигналов SIGUSR1 и SIGUSR2
void handleUserSignals(int signum, siginfo_t* info, void *ctx) {
    printf(
        "Process %d receive signal %s from process %d\nCurrent time:
        %lld\n",
        // Текущий процесс, принимающий сигнал
        getpid(),
        // Тип принятого сигнала
        (signum == SIGUSR1 ? "SIGUSR1" : "SIGUSR2"),
        // PID процесса, отправившего сигнал
        info->si_pid,
        // Текущее время в микросекундах
        getTimeInMicroseconds()
    );
    /* Обновляем состояние программы: увеличиваем счетчики сигналов */
    updateStates(signum);
}

```



```

// Обработчик сигнала завершения процесса (SIGTERM)
void handleTerminateSignal(int signum, siginfo_t* info, void *ctx) {
    printf(
        "Pid %d terminate:\ncalls of SIGUSR1: %d\ncalls of SIGUSR2:
%d.\n",
        // PID процесса, который завершается
        info->si_pid,
        // Количество принятых сигналов SIGUSR1
        sigUser1Count,
        // Количество принятых сигналов SIGUSR2
        sigUser2Count
    );
}

// Установка обработчиков сигналов
void setSignals() {
    // Обработчик для сигналов SIGUSR1 и SIGUSR2
    struct sigaction user_action;
    // Указываем функцию-обработчик
    user_action.sa_sigaction = handleUserSignals;
    // Очищаем набор блокируемых сигналов
    sigemptyset(&user_action.sa_mask);
    // Флаг для получения дополнительной информации о сигнале
    user_action.sa_flags = SA_SIGINFO;
    // Обработчик для сигнала SIGTERM
    struct sigaction term_action;
    // Указываем функцию-обработчик
    term_action.sa_sigaction = handleTerminateSignal;
    // Очищаем набор блокируемых сигналов
    sigemptyset(&term_action.sa_mask);
    // Флаг для получения дополнительной информации о сигнале
    term_action.sa_flags = SA_SIGINFO;

    // Назначаем обработчик для SIGUSR1
    sigaction(SIGUSR1, &user_action, NULL);
    // Назначаем обработчик для SIGUSR2
    sigaction(SIGUSR2, &user_action, NULL);
    // Назначаем обработчик для SIGTERM
    sigaction(SIGTERM, &term_action, NULL);
}

// Инициализация разделяемой памяти
void initCommonMemo() {
    int shm_id = shmget(IPC_PRIVATE, sizeof(SharedMemory), IPC_CREAT |
0666);
    /* Создаем сегмент разделяемой памяти. Размер равен размеру
структуры SharedMemory
    IPC_PRIVATE – создаем новую память, доступную только текущим
процессам
    Права доступа: чтение и запись (0666) */

```

```

// Если создание памяти завершилось с ошибкой
if (shm_id < 0) {
    // Выводим сообщение об ошибке
    printf("shmget error");
    // Завершаем выполнение программы
    exit(1);
}

shared_memory = (SharedMemory *)shmat(shm_id, NULL, 0);
/* Присоединяем сегмент памяти к адресному пространству текущего
процесса
    NULL позволяет системе выбрать подходящий адрес для привязки */

// Если привязка завершилась с ошибкой
if (shared_memory == (SharedMemory *)-1) {
    // Выводим сообщение об ошибке
    printf("shmat error");
    // Завершаем выполнение программы
    exit(1);
}

memset(shared_memory, 0, sizeof(SharedMemory));
// Очищаем область памяти, заполняя ее нулями

for (int i = 0; i < PROCESSES_COUNT; ++i) {
    /* Инициализируем массив PID, устанавливая все значения в 0 */
    shared_memory->pids[i] = 0;
}
}

// Главная функция программы
int main(void) {
    /* Устанавливаем обработчики сигналов для SIGUSR1, SIGUSR2 и
SIGTERM */
    setSignals();
    /* Инициализируем разделяемую память для хранения данных о
процессах */
    initCommonMemo();
    /* Выводим идентификатор главного (родительского) процесса */
    printf("Main process %d\n", getpid());
    /* Сохраняем PID главного процесса в разделяемой памяти */
    shared_memory->pids[0] = getpid();
    /* Создаем дерево процессов, определенное заданием */
    createAllProcess();
    /* Даем время на создание всех процессов, чтобы они были готовы к
работе */
    usleep(5000);
}

```

```

    /* Запускаем основной блок работы процессов, который включает
    передачу сигналов */
    work();

    // Проверяем, не является ли текущий процесс главным
    if (getpid() != shared_memory->pids[0])
        /* Завершаем текущий процесс, отправляя ему сигнал SIGTERM */
        kill(getpid(), SIGTERM);

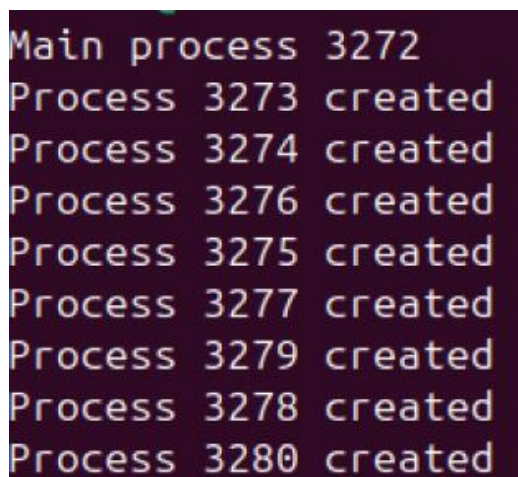
    /* Переменная для хранения статуса завершения процессов */
    int status = 0;
    // Цикл ожидания завершения всех дочерних процессов
    for (int i = 1; i < PROCESSES_COUNT; ++i) {
        // Ожидаем завершения очередного дочернего процесса
        pid_t exitedPid = wait(&status);
        // Проверяем, что процесс завершился корректно
        if (exitedPid > 0 && WIFEXITED(status)) {
            // Выводим сообщение о завершении процесса
            printf("Process %d has correctly killed\n", exitedPid);
        }
    }
    return 0;
}

```

## 2.2 Результат работы программы

### 2.2.1 Создание процессов

На рисунке 1 представлен вывод в консоль при создании сигналов.



```

Main process 3272
Process 3273 created
Process 3274 created
Process 3276 created
Process 3275 created
Process 3277 created
Process 3279 created
Process 3278 created
Process 3280 created

```

Рисунок 1 – Создание сигналов

### 2.2.2 Обмен сигналами

На рисунке 2 показан вывод в консоль, где можно увидеть, как процессы обмениваются сигналами.

```
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3274 receive signal SIGUSR2 from process 3273
Current time: 1732387420072849
Process 3274 receive signal SIGUSR2 from process 3273
Current time: 1732387420073774
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
Process 3273 sends signal SIGUSR2 to process 3274
```

Рисунок 2 – Обмен сигналами

### 2.2.3 Завершение процессов

На рисунке 3 представлен вывод в консоль при успешном завершении каждого процесса.

```
Pid 3273 terminate:
calls of SIGUSR1: 0
calls of SIGUSR2: 10.
Process 3274 receive signal SIGUSR2 from process 3273
Current time: 1732387420143171
Pid 3274 terminate:
calls of SIGUSR1: 0
calls of SIGUSR2: 13.
Process 3276 has correctly killed
Process 3277 has correctly killed
Process 3278 has correctly killed
Process 3275 receive signal SIGUSR1 from process 3274
Current time: 1732387420143331
Process 3275 has correctly killed
Process 3274 has correctly killed
Process 3273 has correctly killed
```

Рисунок 3 – Завершение процессов

## ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были изучены и применены на практике механизмы межпроцессного взаимодействия в операционной системе Linux с использованием сигналов. Работа позволила подробно исследовать концепцию сигналов, их роль в организации взаимодействия между процессами, а также методы обработки и управления ими. Реализация дерева процессов, заданного в индивидуальном варианте, дала возможность продемонстрировать различные подходы к созданию процессов и управлению их жизненным циклом, включая передачу сигналов и их обработку.

Создание дерева процессов с заданной структурой потребовало последовательного применения системных вызовов `fork()` для порождения дочерних процессов, а также использования сигналов `SIGUSR1` и `SIGUSR2` для организации взаимодействия между ними. Каждому процессу была назначена специфическая роль в соответствии с заданием, включая передачу сигналов другим процессам, их обработку и логирование событий. Это обеспечило наглядную демонстрацию асинхронного обмена информацией между процессами. Введение механизма подсчёта отправленных и полученных сигналов позволило организовать вывод итоговой статистики работы каждого процесса, что подчеркнуло значимость точного контроля над передачей сигналов.

Особое внимание было уделено обработке сигналов. Для этого использовались системные вызовы `signal()` и `sigaction()`, которые позволили установить пользовательские обработчики сигналов. Это обеспечило корректное завершение работы процессов при получении сигнала `SIGTERM`, а также организовало вывод сообщений о завершении работы каждого процесса с указанием количества отправленных и полученных сигналов. Такой подход позволил достичь устойчивости программы и её предсказуемого поведения даже в условиях интенсивного обмена сигналами.

Дополнительно было реализовано логирование действий каждого процесса в реальном времени. Это включало информацию о номере процесса, идентификаторах родительского и текущего процессов, типах отправленных или полученных сигналов и времени выполнения операции с микросекундной точностью. Реализация такого вывода продемонстрировала возможность создания высокодетализированных отчётов о взаимодействии процессов, что является важным аспектом при разработке и отладке сложных систем.

Построение дерева процессов и выполнение цикла передачи сигналов выявило ключевые аспекты управления процессами, такие как организация синхронизации, обработка ошибок и контроль завершения процессов. Использование системного вызова `kill()` продемонстрировало возможность адресной передачи сигналов как к конкретным процессам, так и к группам

процессов, что является основой для организации эффективного взаимодействия в многозадачных системах.

Результаты работы подтверждают эффективность использования сигналов как одного из базовых механизмов межпроцессного взаимодействия. Они обеспечивают высокий уровень контроля и гибкости в управлении процессами, что особенно важно при разработке сложных программных систем. Программная реализация продемонстрировала, что с помощью сигналов можно успешно организовать взаимодействие процессов в иерархических структурах с соблюдением заданных условий и последовательности действий.

В заключение можно отметить, что выполнение лабораторной работы позволило не только изучить теоретические аспекты сигналов в операционной системе Linux (см. рисунок 4), но и закрепить их посредством практической реализации. Полученные знания и навыки могут быть полезны в дальнейшем для создания более сложных систем, требующих взаимодействия и синхронизации большого количества процессов.

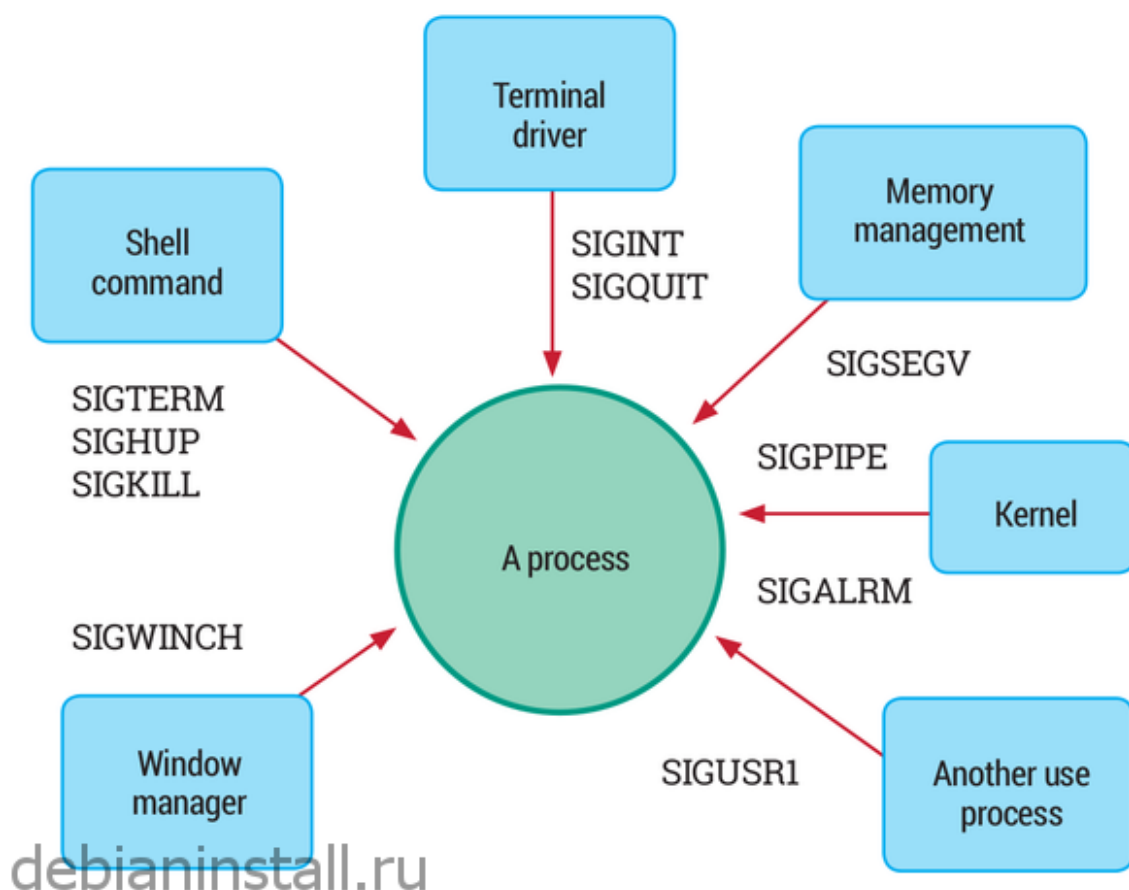


Рисунок 4 – Сигналы в ОС Linux

## **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Леванцевич, В. А. Использование сигналов в ОС Linux / В. А. Леванцевич // Лабораторные работы. – 2024. – №8.