МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

# МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ РАБОТАМ

## «Разработка Linux-приложений»

Направление подготовки 09.03.04 «Программная инженерия»
Направленность (профиль) «Разработка и сопровождение программного обеспечения»
Квалификация выпускника: бакалавр

Ставрополь
2026

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Настоящие методические рекомендации предназначены для выполнения лабораторных работ по дисциплине «Разработка Linux-приложений».

Дисциплина направлена на формирование у студентов системных теоретических знаний и углублённых практических навыков в области создания программного обеспечения для операционной системы Linux.

**Цель освоения дисциплины**

Формирование у студентов системных теоретических знаний и углубленных практических навыков в области разработки приложений для операционной системы Linux, включая использование системных вызовов, управление процессами и потоками, межпроцессное взаимодействие, сетевое программирование, а также освоение инструментов отладки, профилирования и оптимизации системного программного обеспечения.

**Задачи дисциплины**

1. Изучение архитектуры операционной системы Linux и механизмов взаимодействия пользовательских приложений с ядром через системные вызовы.

2. Освоение низкоуровневого файлового ввода-вывода и работы с файловыми дескрипторами.

3. Формирование навыков управления процессами (создание, завершение, синхронизация) и обработки сигналов.

4. Изучение многопоточного программирования с использованием библиотеки pthreads и средств синхронизации (мьютексы, семафоры, условные переменные).

5. Освоение методов межпроцессного взаимодействия (IPC): каналы, очереди сообщений, разделяемая память, сокеты.

6. Изучение основ сетевого программирования: создание клиент-серверных приложений с использованием сокетов TCP/UDP, мультиплексирование ввода-вывода (select/poll/epoll).

7. Освоение инструментов отладки и профилирования: gdb, strace, ltrace, valgrind, perf.

8. Формирование навыков создания статических и динамических библиотек, работы с системой сборки Make.

9. Изучение современных практик контейнеризации Linux-приложений (Docker).

Методические рекомендации включают 23 лабораторные работы, каждая из которых посвящена определённому аспекту разработки Linux-приложений. Лабораторные работы построены по единой структуре и содержат следующие разделы:

– цель работы – формулировка ожидаемых результатов освоения темы;

– задачи работы – перечень конкретных действий, которые необходимо выполнить;

– теоретическая часть – изложение основных понятий, системных вызовов и принципов работы;

– методика выполнения работы – пошаговые инструкции с примерами команд и кода;

– пример выполнения задания – демонстрация решения типовой задачи, не входящей в индивидуальные задания;

– индивидуальные задания – варианты заданий для самостоятельного выполнения;

– требования к отчёту – перечень материалов, которые должны быть представлены по итогам работы;

– контрольные вопросы – вопросы для проверки понимания изученного материала.

Лабораторные работы охватывают широкий спектр тем: от установки среды разработки и основ компиляции до сетевого программирования, профилирования и создания динамических библиотек. Особое внимание уделяется практическому применению системных вызовов, синхронизации потоков, межпроцессному взаимодействию и отладке.

# Лабораторная работа №1
# Установка и настройка среды разработки Linux-приложений

**Цель работы:** сформировать практические навыки по развёртыванию рабочей среды для разработки системных приложений в ОС Linux.

**Задачи:**

– Установить гипервизор VirtualBox (или VMware) на основную ОС.

– Создать виртуальную машину и установить Ubuntu (LTS-версию).

– Настроить сетевой доступ (NAT, при необходимости – мостовой адаптер).

– Настроить общую папку для обмена файлами между хостом и гостевой ОС.

– Установить пакеты build-essential, gcc, g++, gdb, make, git, а также дополнительные утилиты (cmake, valgrind, strace).

– Освоить базовые команды командной строки Linux.

– Проверить работоспособность инструментов на простейшей программе, освоить компиляцию, запуск и базовую отладку в GDB, а также сборку через Make.

## Теоретические сведения

### 1. Архитектура Linux и место прикладного разработчика

Linux – многопользовательская, многозадачная ОС с монолитным ядром. Взаимодействие прикладных программ с ядром происходит через системные вызовы (system calls). Стандартная библиотека C (glibc) предоставляет удобные обёртки для этих вызовов. Разработка системных приложений требует понимания файловых дескрипторов, процессов, сигналов, потоков и межпроцессного взаимодействия.

## 2. Ключевые инструменты разработчика в Linux

| Инструмент | Назначение |
|---|---|
| GCC / G++ | Компиляторы C и C++ |
| GDB | Отладчик (точки останова, просмотр переменных, трассировка) |
| Make | Автоматизация сборки на основе Makefile |
| Git | Система контроля версий |
| Valgrind | Обнаружение утечек памяти и ошибок работы с памятью |
| Strace / Ltrace | Трассировка системных вызовов и вызовов библиотечных функций |
| Perf | Профилирование производительности |

## 3. Виртуализация для разработки

Использование виртуальных машин (VirtualBox, VMware) даёт:

– изоляцию среды разработки от основной ОС;

– возможность экспериментировать без риска;

– снимки состояния (snapshots) для отката.

## 4. Основные команды командной строки Linux

| Команда | Действие |
|---|---|
| ls, ls -la | просмотр содержимого каталога |
| cd, pwd | навигация, текущий каталог |
| mkdir, rmdir | создание/удаление каталогов |
| cp, mv, rm | копирование, перемещение, удаление |
| cat, less, head, tail | просмотр файлов |
| chmod, chown | права доступа и владельцы |
| ps, top, htop | управление процессами |
| apt update, apt install | работа с пакетами |
| man команда | справочная система |

## 5. Управление пакетами в Ubuntu

Менеджер apt (Advanced Package Tool):

**bash**

```
sudo apt update          # обновление списка пакетов
sudo apt upgrade         # обновление установленных пакетов
sudo apt install <пакет> # установка
```

Пакет **build-essential** содержит **gcc**, **g++**, **make** и заголовочные файлы, необходимые для компиляции.

## Методические указания к выполнению работы

### 1. Установка VirtualBox и создание виртуальной машины

1.1. Скачать VirtualBox с официального сайта, установить.

1.2. Скачать ISO-образ Ubuntu (22.04 LTS или новее) с ubuntu.com.

1.3. В VirtualBox нажать «Создать»:

– **Имя:** LinuxDev

– **Тип:** Linux, версия: Ubuntu (64-bit)

– **Память:** 4096 МБ

– **Диск:** VDI, динамический, 30 ГБ

1.4. В настройках ВМ:

– **Носители –** загрузить ISO-образ.

– **Сеть** – Адаптер 1: NAT (интернет), Адаптер 2: Сетевой мост (по желанию).

– **Общие папки** – добавить папку хоста (например, D:\LinuxShared), указать имя **shared**, включить «Автомонтирование» и «Постоянная».

### 2. Установка Ubuntu

2.1. Запустить ВМ, следовать инструкциям установщика:

– Язык – русский/английский.

– Тип установки – «Стереть диск и установить Ubuntu».

– Имя пользователя, пароль, имя компьютера (например, developer).

2.2. После установки перезагрузить, войти в систему.

### 3. Установка гостевых дополнений VirtualBox (Guest Additions)

**bash**

sudo apt update

sudo apt install -y build-essential linux-headers-$(uname -r)

*# В меню VirtualBox: Устройства → Подключить образ диска с дополнениями гостевой ОС*

mount /dev/cdrom /mnt

cd /mnt

sudo ./VBoxLinuxAdditions.run

reboot

**4. Настройка общей папки**

4.1. После перезагрузки общая папка доступна в **/media/sf_shared**. Добавить пользователя в группу **vboxsf**:

**bash**

sudo usermod -aG vboxsf $USER

4.2. Выйти из системы и зайти снова. Проверить: **ls /media/sf_shared**.

**5. Установка инструментов разработчика**

5.1. Открыть терминал (Ctrl+Alt+T) и выполнить:

**bash**

sudo apt update

sudo apt upgrade -y

sudo apt install -y build-essential gdb git vim nano cmake valgrind strace ltrace

5.2. Проверить установку:

**bash**

gcc --version

g++ --version

gdb --version

make --version

git –version

**6. Первая программа: компиляция, запуск, отладка**

6.1. Создать рабочий каталог:

**bash**

```
mkdir ~/linux_labs
cd ~/linux_labs
```

6.2. Файл hello.c:

```
#include <stdio.h>
int main() {
    int a = 10, b = 20;
    int sum = a + b;
    printf("Hello, Linux! Sum = %d\n", sum);
    return 0;
}
```

6.3. Компиляция с отладочной информацией (-g):

**bash**

```
gcc -Wall -g hello.c -o hello
./hello
```

6.4. Отладка в GDB:

**bash**

```
gdb ./hello
(gdb) break main
(gdb) run
(gdb) next 3          # выполнить 3 строки
(gdb) print sum       # посмотреть значение переменной
```

(gdb) continue

(gdb) quit

**7. Сборка через Make**

7.1. Создать Makefile:

**makefile**

```
CC = gcc
CFLAGS = -Wall -g
TARGET = hello
SRCS = hello.c

all: $(TARGET)

$(TARGET): $(SRCS)
	$(CC) $(CFLAGS) -o $@ $^

clean:
	rm -f $(TARGET)
```

7.2. Запуск:

**bash**

```
make
make clean
```

## Пример выполнения задания

**Задача примера:** Установить Ubuntu 22.04, дополнительно установить clang и скомпилировать программу с его помощью.

### Шаг 1. Установка Ubuntu 22.04 LTS в VirtualBox

(Скриншот окна VirtualBox с запущенной ВМ – приведён в отчёте.)

### Шаг 2. Установка гостевых дополнений и общей папки

Общая папка shared на хосте C:\LinuxShared смонтирована в /media/sf_shared.

Выполнена команда:

bash

sudo usermod -aG vboxsf petrov

После перезахода ls /media/sf_shared показывает файлы хоста.

### Шаг 3. Установка инструментов

bash

sudo apt install -y build-essential gdb git clang valgrind

Проверка clang --version → clang version 14.0.0.

### Шаг 4. Написание программы

Файл calc.c:

```
#include <stdio.h>
int main() {
   int x = 5, y = 7;
   printf("Product: %d\n", x * y);
   return 0;
}
```

### Шаг 5. Компиляция через clang

bash

clang -Wall -g calc.c -o calc

./calc

Вывод: Product: 35

**Шаг 6. Отладка через GDB (скомпилированного clang)**

bash

gdb ./calc

(gdb) break main

(gdb) run

(gdb) print x

$1 = 5

(gdb) print y

$2 = 7

(gdb) continue

**Шаг 7. Makefile для компиляции clang**

makefile

```
CC = clang
CFLAGS = -Wall -g
calc: calc.c
	$(CC) $(CFLAGS) -o calc calc.c
clean:
	rm -f calc
```

Запуск make → сборка прошла успешно.

**Вывод:** среда разработки полностью настроена. Освоены базовые команды Linux, компиляция через GCC и clang, отладка в GDB, автоматизация сборки через Make. Общая папка работает корректно. Трудностей не возникло.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >10 – цикл повторяется: вариант 11 → вариант 1, вариант 12 → вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Установить Ubuntu 20.04 LTS в VirtualBox. Настроить общую папку. Написать программу на C, которая запрашивает имя пользователя и выводит приветствие «Hello, <имя>!». Скомпилировать с помощью GCC (ключ -Wall -g), отладить в GDB (установить точку останова на строке с printf, вывести значение переменной). |
| 2 | Установить Ubuntu 22.04 LTS в VirtualBox. Дополнительно установить компилятор clang (sudo apt install clang). Написать программу, вычисляющую факториал числа (число вводится с клавиатуры, использовать scanf). Скомпилировать программу clang (с ключами -Wall -g), проверить её работу в GDB. |
| 3 | Установить Ubuntu 24.04 LTS (или 22.04) в VirtualBox. Настроить мостовой сетевой адаптер (Bridge). Написать программу, которая выводит PID процесса (через getpid()) и текущее время (через time()). Скомпилировать GCC, запустить. Проверить наличие сетевого подключения (команда ip a). |
| 4 | Вместо VirtualBox использовать VMware Workstation Player. Установить Ubuntu (любую LTS). Настроить общие папки между хостом и гостевой ОС. Написать программу, выводящую PID (getpid()) и текущее время (time()). Скомпилировать и запустить. Продемонстрировать доступ к файлам из общей папки. |
| 5 | Установить Ubuntu 22.04 LTS. Настроить SSH-сервер (sudo apt install openssh-server). Подключиться с хост-машины к виртуальной машине по SSH (используя NAT с пробросом портов или мостовой адаптер). Выполнить удалённую компиляцию (через SSH) программы на C, выводящей «SSH connection successful». Запустить программу удалённо. |
| 6 | Установить редактор VS Code (скачать .deb пакет с официального сайта, установить через sudo dpkg -i <файл>.deb, при необходимости исправить зависимости sudo apt install -f). Написать программу «Hello, World!», собрать её из встроенного терминала VS Code (или используя задачи Tasks). Сделать скриншот окна VS Code с кодом и результатом сборки. |
| 7 | Создать нового пользователя developer с домашней папкой (sudo adduser developer). Настроить для него права sudo (добавить в группу sudo). Войти под пользователем developer, создать |

| | |
|---|---|
| | программу, которая выводит имя текущего пользователя (getlogin() или getenv("USER")). Скомпилировать и запустить. Продемонстрировать, что без sudo программа работает, а с sudo — тоже (если требуется). |
| 8 | Установить Valgrind (sudo apt install valgrind). Написать программу на C с намеренной утечкой памяти (выделить блок через malloc, не освобождать free). Проверить программу с помощью valgrind --leak-check=full ./program. Исправить утечку, повторно проверить. В отчёте привести вывод Valgrind до и после исправления. |
| 9 | Настроить статический IP-адрес для Host-Only сети в VirtualBox. Создать вторую сетевую карту ВМ с типом «Host-Only». Настроить внутри Ubuntu статический IP (например, 192.168.56.10/24) через netplan. Проверить связь с хост-машиной (ping). Написать программу, которая выводит все IP-адреса интерфейсов (можно использовать getifaddrs() или просто вызвать system("ip a")). |
| 10 | Установить Docker (sudo apt install docker.io). Добавить своего пользователя в группу docker (sudo usermod -aG docker $USER). Запустить тестовый контейнер hello-world. Написать небольшой Dockerfile для компиляции программы на C (пример: FROM gcc:latest, скопировать hello.c, скомпилировать). Собрать образ и запустить контейнер. Вывести результат. |

**Обратить внимание:**

1. Все действия должны быть задокументированы в отчёте (команды, скриншоты, вывод программ).

2. Для вариантов, требующих установки дополнительных пакетов, обязательно указывать версии установленных инструментов.

3. В случае возникновения ошибок (например, при установке Docker или настройке сети) описать их и способ решения.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами).
4. Скриншоты (обязательно):
   – Версия Ubuntu (lsb_release -a).
   – Вывод gcc --version (или clang – по варианту).
   – Каталог с исходным кодом и скомпилированной программой.
   – Результат запуска программы.
   – Сессия GDB (с точкой останова, выводом переменной).
   – Выполнение make и make clean.
   – (По варианту) скриншот, подтверждающий индивидуальное задание (например, подключение по SSH).
5. Исходный код программы (вставка текстом).
6. Выводы по работе (что сделано, какие навыки получены, возникшие проблемы и их решение).
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое системный вызов? Приведите три примера.
2. Какие компоненты входят в пакет build-essential?
3. В чём отличие NAT от сетевого моста в VirtualBox?
4. Как просмотреть список всех установленных пакетов в Ubuntu?
5. Что означают права доступа rwxr-xr-- для файла?
6. Какие команды используются для создания, копирования и удаления файлов?
7. Зачем нужен ключ -g при компиляции в GCC?
8. Как в GDB установить точку останова на строке 10?
9. Каково назначение утилиты strace?
10. Для чего в Makefile используются цели all и clean?

# Лабораторная работа №2
# Основы работы с GCC и Make

**Цель работы:** освоить процесс компиляции и сборки программ на языке C с использованием компилятора GCC и системы автоматизации сборки Make, сформировать практические навыки работы с ключами компилятора, создания многофайловых проектов и написания Makefile.

**Задачи:**

– Изучить основные ключи компилятора GCC (-c, -o, -Wall, -g, -I, -L, -l).

– Научиться выполнять раздельную компиляцию и компоновку нескольких модулей.

– Освоить создание статических библиотек (.a) и их использование.

– Изучить структуру Makefile: цели, зависимости, правила, переменные.

– Научиться автоматизировать сборку проекта с помощью Make (цели all, clean, install).

– Получить навыки отладки сборки с помощью утилит make -n, make -B.

## Теоретические сведения

### Компилятор GCC

GCC (GNU Compiler Collection) – стандартный компилятор для C/C++ в Linux. Процесс компиляции состоит из четырёх этапов:

– **Препроцессинг** – обработка директив #include, #define, #ifdef (результат – .i файл).

– **Компиляция** – преобразование кода в ассемблерный текст (.s).

– **Ассемблирование** – превращение ассемблерного кода в объектный файл (.o).

– **Компоновка (линковка)** – объединение объектных файлов и библиотек в исполняемый файл.

**Основные ключи GCC**

| Ключ | Назначение |
|---|---|
| -c | Только компиляция в объектный файл (без линковки) |
| -o <файл> | Задать имя выходного файла |
| -Wall | Включить большинство предупреждений компилятора |
| -Werror | Превращать предупреждения в ошибки |
| -g | Добавить отладочную информацию для GDB |
| -O0 … -O3 | Уровни оптимизации (0 – без оптимизации) |
| -I <путь> | Добавить каталог для поиска заголовочных файлов |
| -L <путь> | Добавить каталог для поиска библиотек |
| -l<имя> | Линковать с библиотекой lib<имя>.a или lib<имя>.so |

Примеры:

**bash**

```
gcc -c main.c -o main.o          # компиляция без линковки
gcc main.o utils.o -o app        # линковка двух объектников
gcc -Wall -g main.c -o app       # полная компиляция с предупреждениями и отладкой
```

**Раздельная компиляция**

Раздельная компиляция позволяет

– ускорить сборку (перекомпилируются только изменённые файлы),

– организовать код логически (модули, библиотеки);

– использовать заголовочные файлы (.h) для объявления функций и типов.

Пример структуры проекта:


utils.h – объявления функций, utils.c – реализация, main.c – использование.

**Статические библиотеки**

Статическая библиотека (.a) – архив объектных файлов. Создаётся с помощью ar (archiver).

**bash**

```
gcc -c utils.c -o utils.o
ar rcs libutils.a utils.o      # создание библиотеки
gcc main.c -L. -lutils -o app  # линковка с библиотекой
```

**Система сборки Make**

**Make** – утилита, которая автоматически определяет, какие части программы необходимо перекомпилировать, и выполняет соответствующие команды. Правила описываются в файле Makefile (или makefile).

**Синтаксис правила:**

**makefile**

```
цель: зависимости
    команда
```

– **цель** – обычно имя файла, который будет создан (или метка, например, clean).

– **зависимости** – файлы, от которых зависит цель.

– **команда** – действие (обязательно начинается с символа табуляции).

**Переменные** в Makefile позволяют избежать повторений:

**makefile**

```
CC = gcc
CFLAGS = -Wall -g
TARGET = app
OBJS = main.o utils.o

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $@ $^

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f $(OBJS) $(TARGET)
```

**Автоматические переменные:**

– $@ – имя цели

– $^ – список всех зависимостей

– $< – первая зависимость

**Стандартные цели:**

– all – сборка всего проекта (обычно первая цель).

– clean – удаление временных файлов.

– install – установка программы в систему.

**Полезные опции make:**

– make -n – показать команды без выполнения.

– make -B – принудительная пересборка всех целей.

– make -f имя – указать файл правил.

## Методика выполнения работы

### 1. Создание простой программы на C

1.1. Создайте каталог lab2:

**bash**

```
mkdir ~/linux_labs/lab2
cd ~/linux_labs/lab2
```

1.2. Создайте файл hello.c:

**c**

```
#include <stdio.h>
int main() {
   printf("Hello, GCC!\n");
   return 0;
}
```

### 2. Компиляция с различными ключами GCC

Выполните последовательно:

```
bash
gcc hello.c -o hello1          # стандартная компиляция
./hello1

gcc -Wall -g hello.c -o hello2 # с предупреждениями и отладкой
./hello2

gcc -c hello.c -o hello.o      # только компиляция в объектный файл
file hello.o
gcc hello.o -o hello3          # линковка объектного файла
```

./hello3

### 3. Многофайловый проекта

3.1. Создайте три файла:

**math_utils.h:**

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int multiply(int a, int b);

#endif
```

**math_utils.c:**

```
#include "math_utils.h"

int add(int a, int b) {
   return a + b;
}

int multiply(int a, int b) {
   return a * b;
}
```

**main.c:**

```
#include <stdio.h>
#include "math_utils.h"

int main() {
   int x = 5, y = 3;
   printf("%d + %d = %d\n", x, y, add(x, y));
   printf("%d * %d = %d\n", x, y, multiply(x, y));
   return 0;
}
```

3.2. Выполните раздельную компиляцию и линковку вручную:

**bash**

```
gcc -Wall -c main.c -o main.o
gcc -Wall -c math_utils.c -o math_utils.o
gcc main.o math_utils.o -o calculator
./calculator
```

### 4. Создание статической библиотеки

Создайте библиотеку из math_utils.o:

**bash**

```
ar rcs libmath.a math_utils.o

# использование библиотеки

gcc main.c -L. -lmath -o calc_lib

./calc_lib
```

### 5. Написание Makefile

Создайте файл Makefile в том же каталоге:

**makefile**

```
# Компилятор и флаги
CC = gcc
CFLAGS = -Wall -g
TARGET = calculator
SRCS = main.c math_utils.c
OBJS = $(SRCS:.c=.o)

# Цель по умолчанию
all: $(TARGET)

# Линковка
$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $@ $^

# Компиляция .c в .o (неявное правило или явное)
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@
```

```
# Очистка
clean:
	rm -f $(OBJS) $(TARGET)

# Пересборка
rebuild: clean all

# Объявление phony-целей (не файлы)
.PHONY: all clean rebuild
```

Проверьте работу:

```
bash
make          # сборка
make clean    # удаление
make rebuild  # полная пересборка
```

**6. Расширенный Makefile (с библиотекой)**

Создайте ещё один Makefile для версии с библиотекой:

**makefile**

```
CC = gcc
CFLAGS = -Wall -g
LIB = libmath.a
TARGET = calc_lib

all: $(TARGET)

$(LIB): math_utils.o
	ar rcs $@ $^

$(TARGET): main.c $(LIB)
	$(CC) $(CFLAGS) main.c -L. -lmath -o $@

clean:
	rm -f *.o $(LIB) $(TARGET)

.PHONY: all clean
```

**7. Использование переменных окружения и автоматических переменных**

Добавьте в Makefile:

**makefile**

```
# Переменные для управления выводом
VERBOSE = 0
ifeq ($(VERBOSE),1)
  Q =
else
  Q = @
endif

$(TARGET): $(OBJS)
	$(Q)$(CC) $(CFLAGS) -o $@ $^
	$(Q)echo "Linking done"
```

Попробуйте make VERBOSE=1, чтобы увидеть все команды.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6 – цикл повторяется: вариант 7 → вариант 1, вариант 8 → вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Создать проект из трёх файлов: main.c, string_utils.c, string_utils.h. Реализовать функции to_upper(char *str) и to_lower(char *str). Написать Makefile с целями all, clean, test (запуск программы с предопределёнными тестами). |
| 2 | Создать статическую библиотеку libgeom.a, содержащую функции circle_area(double r) и rectangle_area(double a, double b). Написать программу, которая запрашивает у пользователя тип фигуры и вычисляет площадь. Собрать проект с использованием Makefile, где библиотека собирается отдельно. |
| 3 | Создать Makefile для проекта, содержащего не менее 4 модулей (например, main.c, io.c, calc.c, print.c). Использовать автоматические переменные, раздельные правила для компиляции и линковки. Добавить цель debug (флаги -g -O0) и цель release (флаги -O2). |

| | |
|---|---|
| 4 | Написать программу для работы с массивом (сортировка пузырьком, поиск максимума). Разделить на модули: array.c, array.h, main.c. Создать Makefile, который поддерживает сборку как с использованием объектных файлов, так и с использованием статической библиотеки (выбор через переменную USE_LIB=1). |
| 5 | Создать Makefile для проекта, который генерирует два исполняемых файла: server и client (каркасы без сетевой логики). Каждый модуль компилируется отдельно. Добавить цель docs, которая запускает doxygen (предварительно установить sudo apt install doxygen) для генерации документации. |
| 6 | Реализовать программу «Калькулятор комплексных чисел» (сложение, вычитание). Разделить на complex.c, complex.h, main.c. Создать Makefile с использованием паттернов % и автоматических переменных. Добавить цель run для запуска программы после сборки. |

**Обратить внимание:**

1. Все действия должны быть задокументированы в отчёте (команды, скриншоты, вывод программ).

2. Для вариантов, требующих установки дополнительных пакетов, указывать их версии.

3. В случае ошибок описать их и способы решения.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист(ФИО, группа, дата, название работы).

2. Цель и задачи (из методички).

3. Ход выполнения (пошаговое описание с командами).

4. Скриншоты (обязательно):

- вывод gcc --version;

- содержимое каталога после компиляции (ls -la);

- результат работы программы;

- выполнение make и make clean;

- (по варианту) скриншот, подтверждающий индивидуальное задание.

5. Исходный код программы (вставка текстом).

6. Выводы по работе (что сделано, какие навыки получены, возникшие проблемы и их решение).

7. Ответы на контрольные вопросы (письменно).

**Отчёт сдаётся преподавателю до следующего лабораторного занятия.**

## Контрольные вопросы

1. Перечислите и кратко опишите этапы компиляции программы на C компилятором GCC.

2. Для чего используются ключи -c, -o, -Wall, -g?

3. Что такое объектный файл (.o)? Чем он отличается от исполняемого?

4. Как создать статическую библиотеку (.a) и как её линковать с программой?

5. Какова структура правила в Makefile? Что такое цель, зависимости, команда?

6. Что такое автоматические переменные $@, $^, $<? Приведите примеры.

7. Зачем нужны .PHONY цели? Приведите пример.

8. Как Make определяет, нужно ли перекомпилировать файл?

9. Что произойдёт, если в Makefile перед командой поставить символ @?

10. Как можно передать переменную в Makefile из командной строки (например, CFLAGS=-O2)?

11. В чём преимущество раздельной компиляции перед компиляцией всех файлов одной командой?

12. Как указать в Makefile, чтобы программа собиралась с разными наборами флагов (например, debug и release)?

# Лабораторная работа №3
# Отладка с GDB

**Цель работы:** освоить базовые приёмы отладки программ на языке C/C++ с использованием GNU Debugger (GDB).

**Задачи:**

– Изучить процесс компиляции программы с отладочной информацией (ключ -g).

– Освоить запуск программы под управлением GDB.

– Научиться устанавливать точки останова (break, watch) и управлять ими.

– Освоить пошаговое выполнение кода (next, step, continue).

– Изучить методы просмотра значений переменных (print, display, x).

– Научиться анализировать стек вызовов (backtrace, frame, info locals).

– Освоить простейшие приёмы поиска ошибок (сегментации, бесконечных циклов, неверных значений).

## Теоретические сведения

### 1. Назначение отладчика GDB

**GDB (GNU Debugger)** – мощный инструмент для отладки программ, написанных на C, C++, Fortran, Rust и других языках. Он позволяет:

– запускать программу в контролируемой среде;

– останавливать выполнение в заданных точках;

– просматривать и изменять содержимое памяти, переменных, регистров;

– выполнять программу по шагам;

– исследовать стек вызовов;

– отслеживать изменения переменных (watchpoints).

**2. Компиляция для отладки**

Чтобы GDB мог сопоставлять машинные команды с исходным кодом, необходимо добавить отладочную информацию:

**bash**

```
gcc -g program.c -o program
```

Ключ -g включает запись имён переменных, номеров строк, типов данных. Рекомендуется также использовать -Wall для предупреждений и отключать оптимизацию (-O0), так как оптимизация может переставлять инструкции и усложнять отладку.

**bash**

```
gcc -Wall -g -O0 program.c -o program
```

**3. Основные команды GDB**

| Команда (сокращение) | Назначение |
|---|---|
| file <программа> | Загрузить исполняемый файл для отладки |
| run (r) | Запустить программу (или перезапустить) |
| break (b) | Установить точку останова: b main (функция), b 15 (строка), b file.c:20 |
| watch | Установить точку наблюдения за переменной (остановка при изменении) |
| continue (c) | Продолжить выполнение до следующей точки останова |
| next (n) | Выполнить строку (не заходя в функции) |
| step (s) | Выполнить строку (заходя в функции) |
| finish | Выполнить до выхода из текущей функции |
| print (p) | Вывести значение выражения: p x, p &x, p *ptr |
| display | Автоматически показывать выражение при каждой остановке |
| undisplay | Удалить автоматическое отображение |
| info breakpoints | Показать все точки останова |
| delete (d) | Удалить точку останова (по номеру) |
| disable/enable | Временно отключить/включить точку останова |
| backtrace (bt) | Показать стек вызовов |
| frame (f) | Переключиться на указанный кадр стека |
| info locals | Показать локальные переменные текущей функции |

| | |
|---|---|
| list (l) | Показать исходный код вокруг текущей строки |
| quit (q) | Выйти из GDB |

**4. Точки останова (breakpoints)**

Точки останова – это места, где выполнение программы приостанавливается. Виды:

– **Обычные** – на строку, функцию или адрес.

– **Условные** – срабатывают только при выполнении условия, например:

– gdb

– break 25 if i == 100

– **Точки наблюдения** (watchpoints) – останавливаются, когда значение переменной изменяется:

– gdb

– watch my_var

– **Точки перехвата** (catchpoints) – останавливаются при исключениях, вызовах системных функций и т.п.

**5. Пошаговое выполнение**

– next (n) – выполнить текущую строку. Если строка содержит вызов функции, функция выполняется целиком без остановки внутри.

– step (s) – выполнить текущую строку; при вызове функции войти в неё и остановиться на первой строке тела функции.

– until – продолжить выполнение до достижения строки с большим номером (полезно для выхода из циклов).

**6. Просмотр и изменение данных**

– print – выводит значение выражения. Можно использовать форматирование: p/x var (шестнадцатеричное), p/d var (десятичное).

– display – добавляет выражение в список автоматически отображаемых при каждой остановке.

– set variable var=значение – изменяет значение переменной во время отладки.

– x – просмотр памяти: x/10xw адрес (10 4-байтовых слов в шестнадцатеричном виде).

**7. Анализ стека**

Когда программа останавливается, backtrace показывает цепочку вызовов:

(gdb) bt

#0 function_b (x=5) at prog.c:10

#1 0x08048423 in function_a (y=2) at prog.c:20

#2 main () at prog.c:30

Команда frame 1 переключает контекст на кадр #1, после чего можно просматривать локальные переменные этого уровня с помощью info locals.

**8. Типичные ошибки, выявляемые с помощью GDB**

– **Segmentation fault (core dumped)** – остановка программы. GDB покажет строку и значения указателей.

– **Бесконечный цикл** – прерывание Ctrl+C, затем backtrace и анализ.

– **Неправильные вычисления** – просмотр переменных на разных этапах.

– **Утечки памяти** (лучше через Valgrind, но GDB может показать моменты выделения/освобождения).

## Методика выполнения работы

**1. Подготовка тестовой программы**

**1.1. Создайте каталог lab3 и файл buggy.c:**

#include <stdio.h>

#include <stdlib.h>

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int divide(int a, int b) {
    return a / b;   // потенциальное деление на ноль
}

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int idx, value, res;

    printf("Enter index (0-4): ");
    scanf("%d", &idx);
    value = arr[idx];   // возможен выход за границы
    printf("arr[%d] = %d\n", idx, value);

    printf("Enter numerator and denominator: ");
    int x, y;
    scanf("%d %d", &x, &y);
    res = divide(x, y);
    printf("%d / %d = %d\n", x, y, res);

    printf("Factorial of %d is %d\n", value, factorial(value));

    // Искусственная утечка памяти (для демонстрации watchpoint не нужна)
    int *p = (int*)malloc(sizeof(int));
```

```
    *p = 42;
    // free(p);  // забыли освободить

    return 0;
}
```

Скомпилируйте с отладочной информацией:

**bash**

```
gcc -Wall -g -O0 buggy.c -o buggy
```

### 1.2. Запуск под GDB и установка точек останова

**bash**

```
gdb ./buggy
```

Внутри GDB:

```
gdb
(gdb) break main          # точка останова на входе в main
(gdb) break 15            # строка 15 (arr[idx])
(gdb) info breakpoints    # просмотр установленных точек
(gdb) run                 # запуск программы
```

Программа остановится на main. Введите continue или c для продолжения до следующей точки.

### 1.3. Пошаговое выполнение и просмотр переменных

**gdb**

```
(gdb) break divide        # точка останова внутри divide
(gdb) continue
# Программа запросит индекс – введите, например, 2
# Остановится на строке с делением
(gdb) print a
```

(gdb) print b

(gdb) next # выполнить деление

(gdb) print res # результат

(gdb) step # выйти из функции

### 1.4. Работа со стеком вызовов

Вызовите деление на ноль (введите 5 0). GDB остановится на divide:

**gdb**

(gdb) backtrace # покажет main -> divide

(gdb) frame 1 # перейти в main

(gdb) info locals # локальные переменные main

(gdb) up # перейти вверх по стеку (к вызывающей функции)

(gdb) down # обратно вниз

### 1.5. Условные точки останова и точки наблюдения

**gdb**

(gdb) break 20 if y == 0 # остановиться, если делитель равен 0

(gdb) watch idx # остановиться при изменении idx

(gdb) continue

При изменении idx программа остановится, покажет старое и новое значение.

### 1.6. Автоматический вывод переменных

**gdb**

(gdb) display value

(gdb) display arr[0]@5 # показать 5 элементов массива

(gdb) continue

# теперь при каждой остановке GDB будет показывать эти значения

(gdb) undisplay 1 # удалить первый display

### 1.7. Поиск ошибки выхода за границы массива

Запустите программу и введите 10 (индекс вне диапазона). GDB покажет сегментацию:

**gdb**

```
(gdb) run
Enter index (0-4): 10
Program received signal SIGSEGV, Segmentation fault.
main () at buggy.c:15
15      value = arr[idx];
(gdb) print idx            # idx = 10
(gdb) print arr[0]@5       # массив из 5 элементов
```

### 1.8. Исправление ошибок на лету

Можно изменить значение переменной, чтобы избежать краха:

**gdb**

```
(gdb) set var idx = 0      # принудительно установить индекс в 0
(gdb) continue
```

Программа продолжит работу с исправленным значением.

### 1.9. Выход из GDB

**gdb**

```
(gdb) quit
```

## Пример выполнения задания

**Задача варианта:** написать программу, содержащую рекурсивную функцию для вычисления чисел Фибоначчи с ошибкой (бесконечная рекурсия). Используя GDB, обнаружить проблему, исправить ошибку.

**Шаг 1. Исходная программа с ошибкой**

Файл fib_bug.c:

```
#include <stdio.h>

// Ошибочная рекурсия: нет условия выхода для n=1
int fib(int n) {
    return fib(n-1) + fib(n-2);  // Базовый случай отсутствует!
}

int main() {
    int n;
    printf("Enter Fibonacci index: ");
    scanf("%d", &n);
    printf("fib(%d) = %d\n", n, fib(n));
    return 0;
}
```

Компиляция с отладочной информацией:

```
bash
gcc -Wall -g -O0 fib_bug.c -o fib_bug
```

**Шаг 2. Запуск под GDB и обнаружение бесконечной рекурсии**

```
bash
gdb ./fib_bug
(gdb) run
Enter Fibonacci index: 5
```

Программа зависает (бесконечная рекурсия). Нажимаем Ctrl+C:

Program received signal SIGINT, Interrupt.

fib (n=0) at fib_bug.c:5

5 return fib(n-1) + fib(n-2);

Смотрим стек вызовов:

gdb

(gdb) backtrace

#0 fib (n=0) at fib_bug.c:5

#1 0x... in fib (n=1) at fib_bug.c:5

#2 0x... in fib (n=2) at fib_bug.c:5

#3 0x... in fib (n=3) at fib_bug.c:5

#4 0x... in fib (n=4) at fib_bug.c:5

#5 0x... in fib (n=5) at fib_bug.c:5

#6 main () at fib_bug.c:12

Видно, что рекурсия ушла в отрицательные/нулевые значения без остановки.

**Шаг 3. Установка точки останова и анализ**

Установим точку останова на входе в fib и запустим заново:

gdb

(gdb) break fib

(gdb) run

Enter Fibonacci index: 5

Остановка произойдёт много раз. Посмотрим на аргумент:

gdb

(gdb) print n

$1 = 5

(gdb) continue

...

(gdb) print n

$2 = 4

...

(gdb) print n

$3 = 3

...

(gdb) print n

$4 = 2

...

(gdb) print n

$5 = 1

...

(gdb) print n

$6 = 0

...

(gdb) print n

$7 = -1

Ошибка очевидна: нет условий if (n == 0) return 0; и if (n == 1) return 1;.

**Шаг 4. Исправление программы**

Создаём исправленную версию fib_fixed.c:

```
#include <stdio.h>

int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

```
int main() {
    int n;
    printf("Enter Fibonacci index: ");
    scanf("%d", &n);
    printf("fib(%d) = %d\n", n, fib(n));
    return 0;
}
```

Компилируем и проверяем:

```
bash
gcc -Wall -g fib_fixed.c -o fib_fixed
./fib_fixed
Enter Fibonacci index: 5
fib(5) = 5
```

**Шаг 5. Повторная отладка для проверки**

```
bash
gdb ./fib_fixed
(gdb) break fib
(gdb) run
Enter Fibonacci index: 5
(gdb) continue
...
(gdb) backtrace   # на каждом вызове стек растёт, но не бесконечно
(gdb) quit
```

Программа работает корректно.

**Вывод:** с помощью GDB (прерывание Ctrl+C, backtrace, просмотр переменных) была обнаружена бесконечная рекурсия, вызванная отсутствием базовых условий. После добавления условий программа завершается успешно.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >5– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Написать программу, которая выделяет динамический массив и заполняет его, затем обращается к элементу за пределами. С помощью GDB найти место ошибки, исправить индекс. Продемонстрировать использование watch для отслеживания изменения счётчика цикла. |
| 2 | Создать программу с функцией, которая принимает указатель и разыменовывает его без проверки (нулевой указатель). В GDB установить точку останова на входе в функцию, проверить значение указателя. Использовать step для входа в функцию, затем print для вывода. |
| 3 | Написать программу, содержащую ошибку деления на ноль. С помощью условной точки останова (break if denominator == 0) остановиться перед делением и изменить значение делителя на 1 (команда set variable). Показать, что программа корректно завершилась. |
| 4 | Реализовать программу, которая вызывает функцию compute() из main(), а та вызывает helper(). В helper допущена логическая ошибка (неправильная формула). Используя backtrace и frame, определить значения аргументов на всех уровнях. Изменить значение локальной переменной в helper и продолжить выполнение. |
| 5 | Написать программу, работающую с массивом структур. Допустить ошибку: неправильное вычисление размера при malloc (выделено меньше памяти). Запустить под GDB, выявить segfault. Использовать x для просмотра памяти до и после выделения. Исправить размер. |

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – компиляция программы с ключом -g;
   – запуск GDB;
   – установка точек останова, запуск программы;
   – пошаговое выполнение, просмотр переменных;
   – анализ стека;
   – использование точек наблюдения и условных точек;
   – исправление ошибок (если есть).
4. Листинг исходного кода (с комментариями).
5. Скриншоты (обязательные):
   – сессия GDB с установкой точек останова (info breakpoints);
   – остановка на точке останова и команда print;
   – использование backtrace;
   – пример условной точки останова или watch;
   – финальный успешный запуск программы.
6. Выводы (что освоено, какие ошибки были найдены, сложности).
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Для чего нужен ключ -g при компиляции? Что произойдёт, если его не указать?

2. Как запустить программу под GDB?

3. Чем отличается next от step? Приведите пример, где использование step необходимо.

4. Как установить точку останова на строке 42 файла main.c?

5. Как просмотреть значение переменной x в GDB?

6. Что выводит команда backtrace?

7. Как переключиться на другой кадр стека и посмотреть локальные переменные?

8. Как установить точку останова, которая сработает только когда переменная i станет равна 10?

9. Как изменить значение переменной во время отладки?

10. Что такое точка наблюдения (watch)? Для чего она используется?

11. Как временно отключить точку останова, не удаляя её?

12. Как выйти из GDB?

# Лабораторная работа №4
# Низкоуровневый файловый ввод-вывод

**Цель работы:** научиться работать с файловыми дескрипторами, используя системные вызовы Linux (open, read, write, close), освоить обработку ошибок системных вызовов, сравнить производительность низкоуровневого ввода-вывода с библиотечными функциями стандартной библиотеки C.

**Задачи:**

– Изучить процесс компиляции программы с отладочной информацией (ключ -g).

– Изучить системные вызовы для работы с файлами: open, read, write, close, lseek.

– Освоить флаги открытия файлов (O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC, O_APPEND) и права доступа (mode).

– Научиться обрабатывать ошибки системных вызовов с использованием errno и perror.

– Реализовать программу копирования файла с использованием системных вызовов.

– Сравнить производительность низкоуровневого ввода-вывода и библиотечных функций (fopen/fread/fwrite).

– Изучить влияние размера буфера на эффективность копирования.

## Теоретические сведения

### 1. Файловые дескрипторы

В Linux все операции с файлами (обычные файлы, устройства, каналы, сокеты) осуществляются через **файловые дескрипторы** – целые неотрицательные числа, которые ядро выделяет при открытии файла. Стандартные дескрипторы:

– 0 – стандартный ввод (stdin)

– 1 – стандартный вывод (stdout)

– 2 – стандартный вывод ошибок (stderr)

**2. Системные вызовы для работы с файлами**

– int open(const char *pathname, int flags, mode_t mode); – открывает или создаёт файл. Возвращает файловый дескриптор или -1 при ошибке. Флаги: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT (создать, если не существует), O_TRUNC (обрезать до нуля), O_APPEND (добавление в конец). Режим (mode) задаёт права доступа (например, 0644 – rw-r--r--).

– ssize_t read(int fd, void *buf, size_t count); – читает до count байт в буфер. Возвращает количество прочитанных байт, 0 – конец файла, -1 – ошибка.

– ssize_t write(int fd, const void *buf, size_t count); – записывает до count байт из буфера. Возвращает количество записанных байт или -1.

– int close(int fd); – закрывает файловый дескриптор.

– off_t lseek(int fd, off_t offset, int whence); – перемещает указатель чтения/записи.

**3. Обработка ошибок**

Системные вызовы при ошибке возвращают -1 и устанавливают глобальную переменную errno. Для вывода сообщения об ошибке используется perror(const char *s) – выводит строку s и текстовое описание ошибки. Также можно использовать strerror(errno).

**4. Библиотечные функции ввода-вывода**

Стандартная библиотека C предоставляет fopen, fread, fwrite, fclose, которые работают с FILE *. Они используют буферизацию (пользовательский уровень), что может повысить производительность. Низкоуровневые вызовы

(без буферизации) дают более прямой доступ, но требуют ручного управления буферами.

**5. Влияние размера буфера**

При копировании файла использование слишком маленького буфера (например, 1 байт) приводит к большому количеству системных вызовов, что снижает скорость. Оптимальный размер буфера обычно кратен размеру блока файловой системы (обычно 4096 или 8192 байт).

## Методика выполнения работы

**1. Подготовка рабочего каталога и тестовых файлов**

1.1. Создайте каталог для лабораторной работы:

**bash**

```
mkdir ~/linux_labs/lab4
cd ~/linux_labs/lab4
```

1.2. Создайте тестовый файл source.txt с произвольным содержимым (например, несколько строк текста):

```
bash
echo "Hello, Linux file I/O!" > source.txt
for i in {1..100}; do echo "Line $i" >> source.txt; done
```

**2. Реализация программы копирования с использованием системных вызовов**

2.1. Создайте файл copy_sys.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <errno.h>

#define BUFFER_SIZE 4096

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int src_fd = open(argv[1], O_RDONLY);
    if (src_fd == -1) {
        perror("open source");
        exit(EXIT_FAILURE);
    }

    int dst_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dst_fd == -1) {
        perror("open destination");
        close(src_fd);
        exit(EXIT_FAILURE);
    }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read, bytes_written;

    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        bytes_written = write(dst_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            perror("write");
```

```
            close(src_fd);
            close(dst_fd);
            exit(EXIT_FAILURE);
        }
    }

    if (bytes_read == -1) {
        perror("read");
    }

    close(src_fd);
    close(dst_fd);

    printf("Copy completed successfully.\n");
    return 0;
}
```

2.2. Скомпилируйте программу:

**bash**

```
gcc -Wall -g copy_sys.c -o copy_sys
```

2.3. Выполните копирование:

**bash**

```
./copy_sys source.txt dest_sys.txt
```

2.4. Проверьте результат:

**bash**

```
diff source.txt dest_sys.txt
```

**3. Реализация копирования с библиотечными функциями**

3.1. Создайте файл copy_std.c:

```
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 4096

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    FILE *src = fopen(argv[1], "rb");
    if (!src) {
        perror("fopen source");
        exit(EXIT_FAILURE);
    }

    FILE *dst = fopen(argv[2], "wb");
    if (!dst) {
        perror("fopen destination");
        fclose(src);
        exit(EXIT_FAILURE);
    }

    char buffer[BUFFER_SIZE];
    size_t bytes_read;

    while ((bytes_read = fread(buffer, 1, BUFFER_SIZE, src)) > 0) {
```

```
        if (fwrite(buffer, 1, bytes_read, dst) != bytes_read) {
            perror("fwrite");
            break;
        }
    }

    fclose(src);
    fclose(dst);

    printf("Copy completed successfully.\n");
    return 0;
}
```

3.2. Скомпилируйте и выполните:

**bash**

```
gcc -Wall -g copy_std.c -o copy_std
./copy_std source.txt dest_std.txt
diff source.txt dest_std.txt
```

**4. Сравнение производительности**

4.1. Создайте большой тестовый файл (например, 100 МБ):

**bash**

```
dd if=/dev/zero of=bigfile.dat bs=1M count=100
```

4.2. Измерьте время копирования системными вызовами:

**bash**

```
time ./copy_sys bigfile.dat copy1.dat
```

4.3. Измерьте время копирования библиотечными функциями:

**bash**

```
time ./copy_std bigfile.dat copy2.dat
```

4.4. Сравните результаты. Обычно библиотечные функции работают быстрее за счёт внутренней буферизации.

**5. Исследование влияния размера буфера**

5.1. Измените в copy_sys.c значение BUFFER_SIZE на 1, 64, 512, 4096, 65536. Для каждого размера перекомпилируйте и измерьте время копирования bigfile.dat. Занесите результаты в таблицу.

5.2. Пример кода для измерения времени (можно добавить в программу или использовать time).

**6. Обработка ошибок**

6.1. Проверьте поведение программы при попытке открыть несуществующий исходный файл:

**bash**

```
./copy_sys nonexistent.txt out.txt
```

Программа должна вывести сообщение об ошибке (с помощью perror).

6.2. Проверьте, что программа корректно обрабатывает отсутствие прав на запись в целевой каталог (например, попробуйте скопировать в /root/test.txt без sudo).

## Пример выполнения задания

**Задача варианта:** реализовать программу копирования с системными вызовами, добавить поддержку копирования только первых N байт (флаг -n <bytes>). Измерить производительность для буфера 1024 байта.

**Шаг 1. Модификация программы**

Создан файл copy_limit.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    int opt;
    long limit = -1;
    while ((opt = getopt(argc, argv, "n:")) != -1) {
        if (opt == 'n') limit = atol(optarg);
    }

    if (argc - optind < 2) {
        fprintf(stderr, "Usage: %s [-n bytes] <source> <destination>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *src_path = argv[optind];
    const char *dst_path = argv[optind + 1];
```

```
    int src_fd = open(src_path, O_RDONLY);
    if (src_fd == -1) { perror("open source"); exit(EXIT_FAILURE); }

    int dst_fd = open(dst_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dst_fd == -1) { perror("open destination"); close(src_fd); exit(EXIT_FAILURE); }

    char buffer[BUFFER_SIZE];
    ssize_t bytes_read, bytes_written;
    long total_copied = 0;

    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        if (limit != -1 && total_copied + bytes_read > limit) {
            bytes_read = limit - total_copied;
        }
        bytes_written = write(dst_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) { perror("write"); break; }
        total_copied += bytes_written;
        if (limit != -1 && total_copied >= limit) break;
    }

    if (bytes_read == -1) perror("read");
    close(src_fd);
    close(dst_fd);
    printf("Copied %ld bytes.\n", total_copied);
    return 0;
}
```

**Шаг 2. Компиляция и тестирование**

**bash**

```
gcc -Wall -g copy_limit.c -o copy_limit
./copy_limit -n 1024 source.txt out.txt
```

Вывод:

```
Copied 1024 bytes.
```

Проверка размера:

bash

```
wc -c out.txt
1024 out.txt
```

**Шаг 3. Измерение производительности с буфером 1024**

**bash**

```
dd if=/dev/zero of=big.dat bs=1M count=50
time ./copy_limit big.dat copy_big.dat
```

Результат (пример):

```
real 0m0.523s
user 0m0.008s
sys  0m0.512s
```

**Вывод:** программа корректно копирует файлы, поддерживает ограничение длины, обрабатывает ошибки. Размер буфера влияет на время выполнения: для буфера 1024 байта скорость ниже, чем для 4096 (за счёт большего числа системных вызовов read/write).

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать программу копирования с системными вызовами. Добавить прогресс-бар (вывод процента выполнения). |
| 2 | Реализовать программу, которая копирует только первые N байт из исходного файла (N задаётся параметром командной строки). Сравнить время копирования с обычным копированием. |
| 3 | Написать программу, которая выводит размер файла (без использования stat – только с помощью lseek и указателя конца файла). |
| 4 | Реализовать программу «конкатенации» двух файлов в третий с использованием системных вызовов. Имена файлов задаются аргументами командной строки. |
| 5 | Создать программу, которая копирует файл, но пропускает каждую вторую строку (строки разделяются символом \n). Использовать буферизованное чтение. |
| 6 | Реализовать программу, которая измеряет скорость чтения файла с разными размерами буфера (от 64 байт до 1 МБ) и выводит таблицу результатов (размер буфера → скорость МБ/с). |

**Обратить внимание:**

- Все программы должны корректно обрабатывать ошибки (отсутствие файла, недостаток прав, ошибки чтения/записи).
- Для измерения производительности использовать clock_gettime или утилиту time.
- В отчёте привести график или таблицу зависимости скорости от размера буфера.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание тестовых файлов;
   – реализация копирования системными вызовами;
   – реализация копирования библиотечными функциями;
   – сравнение производительности;
   – исследование размера буфера;
   – обработка ошибок.
4. Скриншоты (обязательно):
   – успешное копирование файла;
   – сообщение об ошибке при открытии несуществующего файла;
   – вывод команды time для обоих способов;
   – таблица зависимости времени от размера буфера.
5. Исходный код программы (вставка текстом).
6. Выводы по работе (что сделано, какие закономерности обнаружены, какие преимущества/недостатки у системных вызовов).
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое файловый дескриптор? Какие дескрипторы открываются по умолчанию для каждого процесса?

2. Какие основные флаги используются в системном вызове open? Для чего нужен флаг O_TRUNC?

3. Как обработать ошибку системного вызова? Что такое errno и perror?

4. Чем отличаются системные вызовы read/write от библиотечных функций fread/fwrite?

5. Почему использование буфера маленького размера (например, 1 байт) замедляет копирование?

6. Какое значение возвращает read при достижении конца файла?

7. Что произойдёт, если write вернёт значение меньше, чем запрошено?

8. Для чего нужен вызов close? Что случится, если не закрыть файл?

9. Как с помощью lseek определить размер файла, не читая его?

10. В чём преимущество использования системных вызовов перед библиотечными функциями? А недостатки

# Лабораторная работа №5

# Произвольный доступ к файлам и блокировки

**Цель работы:** освоить позиционирование в файле с помощью системного вызова lseek, реализовать произвольный доступ к записям фиксированной длины, изучить механизмы кооперативных блокировок с использованием fcntl для предотвращения конфликтного доступа нескольких процессов к одному файлу.

**Задачи:**

– Изучить системный вызов lseek для перемещения указателя чтения/записи в файле.

– Реализовать программу для работы с файлом как с простой базой данных (записи фиксированной длины).

– Освоить операции записи и чтения записей по индексу (произвольный доступ).

– Изучить структуру struct flock и команды F_SETLK, F_SETLKW, F_GETLK для управления блокировками.

– Реализовать кооперативные блокировки для предотвращения одновременной записи.

– Проверить работу блокировок при одновременном запуске нескольких процессов.

## Теоретические сведения

### 1. Произвольный доступ к файлу (позиционирование)

Системный вызов lseek позволяет перемещать файловый указатель (offset) без чтения или записи данных. Прототип:

off_t lseek(int fd, off_t offset, int whence);

whence может быть:

– SEEK_SET – от начала файла;

– SEEK_CUR – от текущей позиции;

– SEEK_END – от конца файла.

Возвращает новое смещение от начала файла или -1 при ошибке.

Пример: lseek(fd, 100, SEEK_SET); – перейти к 101-му байту.

**2. Записи фиксированной длины**

При работе с файлом как с базой данных удобно использовать записи одинакового размера. Каждая запись имеет структуру (например, struct record). Доступ к i-й записи осуществляется по смещению i * sizeof(struct record). Это позволяет читать или записывать любую запись без последовательного просмотра.

**3. Файловые блокировки (file locking)**

В Linux для кооперативной блокировки используется fcntl с командой F_SETLK (неблокирующая) или F_SETLKW (блокирующая). Блокировки бывают:

– read lock (общая) – несколько процессов могут читать;

– write lock (исключительная) – только один процесс может писать.

Структура struct flock:

**c**

```
struct flock {
   short l_type;    // F_RDLCK, F_WRLCK, F_UNLCK
   short l_whence;  // SEEK_SET, SEEK_CUR, SEEK_END
   off_t l_start;   // начало блокируемой области
   off_t l_len;     // длина (0 – до конца файла)
   pid_t l_pid;    // идентификатор процесса (возвращается при F_GETLK)
};
```

**4. Команды fcntl для блокировок**

– F_SETLK – установить или снять блокировку. Если блокировка не может быть установлена (конфликт), возвращает -1 и errno = EAGAIN или EACCES.

– F_SETLKW – блокирующая версия (ждёт, пока блокировка не станет доступна).

– F_GETLK – проверить, можно ли установить блокировку (возвращает информацию о конфликтующей блокировке).

**5. Кооперативные блокировки**

Блокировки fcntl являются кооперативными (advisory) – они не принудительны. Процессы, которые не используют fcntl, могут игнорировать блокировки и писать в файл одновременно. Поэтому все процессы, работающие с файлом, должны добровольно следовать протоколу блокировок.

**6. Атомарность операций**

Комбинация lseek + read/write не является атомарной. Для атомарного перемещения и чтения/записи можно использовать pread и pwrite, которые совмещают позиционирование и ввод-вывод в одном системном вызове:

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

## Методика выполнения работы

**1. Подготовка рабочего каталога и заголовочного файла**

1.1. Создайте каталог для лабораторной работы:

**bash**

```
mkdir ~/linux_labs/lab5
cd ~/linux_labs/lab5
```

1.2. Создайте заголовочный файл record.h, определяющий структуру записи:

```
#ifndef RECORD_H
#define RECORD_H

#define RECORD_SIZE 256   // фиксированный размер записи (байт)
#define MAX_NAME 32
#define MAX_ 200

typedef struct {
   int id;
   char name[MAX_NAME];
   char [MAX_];
} Record;

#endif
```

Реализация базовых операций с записями

2.1. Создайте файл db_ops.c с функциями записи и чтения записи по индексу:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include "record.h"

int write_record(int fd, int index, const Record *rec) {
   off_t offset = index * RECORD_SIZE;
```

```
    if (lseek(fd, offset, SEEK_SET) == -1) {
        perror("lseek");
        return -1;
    }
    if (write(fd, rec, RECORD_SIZE) != RECORD_SIZE) {
        perror("write");
        return -1;
    }
    return 0;
}

int read_record(int fd, int index, Record *rec) {
    off_t offset = index * RECORD_SIZE;
    if (lseek(fd, offset, SEEK_SET) == -1) {
        perror("lseek");
        return -1;
    }
    if (read(fd, rec, RECORD_SIZE) != RECORD_SIZE) {
        perror("read");
        return -1;
    }
    return 0;
}
```

2.2. Создайте основную программу db_demo.c, демонстрирующую запись и чтение:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <string.h>
#include "record.h"

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file> [write|read]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];
    int fd = open(filename, O_RDWR | O_CREAT, 0644);
    if (fd == -1) { perror("open"); exit(EXIT_FAILURE); }

    if (argc == 2) {
        // Демонстрация: записать и прочитать несколько записей
        Record rec1 = {1, "Alice", "Hello from Alice"};
        Record rec2 = {2, "Bob", "Hi from Bob"};
        write_record(fd, 0, &rec1);
        write_record(fd, 1, &rec2);
        printf("Written records at indices 0 and 1.\n");

        Record r;
        read_record(fd, 0, &r);
        printf("Record 0: id=%d, name=%s, =%s\n", r.id, r.name, r.);
        read_record(fd, 1, &r);
        printf("Record 1: id=%d, name=%s, =%s\n", r.id, r.name, r.);
    } else if (strcmp(argv[2], "write") == 0 && argc == 5) {
        int idx = atoi(argv[3]);
        Record rec;
        rec.id = idx;
        strcpy(rec.name, argv[4]);
```

```
            strcpy(rec., "manual");
            write_record(fd, idx, &rec);
            printf("Written record at index %d\n", idx);
        } else if (strcmp(argv[2], "read") == 0 && argc == 4) {
            int idx = atoi(argv[3]);
            Record rec;
            read_record(fd, idx, &rec);
            printf("Record %d: id=%d, name=%s, =%s\n", idx, rec.id, rec.name,
rec.);
        }
        close(fd);
        return 0;
    }
```

2.3. Скомпилируйте и протестируйте:

**bash**

```
gcc -Wall -g db_ops.c db_demo.c -o db_demo
./db_demo test.db
./db_demo test.db read 0
./db_demo test.db write 2 "Charlie"
./db_demo test.db read 2
```

Реализация блокировок с помощью fcntl

3.1. Добавьте в db_ops.c функции для установки и снятия блокировки на запись (исключительной) и чтение (общей):

```
#include <fcntl.h>

int lock_record(int fd, int index, short type) {
    struct flock fl;
    fl.l_type = type;       // F_RDLCK, F_WRLCK, F_UNLCK
```

```
    fl.l_whence = SEEK_SET;
    fl.l_start = index * RECORD_SIZE;
    fl.l_len = RECORD_SIZE;
    fl.l_pid = 0;
    // F_SETLKW – блокирующая блокировка
    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl lock");
        return -1;
    }
    return 0;
}

int unlock_record(int fd, int index) {
    return lock_record(fd, index, F_UNLCK);
}
```

3.2. Модифицируйте функции write_record и read_record, чтобы они использовали блокировки:

```
int write_record_locked(int fd, int index, const Record *rec) {
    if (lock_record(fd, index, F_WRLCK) == -1) return -1;
    int ret = write_record(fd, index, rec);
    unlock_record(fd, index);
    return ret;
}

int read_record_locked(int fd, int index, Record *rec) {
    if (lock_record(fd, index, F_RDLCK) == -1) return -1;
    int ret = read_record(fd, index, rec);
    unlock_record(fd, index);
    return ret;
```

```
}
```

Проверка работы блокировок

4.1. Создайте программу lock_test.c, которая запускает два процесса, пытающихся одновременно записать в одну и ту же запись:

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

int main() {
    int fd = open("test.db", O_RDWR | O_CREAT, 0644);
    if (fd == -1) { perror("open"); exit(1); }
    pid_t pid = fork();
    if (pid == -1) { perror("fork"); exit(1); }
    if (pid == 0) {
        // child process
        Record rec = {99, "Child", "from child"};
        printf("Child: trying to write record 0...\n");
        if (lock_record(fd, 0, F_WRLCK) == -1) exit(1);
        sleep(2); // имитация долгой записи
        write_record(fd, 0, &rec);
        unlock_record(fd, 0);
        printf("Child: write done.\n");
        exit(0);
    } else {
        // parent process
        sleep(1);
```

```
        printf("Parent: trying to write record 0...\n");
        if (lock_record(fd, 0, F_WRLCK) == -1) perror("Parent lock");
        else {
            Record rec = {42, "Parent", "from parent"};
            write_record(fd, 0, &rec);
            unlock_record(fd, 0);
            printf("Parent: write done.\n");
        }
        wait(NULL);
    }
    close(fd);
    return 0;
}
```

4.2. Скомпилируйте и запустите:

**bash**

```
gcc -Wall -g db_ops.c lock_test.c -o lock_test
./lock_test
```

Наблюдайте, что второй процесс блокируется и ждёт, пока первый освободит блокировку (из-за F_SETLKW). Если заменить на F_SETLK, то при конфликте будет ошибка EAGAIN.

Использование pread/pwrite для атомарности

5.1. Реализуйте версию без lseek, используя pread и pwrite:

```
int write_record_pwrite(int fd, int index, const Record *rec) {
    off_t offset = index * RECORD_SIZE;
    if (pwrite(fd, rec, RECORD_SIZE, offset) != RECORD_SIZE) {
        perror("pwrite");
        return -1;
    }
```

```
    return 0;
}
```

Изучение кооперативности блокировок

6.1. Запустите программу, которая игнорирует блокировки (прямой write без fcntl) одновременно с программой, использующей блокировки. Убедитесь, что блокировки не защищают файл от процессов, которые их не проверяют. Это демонстрирует кооперативный характер.

## Пример выполнения задания

**Задача варианта:** реализовать программу для телефонного справочника (записи: имя, номер телефона) с произвольным доступом и блокировкой записи при обновлении. Создать два процесса: один обновляет запись, другой читает – убедиться, что читатель не видит неполностью записанные данные.

**Шаг 1. Создание структуры записи**

phonebook.h:

c

```
#ifndef PHONEBOOK_H
#define PHONEBOOK_H

#define RECORD_SIZE 128
#define NAME_LEN 32
#define PHONE_LEN 20

typedef struct {
    int id;
    char name[NAME_LEN];
    char phone[PHONE_LEN];
} PhoneRecord;

#endif
```

**Шаг 2. Реализация с блокировками**

phonebook_ops.c (фрагмент):

c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
```

```
#include "phonebook.h"

int lock_record(int fd, int index, short type) {
   struct flock fl = { .l_type = type, .l_whence = SEEK_SET,
                       .l_start = index * RECORD_SIZE, .l_len = RECORD_SIZE };
   return fcntl(fd, F_SETLKW, &fl); // блокирующая
}

int update_phone(int fd, int index, const char *new_phone) {
   if (lock_record(fd, index, F_WRLCK) == -1) return -1;
   PhoneRecord rec;
   // читаем запись
   pread(fd, &rec, RECORD_SIZE, index * RECORD_SIZE);
   strcpy(rec.phone, new_phone);
   // записываем обновлённую
   pwrite(fd, &rec, RECORD_SIZE, index * RECORD_SIZE);
   unlock_record(fd, index);
   return 0;
}
```

**Шаг 3. Демонстрация**

Программа phone_demo.c запускает два процесса:

- Родитель обновляет запись 0, засыпая внутри блокировки (имитация долгой записи).
- Ребёнок пытается прочитать запись 0 – блокируется до завершения записи.

```
c
// схематично: родитель получает WRLCK, спит 3 сек, пишет; ребёнок ждёт WRLCK (или RDLCK) – в зависимости от протокола.
```

**Шаг 4. Результат**

Вывод:

Parent: lock acquired, updating...

Child: waiting for lock...

Parent: update done.

Child: read record: name=Alice, phone=123456

Блокировка гарантирует, что человек не прочитает частично записанные данные.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >5– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Создать программу для работы с файлом как с массивом целых чисел (фиксированная длина 4 байта). Реализовать атомарное увеличение значения (read-modify-write) с блокировкой записи. |
| 2 | Реализовать программу, которая с помощью lseek находит середину файла и вставляет туда строку (сдвигая остальную часть). Использовать временный файл или ftruncate. |
| 3 | Написать две программы: писатель и читатель. Писатель блокирует запись (запись 0), спит 10 секунд, затем пишет. Читатель пытается прочитать ту же запись с общей блокировкой (RDLCK) – убедиться, что несколько читателей могут одновременно читать, но писатель ждёт. |
| 4 | Создать менеджер задач (запись: id, описание, статус). Добавить функцию удаления записи (помечать как удалённую, не сдвигая остальные). Использовать блокировки при изменении статуса. |
| 5 | Реализовать простой журнал событий (лог), куда несколько процессов могут добавлять записи. Использовать блокировку на добавление в конец файла с помощью lseek до конца и блокировку всей записи (или всего файла). |

**Обратить внимание:**

- Все операции с файлом должны проверять ошибки.
- Использовать pread/pwrite для атомарного доступа по смещению.
- Продемонстрировать кооперативный характер блокировок: показать, что, если процесс не использует fcntl, он может писать в защищённую область.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание структуры записи;
   – реализация произвольного доступа с lseek;
   – добавление блокировок с fcntl;
   – тестирование одновременного доступа (два процесса);
   – использование pread/pwrite.
4. Скриншоты (обязательно):
   – вывод программы db_demo;
   – результат работы lock_test (блокировка);
   – попытка записи без блокировки из другого процесса (если применимо).
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе (что освоено, в чём различие между lseek+write и pwrite, как работают кооперативные блокировки).
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что делает системный вызов lseek? Какие значения whence существуют?

2. Как получить текущую позицию файлового указателя без изменения?

3. Чем отличаются lseek+read от pread? Когда предпочтительнее использовать pread/pwrite?

4. Какой смысл имеют блокировки F_RDLCK и F_WRLCK? Может ли несколько процессов одновременно держать F_RDLCK?

5. В чём разница между F_SETLK и F_SETLKW?

6. Почему блокировки fcntl называют кооперативными (advisory)? Как их обойти?

7. Что произойдёт, если процесс завершится, не сняв блокировку?

8. Как с помощью fcntl проверить, заблокирована ли область?

9. Может ли один процесс установить блокировку на запись, а другой – на чтение той же области?

10. Зачем нужны блокировки при работе с файлами, к которым одновременно обращаются несколько процессов? Приведите пример проблемы, которую они решают.

# Лабораторная работа №6
# Работа с файловой системой

**Цель работы:** научиться работать с каталогами и метаданными файлов в Linux, разработать упрощённый аналог утилиты ls, реализовать рекурсивный обход каталогов, получить информацию о файлах с помощью системного вызова stat, выводить права доступа, размер, время модификации, а также реализовать поиск файлов по имени.

**Задачи:**

– Изучить системные вызовы для работы с каталогами: opendir, readdir, closedir, rewinddir.

– Освоить получение информации о файле с помощью stat, lstat, fstat.

– Научиться интерпретировать поля структуры struct stat (режим доступа, размер, временные метки, тип файла).

– Реализовать рекурсивный обход дерева каталогов.

– Разработать программу, выводящую содержимое каталога в формате, аналогичном ls -l (права, количество ссылок, владелец, группа, размер, дата, имя).

– Реализовать поиск файлов по имени (с поддержкой рекурсии и, возможно, простых шаблонов).

## Теоретические сведения

### 1. Файловая система Linux

Файловая система Linux представляет собой иерархическое дерево, корнем которого является каталог /. Каталоги содержат файлы и подкаталоги. Каждый файл идентифицируется комбинацией пути и имени.

### 2. Работа с каталогами: системные вызовы

– DIR *opendir(const char *name); – открывает каталог для чтения. Возвращает указатель на DIR или NULL при ошибке.

– struct dirent *readdir(DIR *dirp); – читает следующую запись каталога. Возвращает указатель на struct dirent (содержит поле d_name – имя файла) или NULL при ошибке/конце.

– int closedir(DIR *dirp); – закрывает каталог.

– void rewinddir(DIR *dirp); – сбрасывает позицию чтения в начало.

**3. Получение информации о файле: stat, lstat, fstat**

– int stat(const char *path, struct stat *buf); – получает информацию о файле по пути (для символической ссылки – о цели ссылки).

– int lstat(const char *path, struct stat *buf); – получает информацию о самой символической ссылке (не переходит по ней).

– int fstat(int fd, struct stat *buf); – получает информацию по открытому файловому дескриптору.

Структура struct stat содержит:

```
struct stat {
    dev_t     st_dev;      // ID устройства
    ino_t     st_ino;      // номер инода
    mode_t    st_mode;     // тип файла и права доступа
    nlink_t   st_nlink;    // количество жёстких ссылок
    uid_t     st_uid;      // ID владельца
    gid_t     st_gid;      // ID группы
    off_t     st_size;     // размер в байтах
    time_t    st_atime;    // время последнего доступа
    time_t    st_mtime;    // время последней модификации
    time_t    st_ctime;    // время последнего изменения статуса
    // ... другие поля
};
```

**4. Интерпретация st_mode**

Для определения типа файла используются макросы:

– S_ISREG(m) – обычный файл

– S_ISDIR(m) – каталог

– S_ISLNK(m) – символическая ссылка

– S_ISCHR(m) – символьное устройство

– S_ISBLK(m) – блочное устройство

– S_ISFIFO(m) – FIFO (именованный канал)

– S_ISSOCK(m) – сокет

Права доступа кодируются в младших 9 битах (S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH).

**5. Преобразование прав доступа в строку**

Пример функции для преобразования mode_t в строку типа "rwxr-xr--":

– Для каждого из трёх триад (владелец, группа, остальные) проверить биты на чтение, запись, выполнение.

– Учитывать специальные биты (setuid, setgid, sticky) – по желанию.

**6. Временные метки**

st_mtime – время последней модификации содержимого. Для вывода в читаемом виде используется ctime(&st.st_mtime) или localtime и strftime.

**7. Рекурсивный обход каталогов**

Алгоритм:

- Открыть каталог.
- Для каждой записи (кроме . и ..):
  - Сформировать полный путь (путь + "/" + имя).
  - Если запись является каталогом и требуется рекурсия – вызвать функцию рекурсивно.

        - Иначе обработать файл (вывести информацию или проверить условие поиска).
    - Закрыть каталог.

**8. Поиск файлов по имени**

Реализуется как рекурсивный обход с сравнением имени файла с заданным шаблоном. Можно добавить поддержку * (звёздочка) или использовать fnmatch из <fnmatch.h>.

## Методика выполнения работы

**1. Подготовка рабочего каталога и тестовой структуры**

1.1. Создайте каталог для лабораторной работы:

bash

```
mkdir ~/linux_labs/lab6
cd ~/linux_labs/lab6
```

1.2. Создайте тестовую иерархию каталогов и файлов:

bash

```
mkdir -p testdir/subdir1 testdir/subdir2
echo "Hello" > testdir/file1.txt
echo "World" > testdir/subdir1/file2.txt
ln -s file1.txt testdir/symlink
touch testdir/subdir2/empty
chmod 600 testdir/subdir2/empty
```

**2. Реализация базовой версии myls (вывод содержимого текущего каталога)**

2.1. Создайте файл myls.c:

c

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <string.h>

void print_permissions(mode_t mode) {
    printf( (mode & S_IRUSR) ? "r" : "-");
    printf( (mode & S_IWUSR) ? "w" : "-");
    printf( (mode & S_IXUSR) ? "x" : "-");
    printf( (mode & S_IRGRP) ? "r" : "-");
    printf( (mode & S_IWGRP) ? "w" : "-");
    printf( (mode & S_IXGRP) ? "x" : "-");
    printf( (mode & S_IROTH) ? "r" : "-");
    printf( (mode & S_IWOTH) ? "w" : "-");
    printf( (mode & S_IXOTH) ? "x" : "-");
}

void print_file_info(const char *path, const char *name) {
    struct stat st;
    if (lstat(path, &st) == -1) {
        perror("lstat");
        return;
    }
    // тип файла
    if (S_ISDIR(st.st_mode)) printf("d");
    else if (S_ISLNK(st.st_mode)) printf("l");
    else printf("-");
```

```
    print_permissions(st.st_mode);
    printf(" %2ld", (long)st.st_nlink);
    // владелец и группа
    struct passwd *pw = getpwuid(st.st_uid);
    struct group *gr = getgrgid(st.st_gid);
    printf(" %s %s", pw ? pw->pw_name : "?", gr ? gr->gr_name : "?");
    printf(" %8lld", (long long)st.st_size);
    // время модификации
    char timebuf[64];
    struct tm *tm = localtime(&st.st_mtime);
    strftime(timebuf, sizeof(timebuf), "%b %d %H:%M", tm);
    printf(" %s %s", timebuf, name);
    if (S_ISLNK(st.st_mode)) {
        char linkbuf[1024];
        ssize_t len = readlink(path, linkbuf, sizeof(linkbuf)-1);
        if (len != -1) {
            linkbuf[len] = '\0';
            printf(" -> %s", linkbuf);
        }
    }
    printf("\n");
}

int main(int argc, char *argv[]) {
    const char *dir = (argc > 1) ? argv[1] : ".";
    DIR *d = opendir(dir);
    if (!d) { perror("opendir"); return 1; }
    struct dirent *entry;
    while ((entry = readdir(d)) != NULL) {
        if (entry->d_name[0] == '.') continue; // скрытые файлы
```

```
            char fullpath[1024];
            snprintf(fullpath, sizeof(fullpath), "%s/%s", dir, entry->d_name);
            print_file_info(fullpath, entry->d_name);
        }
        closedir(d);
        return 0;
    }
```

2.2. Скомпилируйте и протестируйте:

bash

```
gcc -Wall -g myls.c -o myls
./myls testdir
```

**3. Добавление рекурсивной обработки (опция -R)**

3.1. Модифицируйте программу: добавьте функцию list_dir_recursive(const char *dir, int depth), которая вызывает себя для каждого подкаталога.

3.2. Используйте флаг командной строки -R. Пример кода (фрагмент):

```
    void list_dir_recursive(const char *base, const char *rel, int recursive) {
        char path[1024];
        snprintf(path, sizeof(path), "%s/%s", base, rel);
        DIR *d = opendir(path);
        if (!d) return;
        struct dirent *entry;
        while ((entry = readdir(d)) != NULL) {
            if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
continue;
            char full[1024];
            snprintf(full, sizeof(full), "%s/%s", path, entry->d_name);
```

```
        struct stat st;
        lstat(full, &st);
        // вывод информации...
        if (recursive && S_ISDIR(st.st_mode)) {
            char new_rel[1024];
            snprintf(new_rel, sizeof(new_rel), "%s/%s", rel, entry->d_name);
            list_dir_recursive(base, new_rel, recursive);
        }
    }
    closedir(d);
}
```

**4. Реализация поиска файлов по имени**

4.1. Создайте программу myfind.c, которая рекурсивно обходит каталог и выводит пути файлов, имя которых совпадает с заданным (или содержит подстроку).

4.2. Простой пример (поиск точного совпадения):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>

void find_file(const char *dir, const char *target) {
    DIR *d = opendir(dir);
    if (!d) return;
    struct dirent *entry;
    while ((entry = readdir(d)) != NULL) {
```

```
            if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
continue;
            char path[1024];
            snprintf(path, sizeof(path), "%s/%s", dir, entry->d_name);
            struct stat st;
            lstat(path, &st);
            if (strcmp(entry->d_name, target) == 0) {
                printf("%s\n", path);
            }
            if (S_ISDIR(st.st_mode)) {
                find_file(path, target);
            }
        }
        closedir(d);
    }

    int main(int argc, char *argv[]) {
        if (argc != 3) {
            fprintf(stderr, "Usage: %s <directory> <filename>\n", argv[0]);
            return 1;
        }
        find_file(argv[1], argv[2]);
        return 0;
    }
```

4.3. Улучшите программу: добавьте поддержку частичного совпадения (опция -name "*pattern*"), используйте strstr или fnmatch.

**5. Вывод прав доступа в символьном виде**

5.1. Реализуйте функцию, которая для mode_t возвращает строку типа "-rw-r--r--" (11 символов: тип + 9 битов прав). Для типа файла использовать S_ISDIR и т.д.

**6. Вывод времени модификации в удобном формате**

6.1. Используйте localtime и strftime. Пример:

char buf[64];

strftime(buf, sizeof(buf), "%Y-%m-%d %H:%M:%S", localtime(&st.st_mtime));

**7. Обработка скрытых файлов**

7.1. По умолчанию пропускать имена, начинающиеся с . (кроме . и ..). Добавить опцию -a для показа всех файлов.

## Пример выполнения задания

**Задача варианта:** разработать программу myls с поддержкой опций -l (подробный вывод), -R (рекурсия) и -a (включая скрытые файлы). Протестировать на созданной иерархии.

**Шаг 1. Реализация парсинга опций**

Использован getopt для обработки -l, -R, -a.

Фрагмент main():

```
int show_all = 0, long_format = 0, recursive = 0;
int opt;
while ((opt = getopt(argc, argv, "lRa")) != -1) {
   switch (opt) {
      case 'l': long_format = 1; break;
      case 'R': recursive = 1; break;
      case 'a': show_all = 1; break;
      default: fprintf(stderr, "Usage: %s [-lRa] [dir...]\n", argv[0]); exit(1);
   }
}
const char *dir = (optind < argc) ? argv[optind] : ".";
```

**Шаг 2. Функция вывода одной записи**

Доработана функция print_entry для формата -l:

```
void print_entry(const char *path, const char *name, int long_format) {
   struct stat st;
   lstat(path, &st);
   if (long_format) {
      char type = '-';
      if (S_ISDIR(st.st_mode)) type = 'd';
      else if (S_ISLNK(st.st_mode)) type = 'l';
      printf("%c", type);
      print_permissions(st.st_mode);
```

```
            printf(" %2ld", (long)st.st_nlink);
            struct passwd *pw = getpwuid(st.st_uid);
            struct group *gr = getgrgid(st.st_gid);
            printf(" %-8s %-8s", pw ? pw->pw_name : "?", gr ? gr->gr_name : "?");
            printf(" %8lld", (long long)st.st_size);
            char timebuf[32];
            strftime(timebuf, sizeof(timebuf), "%b %d %H:%M",
localtime(&st.st_mtime));
            printf(" %s ", timebuf);
            printf("%s", name);
            if (S_ISLNK(st.st_mode)) {
               char linkbuf[256];
               ssize_t len = readlink(path, linkbuf, sizeof(linkbuf)-1);
               if (len != -1) { linkbuf[len] = '\0'; printf(" -> %s", linkbuf); }
            }
            printf("\n");
         } else {
            printf("%s\n", name);
         }
      }
```

**Шаг 3. Рекурсивный обход**

Функция list_dir:

```
      void list_dir(const char *base, int long_fmt, int show_all, int recursive) {
         DIR *d = opendir(base);
         if (!d) { perror(base); return; }
         struct dirent *entry;
         while ((entry = readdir(d)) != NULL) {
            if (!show_all && entry->d_name[0] == '.') continue;
            char full[1024];
```

```
            snprintf(full, sizeof(full), "%s/%s", base, entry->d_name);
            print_entry(full, entry->d_name, long_fmt);
        }
        closedir(d);
        if (recursive) {
            DIR *d2 = opendir(base);
            while ((entry = readdir(d2)) != NULL) {
                if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") ==
0) continue;
                char full[1024];
                snprintf(full, sizeof(full), "%s/%s", base, entry->d_name);
                struct stat st;
                lstat(full, &st);
                if (S_ISDIR(st.st_mode)) {
                    printf("\n%s:\n", full);
                    list_dir(full, long_fmt, show_all, recursive);
                }
            }
            closedir(d2);
        }
    }
```

**Шаг 4. Тестирование**

```
bash
$ ./myls -lR testdir
testdir:
drwxr-xr-x 2 user user 4096 Apr 15 10:00 subdir1
drwxr-xr-x 2 user user 4096 Apr 15 10:00 subdir2
-rw-r--r-- 1 user user    6 Apr 15 10:00 file1.txt
lrwxrwxrwx 1 user user    9 Apr 15 10:00 symlink -> file1.txt
```

testdir/subdir1:

-rw-r--r-- 1 user user  6 Apr 15 10:00 file2.txt

testdir/subdir2:

-rw------- 1 user user  0 Apr 15 10:00 empty

**Вывод:** прамма успешно эмулирует основные возможности ls -lR. Реализованы рекурсивный обход, вывод прав доступа, размера, времени, обработка символических ссылок, фильтрация скрытых файлов.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать программу myls с опциями -l, -R, -a. Вывод должен быть максимально похож на стандартный ls. |
| 2 | Создать программу tree, которая выводит дерево каталогов (аналогично команде tree). Использовать рекурсию. Выводить только имена, с отступами (пробелы или \|--). |
| 3 | Написать программу find_by_size, которая ищет файлы, размер которых превышает заданный порог (в байтах). Выводить полные пути и размер. |
| 4 | Реализовать программу du (аналог утилиты du), которая для заданного каталога рекурсивно подсчитывает суммарный размер всех файлов и выводит размер каждого подкаталога в байтах/килобайтах. |
| 5 | Создать программу batch_rename, которая переименовывает все файлы в каталоге: добавляет префикс или суффикс (например, file.txt → backup_file.txt). Поддержка рекурсии. |
| 6 | Написать программу stat_all, которая для каждого файла в каталоге (и подкаталогах) выводит: inode, количество жёстких ссылок, тип файла, права, размер, дату изменения. Использовать stat и lstat. |

**Обратить внимание:**

– Все программы должны корректно обрабатывать ошибки (отсутствие прав, невозможность открыть каталог).

– Для рекурсивного обхода избегать зацикливания на символических ссылках (не переходить по ссылкам на каталоги или отслеживать посещённые пути).

– Вывод времени должен быть локализован (использовать strftime).

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание структуры записи;
   – реализация произвольного доступа с lseek;
   – добавление блокировок с fcntl;
   – тестирование одновременного доступа (два процесса);
   – использование pread/pwrite.
4. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание тестовой файловой структуры;
   – реализация базовой версии myls;
   – добавление рекурсии;
   – добавление вывода подробной информации (-l);
   – реализация поиска файлов (если предусмотрено вариантом).
5. Скриншоты (обязательно):
   – вывод ./myls testdir без опций;
   – вывод ./myls -lR testdir (или согласно варианту);
   – результат работы поисковой программы;
   – демонстрация обработки скрытых файлов (опция -a).
6. Исходный код программы (вставка текстом всех созданных файлов).
7. Выводы по работе (что освоено, какие сложности возникли при рекурсивном обходе, как интерпретировали st_mode).
8. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Какие функции используются для открытия и чтения каталога? Что такое DIR и struct dirent?

2. В чём разница между stat и lstat? Когда нужно использовать lstat?

3. Как определить, является ли файл каталогом, используя st_mode?

4. Какие макросы существуют для проверки типа файла?

5. Как получить имя владельца файла по uid? Как получить имя группы?

6. Какие биты прав доступа хранятся в st_mode? Как вывести их в виде строки rwxr-x---?

7. Как преобразовать время st_mtime в читаемый формат? Какие функции для этого используются?

8. Почему при рекурсивном обходе нужно пропускать каталоги . и ..?

9. Как обработать символическую ссылку, чтобы вывести её содержимое (куда она указывает)?

10. Какие потенциальные проблемы могут возникнуть при рекурсивном обходе больших деревьев каталогов?

# Лабораторная работа №7
# Создание процессов с fork

**Цель работы:** освоить создание и управление процессами в Linux с использованием системного вызова fork(), научиться различать поведение родительского и дочернего процессов, создавать иерархию процессов (родитель-потомок-внук), получать идентификаторы процессов (PID, PPID).

**Задачи:**

– Изучить системный вызов fork() и его поведение (возвращаемые значения, копирование памяти, открытые файловые дескрипторы).

– Научиться различать родительский и дочерний процессы по значению, возвращаемому fork().

– Получать и выводить PID (идентификатор процесса) и PPID (идентификатор родительского процесса) с помощью getpid() и getppid().

– Создавать иерархию процессов: родитель → потомок → внук (последовательный или «дерево»).

– Изучить эффект «осиротения» процессов (когда родитель завершается раньше потомка) и использование wait() для синхронизации.

– Освоить создание нескольких дочерних процессов в цикле.

## Теоретические сведения

1. **Понятие процесса в Linux**

Процесс – это выполняющаяся программа со своим адресным пространством, стеком, дескрипторами файлов и состоянием. Каждый процесс имеет уникальный идентификатор (PID – Process ID). Идентификатор родительского процесса (PPID) указывает на процесс, создавший данный процесс.

2. **Системный вызов fork()**

pid_t fork(void); – создаёт новый процесс путём копирования текущего.

– Возвращает 0 в дочернем процессе.

– Возвращает PID дочернего процесса (положительное число) в родительском процессе.

– Возвращает -1 при ошибке.

После вызова fork() оба процесса выполняют следующую инструкцию. Память, открытые файловые дескрипторы, сигналы и т.д. копируются (для файловых дескрипторов – разделяются).

3. **Различия между родителем и потомком**

– Разные PID и PPID.

– Потомок имеет собственное адресное пространство (изменения переменных в одном процессе не видны в другом).

– Открытые файловые дескрипторы разделяют общий указатель позиции в файле (наследуются).

4. **Идентификаторы процессов: getpid() и getppid()**

– pid_t getpid(void); – возвращает PID текущего процесса.

– pid_t getppid(void); – возвращает PPID текущего процесса (PID родителя).

5. **Создание иерархии процессов**

– Чтобы создать внука, дочерний процесс вызывает fork() внутри своего блока.

– Для создания нескольких дочерних процессов – fork() в цикле.

– Важно правильно обрабатывать возвращаемые значения, чтобы избежать «взрывного» размножения процессов (когда и родитель, и потомок продолжают цикл).

6. **Завершение процессов и wait()**

– Процесс завершается вызовом exit(), возвратом из main() или получением сигнала.

– Зомби-процессы: процесс завершился, но родитель не прочитал его статус.

– pid_t wait(int *status); – приостанавливает выполнение родителя до завершения любого дочернего процесса.

– pid_t waitpid(pid_t pid, int *status, int options); – ожидает конкретный процесс.

– wait(NULL) – игнорирует статус завершения.

7. **Осиротевшие процессы**

Если родитель завершается раньше потомка, то потомок становится «сиротой» и его новым родителем становится процесс init (PID=1) или systemd.

8. **Переменная окружения и fork()**

После fork() дочерний процесс получает копии переменных окружения. Изменения в одном процессе не влияют на другой.

## Методика выполнения работы

1. **Подготовка рабочего каталога**

1.1. Создайте каталог для лабораторной работы:

**bash**

```
mkdir ~/linux_labs/lab7
cd ~/linux_labs/lab7
```

2. **Простейший пример fork(): различие родителя и ребёнка**

2.1. Создайте файл simple_fork.c:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork failed");
        return 1;
    }
    if (pid == 0) {
        // дочерний процесс
        printf("Child: PID = %d, PPID = %d\n", getpid(), getppid());
    } else {
        // родительский процесс
        printf("Parent: PID = %d, Child PID = %d\n", getpid(), pid);
    }
    return 0;
}
```

2.2. Скомпилируйте и запустите несколько раз:

**bash**

```
gcc -Wall -g simple_fork.c -o simple_fork
./simple_fork
```

Обратите внимание, что порядок вывода может меняться (процессы выполняются параллельно).

3. **Создание иерархии «родитель → потомок → внук»**

3.1. Создайте файл parent_child_grand.c:

**c**

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

int main() {
    pid_t child = fork();
    if (child < 0) { perror("fork 1"); return 1; }
    if (child == 0) {
        // дочерний процесс
        printf("Child: PID = %d, PPID = %d\n", getpid(), getppid());
        pid_t grand = fork();
        if (grand < 0) { perror("fork 2"); return 1; }
        if (grand == 0) {
            // внук
            printf("Grandchild: PID = %d, PPID = %d\n", getpid(), getppid());
        } else {
            // дочерний ждёт внука
            wait(NULL);
            printf("Child: grandchild finished\n");
        }
    } else {
        // родитель ждёт дочерний процесс
        wait(NULL);
        printf("Parent: child finished\n");
    }
    return 0;
}
```

3.2. Скомпилируйте и выполните:

**bash**

```
gcc -Wall -g parent_child_grand.c -o hierarchy
./hierarchy
```

Проанализируйте PPID внука – он равен PID дочернего процесса.

4. **Создание нескольких дочерних процессов в цикле**

4.1. Напишите программу multiple_children.c, которая создаёт N дочерних процессов (N задаётся аргументом командной строки). Каждый дочерний процесс выводит свой номер и PID, а родитель ждёт завершения всех.

**c**

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

int main(int argc, char *argv[]) {
    if (argc != 2) { fprintf(stderr, "Usage: %s N\n", argv[0]); return 1; }
    int N = atoi(argv[1]);
    if (N <= 0) N = 1;
    for (int i = 0; i < N; i++) {
        pid_t pid = fork();
        if (pid < 0) { perror("fork"); return 1; }
        if (pid == 0) {
            // дочерний процесс
            printf("Child %d: PID = %d, PPID = %d\n", i+1, getpid(), getppid());
            sleep(1); // имитация работы
            exit(0);
```

```
        }
    }
    // родитель ждёт все дочерние
    for (int i = 0; i < N; i++) {
        wait(NULL);
    }
    printf("Parent: all children finished\n");
    return 0;
}
```

4.2. Скомпилируйте и запустите:

**bash**

```
gcc -Wall -g multiple_children.c -o multiple
./multiple 3
```

5. **Наблюдение за зомби и использование waitpid**

5.1. Создайте программу zombie_demo.c, в которой родитель не вызывает wait(), а дочерний завершается. Запустите в фоне и проверьте статус процесса через ps aux | grep defunct.

5.2. Исправьте программу, добавив wait() или waitpid().

6. **Эффект осиротения**

6.1. Напишите программу, в которой родитель завершается раньше дочернего. Убедитесь, что PPID дочернего процесса становится 1 (или PID init).

**c**

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

```
int main() {
    pid_t pid = fork();
    if (pid == 0) {
        sleep(2); // даём родителю завершиться
        printf("Orphan child: PID = %d, PPID = %d\n", getpid(), getppid());
    } else {
        printf("Parent: PID = %d, exiting now\n", getpid());
        // родитель завершается без wait
    }
    return 0;
}
```

Запустите программу и проверьте PPID дочернего процесса (будет 1).

## Пример выполнения задания

**Задача варианта:** создать программу, которая порождает цепочку из 5 процессов: родитель → потомок1 → потомок2 → … → потомок4. Каждый процесс выводит свой PID и PPID, затем передаёт управление следующему (с помощью fork() и wait()).

**Шаг 1. Реализация цепочки процессов**

Файл chain.c:

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

int main(int argc, char *argv[]) {
    int depth = 5;
    for (int i = 0; i < depth; i++) {
```

```
        pid_t pid = fork();
        if (pid < 0) { perror("fork"); return 1; }
        if (pid == 0) {
            // потомок
            printf("Level %d: PID = %d, PPID = %d\n", i+1, getpid(), getppid());
            if (i == depth-1) {
                printf("Last child (leaf) finished.\n");
            }
            // продолжить цикл в потомке (следующий fork)
        } else {
            // родитель ждёт потомка
            wait(NULL);
            printf("Process %d finished waiting for child\n", getpid());
            return 0; // родитель завершается после ожидания
        }
    }
    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

```
gcc -Wall -g chain.c -o chain
./chain
```

**Шаг 3. Вывод программы (пример)**

```
Level 1: PID = 1234, PPID = 5678
Level 2: PID = 1235, PPID = 1234
Level 3: PID = 1236, PPID = 1235
Level 4: PID = 1237, PPID = 1236
Level 5: PID = 1238, PPID = 1237
```

Last child (leaf) finished.

Process 1237 finished waiting for child

Process 1236 finished waiting for child

Process 1235 finished waiting for child

Process 1234 finished waiting for child

**Вывод:** каждый процесс создаёт ровно одного потомка, ждёт его завершения, затем завершается сам. Формируется линейная иерархия. Идентификаторы подтверждают правильное наследование.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Создать цепочку из N процессов (N задаётся параметром). Каждый процесс выводит свой номер в цепочке, PID и PPID. Родитель ждёт потомка. |
| 2 | Создать программу, порождающую ровно 3 дочерних процесса. Каждый дочерний процесс создаёт по 2 своих потомка (всего 6 внуков). Родитель ждёт всех детей, дети ждут внуков. Вывести дерево процессов (можно схематично). |
| 3 | Реализовать «веер»: родитель создаёт N дочерних процессов. Каждый дочерний процесс выводит свой PID и случайное число (через rand()). Родитель ждёт завершения всех детей. |
| 4 | Написать программу, демонстрирующую проблему зомби. Родитель создаёт дочерний процесс, дочерний завершается, родитель не вызывает wait(). Вывести статус зомби через system("ps aux \| grep defunct"). Затем добавить wait() и показать, что зомби исчез. |
| 5 | Создать иерархию «двоичное дерево» глубиной 3 (1 родитель → 2 детей → 4 внука). Каждый процесс выводит свой уровень и PID. Использовать fork() в цикле с аккуратным ветвлением (чтобы не размножить лишних). |
| 6 | Реализовать программу, в которой родительский процесс создаёт дочерний, затем оба процесса независимо пишут в один и тот же файл (открытый до fork()). Показать, что указатель позиции в файле разделяется. Добавить синхронизацию с помощью wait(), чтобы запись не перемешивалась. |

**Обратить внимание:**

– Проверять возвращаемое значение fork() на ошибки.

– Использовать wait() или waitpid() для предотвращения зомби.

– Понимать, что после fork() оба процесса выполняют один и тот же код; необходимо чётко разделять поведение через if (pid == 0).

– При создании нескольких процессов в цикле – сохранять PID детей в массив, чтобы потом корректно ожидать каждого.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание структуры записи;
   – реализация произвольного доступа с lseek;
   – добавление блокировок с fcntl;
   – тестирование одновременного доступа (два процесса);
   – использование pread/pwrite.
4. Ход выполнения (пошаговое описание с командами и пояснениями):
   – написание простейшей программы с fork();
   – создание иерархии родитель-потомок-внук;
   – создание нескольких дочерних процессов в цикле;
   – демонстрация зомби и осиротевших процессов (если есть в варианте);
   – анализ выводов PID/PPID.
5. Скриншоты (обязательно):
   – вывод simple_fork;
   – вывод hierarchy (цепочка или дерево);
   – вывод multiple с тремя детьми;
   – (по варианту) скриншот с зомби и после исправления.
6. Исходный код программы (вставка текстом всех созданных файлов).
7. Выводы по работе (что освоено, как различать родителя и потомка, почему важен wait(), как создавать иерархии).
8. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что возвращает fork() в родительском и дочернем процессах? Какое значение означает ошибку?

2. Как получить PID текущего процесса? Как получить PPID?

3. Что происходит с открытыми файловыми дескрипторами после fork()? Разделяются ли они или копируются?

4. Что такое зомби-процесс? Как его избежать?

5. Что такое осиротевший процесс? Кто становится его родителем?

6. Зачем нужен вызов wait()? Чем waitpid() отличается от wait()?

7. Можно ли создать процесс-внук, не создавая промежуточный процесс? Почему?

8. Как создать 5 дочерних процессов в цикле, чтобы каждый дочерний не порождал дополнительных детей?

9. Какие данные (переменные, память) копируются при fork(), а какие разделяются?

10. Напишите фрагмент кода, который создаёт 2 дочерних процесса, и родитель ждёт их завершения.

# Лабораторная работа №8

# Замена образа процесса с exec

**Цель работы:** научиться заменять исполняемый код процесса с использованием системных вызовов семейства exec, освоить комбинацию fork() и exec() для запуска внешних программ из дочерних процессов, научиться передавать аргументы командной строки и управлять окружением.

**Задачи:**

– Проверять возвращаемое значение fork() на ошибки.

– Использовать wait() или waitpid() для предотвращения зомби.

– Понимать, что после fork() оба процесса выполняют один и тот же код; необходимо чётко разделять поведение через if (pid == 0).

– При создании нескольких процессов в цикле – сохранять PID детей в массив, чтобы потом корректно ожидать каждого.

## Теоретические сведения

1. **Семейство системных вызовов exec**

Функции семейства exec заменяют текущий образ процесса (код, данные, стек, кучу) новым образом, загружаемым из исполняемого файла. Идентификатор процесса (PID) сохраняется. Успешный вызов exec никогда не возвращается – управление передаётся новой программе. При ошибке возвращается -1.

2. **Варианты функций exec**

| **Функция** | **Поиск по PATH** | **Аргументы** | **Окружение** |
|---|---|---|---|
| execl | нет | список | наследуется |
| execlp | да | список | наследуется |
| execle | нет | список | передаётся |
| execv | нет | массив | наследуется |
| execvp | да | массив | наследуется |
| execvpe | да | массив | передаётся |

l – аргументы передаются как список переменной длины (каждый аргумент отдельно, завершается NULL).

v – аргументы передаются как массив указателей на строки (вектор).

p – выполняется поиск исполняемого файла в каталогах, перечисленных в PATH.

e – можно передать массив переменных окружения.

### 3. Примеры использования

**c**

```
// execlp – с поиском по PATH, аргументы списком
execlp("ls", "ls", "-l", "-a", NULL);

// execvp – с поиском по PATH, аргументы массивом
char *args[] = {"ls", "-l", "-a", NULL};
execvp("ls", args);

// execl – полный путь, без поиска
execl("/bin/ls", "ls", "-l", NULL);

// execle – с передачей окружения
char *env[] = {"PATH=/bin", "HOME=/tmp", NULL};
execle("/bin/ls", "ls", NULL, env);
```

### 4. Комбинация fork() и exec()

Поскольку exec заменяет текущий процесс, для запуска внешней программы без завершения родительского процесса используется схема:

– Родитель вызывает fork(), создавая дочерний процесс.

– В дочернем процессе вызывается exec для запуска нужной программы.

– Родитель может ожидать завершения дочернего процесса с помощью wait() или waitpid().

–

5. **Обработка ошибок exec**

Если exec завершается ошибкой (например, файл не найден, недостаточно прав), возвращается -1, и дочерний процесс продолжает выполнение исходного кода. Поэтому после exec необходимо проверять ошибку и вызывать exit(), чтобы избежать выполнения родительского кода в дочернем процессе.

6. **Наследование файловых дескрипторов**

При вызове exec файловые дескрипторы остаются открытыми (если не установлен флаг FD_CLOEXEC). Это позволяет перенаправлять ввод/вывод дочернего процесса.

7. **Переменные окружения**

По умолчанию дочерний процесс наследует окружение родителя. Можно передать собственное окружение через функции *e (например, execle). Для получения переменной окружения используется getenv(), для установки – setenv() или putenv().

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab8

   cd ~/linux_labs/lab8

2. **Простой пример: замена образа процесса без fork()**

   2.1. Создайте файл exec_demo.c:

   c

   #include <stdio.h>

```
#include <unistd.h>

int main() {
    printf("Before exec: PID = %d\n", getpid());
    execlp("ls", "ls", "-l", NULL);
    // Этот код выполнится только если execlp завершился ошибкой
    perror("execlp failed");
    return 1;
}
```

2.2. Скомпилируйте и запустите:

bash

```
gcc -Wall -g exec_demo.c -o exec_demo
./exec_demo
```

Обратите внимание, что программа ls выполняется, а строка "Before exec" выводится, но код после execlp не выполняется (кроме случая ошибки).

3. **Комбинация fork() и exec(): запуск внешней программы из дочернего процесса**

3.1. Создайте файл fork_exec.c:

c

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

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
```

```
    }
    if (pid == 0) {
        // дочерний процесс
        printf("Child: PID = %d, executing 'ls -la'\n", getpid());
        execlp("ls", "ls", "-la", NULL);
        // Если exec вернулся – ошибка
        perror("execlp failed");
        return 1;
    } else {
        // родительский процесс
        printf("Parent: waiting for child (%d)...\n", pid);
        wait(NULL);
        printf("Parent: child finished\n");
    }
    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g fork_exec.c -o fork_exec
./fork_exec
```

4. **Передача аргументов командной строки в exec**

4.1. Создайте программу, которая принимает команду и аргументы от пользователя и запускает их через execvp.

Файл run_command.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <command> [args...]\n", argv[0]);
        return 1;
    }
    pid_t pid = fork();
    if (pid < 0) { perror("fork"); return 1; }
    if (pid == 0) {
        // передаём аргументы, начиная с argv[1]
        execvp(argv[1], argv + 1);
        perror("execvp failed");
        return 1;
    } else {
        int status;
        wait(&status);
        printf("Command finished with status %d\n", WEXITSTATUS(status));
    }
    return 0;
}
```

4.2. Скомпилируйте и протестируйте:

```
bash
gcc -Wall -g run_command.c -o run_command
./run_command ls -la
./run_command ps aux
./run_command echo Hello World
```

5. **Передача переменных окружения через execle**

5.1. Создайте файл exec_with_env.c, демонстрирующий передачу пользовательского окружения:

c

```
#include <stdio.h>
#include <unistd.h>

extern char **environ;

int main() {
   // новое окружение
   char *new_env[] = {"MY_VAR=Hello from exec", "PATH=/bin", NULL};
   printf("Parent environment: MY_VAR = %s\n", getenv("MY_VAR"));
   execle("/bin/sh", "sh", "-c", "echo $MY_VAR", NULL, new_env);
   perror("execle failed");
   return 1;
}
```

(Обратите внимание: при использовании execle окружение родителя не наследуется.)

6. **Обработка ошибок exec: проверка существования команды**

6.1. Модифицируйте run_command.c, добавив проверку статуса завершения дочернего процесса, и протестируйте с несуществующей командой.

7. **Перенаправление вывода дочернего процесса**

7.1. Создайте программу, которая перенаправляет вывод дочернего процесса в файл:

c

```
// redirect.c
```

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC,
0644);
        if (fd == -1) { perror("open"); return 1; }
        dup2(fd, STDOUT_FILENO);  // перенаправляем stdout
        close(fd);
        execlp("ls", "ls", "-l", NULL);
        perror("execlp");
        return 1;
    } else {
        wait(NULL);
        printf("Output saved to output.txt\n");
    }
    return 0;
}
```

## Пример выполнения задания

**Задача варианта:** реализовать программу my_shell, которая в цикле запрашивает команду у пользователя, создаёт дочерний процесс и выполняет введённую команду с аргументами с помощью execvp. Добавить поддержку встроенной команды exit для выхода.

**Шаг 1. Реализация простого командного интерпретатора**

Файл my_shell.c:

c

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

#define MAX_CMD 1024
#define MAX_ARGS 64

int main() {
    char line[MAX_CMD];
    char *args[MAX_ARGS];
    int background;

    while (1) {
        printf("mysh> ");
        fflush(stdout);
        if (!fgets(line, sizeof(line), stdin)) break;
        line[strcspn(line, "\n")] = '\0';  // удаляем '\n'
        if (strlen(line) == 0) continue;

        // поддержка фонового запуска (& в конце)
        background = 0;
        if (line[strlen(line)-1] == '&') {
            background = 1;
            line[strlen(line)-1] = '\0';
        }

        // встроенная команда exit
```

```
        if (strcmp(line, "exit") == 0) break;

        // разбор аргументов
        int i = 0;
        char *token = strtok(line, " ");
        while (token && i < MAX_ARGS-1) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL;
        if (i == 0) continue;

        pid_t pid = fork();
        if (pid < 0) { perror("fork"); continue; }
        if (pid == 0) {
            // дочерний процесс выполняет команду
            execvp(args[0], args);
            perror("execvp failed");
            exit(1);
        } else {
            if (!background) {
                wait(NULL);
            } else {
                printf("[background] PID %d\n", pid);
            }
        }
    }
    return 0;
}
```

**Шаг 2. Компиляция и тестирование**

```
bash
gcc -Wall -g my_shell.c -o my_shell
./my_shell
```

**Шаг 3. Пример сеанса работы**

```
mysh> ls -la
total 64
drwxr-xr-x 2 user user 4096 Apr 16 10:00 .
...
mysh> echo Hello from shell
Hello from shell
mysh> sleep 5 &
[background] PID 12345
mysh> exit
```

**Вывод:** программа успешно эмулирует базовую функциональность командной оболочки: создание процессов, выполнение внешних команд с аргументами, ожидание завершения или фоновый запуск. Использована комбинация fork() + execvp().

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать простой командный интерпретатор (my_shell) с поддержкой фонового запуска (&). Команды выполняются через execvp. |
| 2 | Написать программу, которая создаёт дочерний процесс и запускает в нём grep для поиска слова в файле. Слово и имя файла передаются как аргументы командной строки. |
| 3 | Создать программу-обёртку для утилиты ps. Добавить поддержку опций (например, -e, -f), передавая их через execlp. |
| 4 | Реализовать программу, которая последовательно запускает несколько команд (заданных в массиве строк) и выводит статус завершения каждой. Использовать fork() и waitpid(). |
| 5 | Написать программу, которая выполняет команду, переданную через аргументы командной строки, и измеряет время её выполнения (использовать gettimeofday до и после wait). |
| 6 | Создать программу, которая запускает дочерний процесс с перенаправлением его стандартного вывода в файл (имя файла задаётся аргументом). Использовать dup2 и execvp. |

**Обратить внимание:**

– Всегда проверять возвращаемое значение exec на ошибку и вызывать exit() в дочернем процессе.

– После fork() дочерний процесс должен использовать только async-signal-safe функции до exec (допустимо, но осторожно с printf).

– При использовании execle помнить, что окружение заменяется полностью, а не дополняется.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – демонстрация exec без fork (замена образа);
   – использование fork() + exec() для запуска внешней программы;
   – передача аргументов командной строки;
   – работа с переменными окружения;
   – перенаправление вывода (если есть в варианте).
4. Скриншоты (обязательно):
   – вывод exec_demo;
   – вывод fork_exec;
   – работа run_command с разными командами;
   – (по варианту) сеанс работы my_shell или другой программы.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе (что освоено, в чём разница между вариантами exec, как комбинировать с fork, как обрабатывать ошибки).
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что делает системный вызов exec? Что происходит с PID процесса после успешного exec?

2. В чём разница между execlp и execvp?

3. Какая функция exec позволяет передать собственное окружение?

4. Почему exec почти всегда используется вместе с fork?

5. Что произойдёт, если вызвать exec в родительском процессе без fork?

6. Как проверить, завершился ли exec ошибкой? Как обработать эту ошибку?

7. Как передать аргументы командной строки через execv?

8. Что такое поиск по PATH в контексте exec? Какие функции его поддерживают?

9. Как перенаправить вывод дочернего процесса в файл перед вызовом exec?

10. Может ли процесс после exec вернуться к исходному коду? Почему?

# Лабораторная работа №9

# Ожидание завершения процессов. Зомби

**Цель работы:** изучить механизмы ожидания завершения дочерних процессов, понять проблему возникновения процессов-зомби, освоить использование системных вызовов wait() и waitpid() для корректного завершения дочерних процессов, научиться обрабатывать статус завершения (код возврата, сигналы).

**Задачи:**

– Изучить системные вызовы wait() и waitpid() для ожидания завершения дочерних процессов.

– Создать процесс-зомби и проанализировать его состояние в системе.

– Научиться предотвращать появление зомби с помощью wait().

– Освоить обработку статуса завершения: макросы WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG.

– Изучить неблокирующий режим ожидания с использованием опции WNOHANG.

– Реализовать обработку нескольких дочерних процессов с помощью цикла wait().

## Теоретические сведения

1. **Состояния процесса**

Процесс в Linux может находиться в нескольких состояниях:

– **Running** (R) – выполняется или готов к выполнению.

– **Sleeping** (S) – ожидание какого-либо события (ввод-вывод, сигнал).

– **Stopped** (T) – остановлен (например, сигналом SIGSTOP).

– **Zombie** (Z) – процесс завершил выполнение, но его запись в таблице процессов сохраняется, потому что родитель не прочитал статус завершения.

2. **Процесс-зомби**

Когда дочерний процесс завершается (через exit() или возвратом из main), он переходит в состояние зомби. Ядро сохраняет минимум информации о процессе (PID, статус завершения, учёт использования ресурсов), ожидая, пока родитель вызовет wait() или waitpid(). Зомби не занимают память, но занимают место в таблице процессов. Большое количество зомби может исчерпать лимиты системы.

3. **Системный вызов wait()**

c

pid_t wait(int *status);

– Приостанавливает выполнение родительского процесса до тех пор, пока не завершится **любой** дочерний процесс.

– Возвращает PID завершившегося дочернего процесса или -1 при ошибке.

– Если указатель status не NULL, в него записывается информация о статусе завершения.

4. **Системный вызов waitpid()**

c

pid_t waitpid(pid_t pid, int *status, int options);

– Позволяет ожидать конкретный дочерний процесс (по PID).

– Опции:

- WNOHANG – не блокировать, вернуть 0, если ни один процесс не завершился.
- WUNTRACED – возвращать также остановленные процессы.

- WCONTINUED – возвращать процессы, возобновлённые сигналом SIGCONT.

– Если pid == -1, ожидает **любой** дочерний процесс (аналогично wait).

– Если pid == 0, ожидает любой процесс из той же группы процессов.

– Если pid < -1, ожидает любой процесс, чья группа процессов равна абсолютному значению pid.

5. **Макросы для анализа статуса завершения**

| **Макрос** | **Назначение** |
|---|---|
| WIFEXITED(status) | Истина, если процесс завершился нормально (через exit или return). |
| WEXITSTATUS(status) | Возвращает код завершения (младшие 8 бит). Использовать только если WIFEXITED истина. |
| WIFSIGNALED(status) | Истина, если процесс завершился из-за сигнала. |
| WTERMSIG(status) | Возвращает номер сигнала. Использовать только если WIFSIGNALED истина. |
| WIFSTOPPED(status) | Истина, если процесс был остановлен сигналом. |
| WSTOPSIG(status) | Возвращает номер сигнала, остановившего процесс. |
| WIFCONTINUED(status) | Истина, если процесс был возобновлён после остановки. |

6. **Причины возникновения зомби**

– Родитель завершился раньше потомка (осиротевший процесс) – в этом случае потомок становится сиротой и принимается процессом init (PID=1), который автоматически вызывает wait(). Зомби не остаётся.

– Родитель работает, но не вызывает wait() – дочерние процессы становятся зомби.

– Родитель вызвал wait() с опцией WNOHANG, но не обработал всех зомби.

7. **Игнорирование сигнала SIGCHLD**

Установка обработчика сигнала SIGCHLD в SIG_IGN (игнорирование) приводит к тому, что ядро автоматически удаляет завершившиеся дочерние процессы, не превращая их в зомби. Однако это поведение нестандартно и зависит от версии системы (в Linux работает).

8. **Блокирующий и неблокирующий режимы**

– wait() и waitpid() без WNOHANG блокируют родительский процесс до завершения дочернего.

- waitpid() с WNOHANG возвращает немедленно:
- PID дочернего процесса, если он завершился;
- 0, если ни один дочерний процесс не завершился;
- -1 при ошибке.

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab9

   cd ~/linux_labs/lab9

2. **Создание процесса-зомби**

   2.1. Создайте файл zombie.c:

   c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    }
    if (pid == 0) {
        // дочерний процесс
        printf("Child: PID = %d, exiting...\n", getpid());
        // дочерний процесс завершается
    } else {
        // родительский процесс
        printf("Parent: PID = %d, child PID = %d\n", getpid(), pid);
        printf("Parent: sleeping 30 seconds...\n");
        sleep(30);  // родитель не вызывает wait()
        printf("Parent: exiting\n");
    }
    return 0;
}
```

2.2. Скомпилируйте и запустите в фоне:

```
bash
gcc -Wall -g zombie.c -o zombie
./zombie &
```

2.3. Пока родитель спит, выполните в другом терминале:

bash

ps aux | grep Z

Вы должны увидеть процесс в состоянии Z (zombie/defunct).

2.4. Дождитесь завершения родительского процесса и снова проверьте – зомби должен исчезнуть (сирота принят процессом init).

3. **Предотвращение зомби с помощью wait()**

3.1. Создайте файл no_zombie.c:

c

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

int main() {
    pid_t pid = fork();
    if (pid < 0) { perror("fork"); return 1; }
    if (pid == 0) {
        printf("Child: PID = %d, exiting...\n", getpid());
        return 42;  // код возврата 42
    } else {
        int status;
        printf("Parent: waiting for child...\n");
        pid_t terminated_pid = wait(&status);
        printf("Parent: child %d terminated\n", terminated_pid);
        if (WIFEXITED(status)) {
            printf("Child exited normally with code %d\n", WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("Child terminated by signal %d\n", WTERMSIG(status));
        }
```

```
    }
    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g no_zombie.c -o no_zombie
./no_zombie
```

Проверьте через ps, что зомби не появилось.

4. **Обработка нескольких дочерних процессов**

4.1. Создайте файл multi_wait.c:

c

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

int main(int argc, char *argv[]) {
    int N = (argc > 1) ? atoi(argv[1]) : 3;
    for (int i = 0; i < N; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            printf("Child %d: PID = %d, sleeping %d sec\n", i+1, getpid(), i+1);
            sleep(i+1);
            exit(i+1);  // код возврата = i+1
        }
    }
    // ожидание всех детей
    int status;
```

```
    pid_t pid;
    while ((pid = wait(&status)) > 0) {
        printf("Parent: child %d terminated", pid);
        if (WIFEXITED(status)) {
            printf(" with code %d\n", WEXITSTATUS(status));
        }
    }
    printf("All children finished\n");
    return 0;
}
```

4.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g multi_wait.c -o multi_wait
./multi_wait 5
```

5. **Неблокирующий режим с WNOHANG**

5.1. Создайте файл nonblocking_wait.c:

c

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child: PID = %d, working 5 seconds...\n", getpid());
        sleep(5);
        printf("Child: finishing\n");
        return 0;
```

```
    } else {
        int status;
        printf("Parent: checking child without blocking...\n");
        for (int i = 0; i < 10; i++) {
            pid_t result = waitpid(pid, &status, WNOHANG);
            if (result == 0) {
                printf("Parent: child still running (check %d)\n", i+1);
            } else if (result == pid) {
                printf("Parent: child finished\n");
                break;
            } else {
                perror("waitpid");
                break;
            }
            sleep(1);
        }
    }
    return 0;
}
```

5.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g nonblocking_wait.c -o nonblocking_wait
./nonblocking_wait
```

6. **Обработка сигнала SIGCHLD**

6.1. Создайте файл sigchld_handler.c (демонстрация автоматической очистки зомби):

c

```
#include <stdio.h>
```

```
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void sigchld_handler(int sig) {
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("Handler: cleaned up child %d\n", pid);
    }
}

int main() {
    signal(SIGCHLD, sigchld_handler);
    for (int i = 0; i < 3; i++) {
        if (fork() == 0) {
            printf("Child %d: PID = %d\n", i+1, getpid());
            return i+1;
        }
    }
    sleep(5);  // даём время на обработку сигналов
    printf("Parent exiting\n");
    return 0;
}
```

## Пример выполнения задания

**Задача варианта:** написать программу, которая создаёт 3 дочерних процесса. Каждый дочерний процесс завершается с разным кодом возврата (1, 2, 3). Родительский процесс должен использовать waitpid() для ожидания каждого конкретного дочернего процесса и выводить его PID и код возврата.

**Шаг 1. Реализация программы**

Файл waitpid_demo.c:

c

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

int main() {
    pid_t pids[3];
    int codes[3] = {1, 2, 3};

    // создание дочерних процессов
    for (int i = 0; i < 3; i++) {
        pid_t pid = fork();
        if (pid < 0) { perror("fork"); return 1; }
        if (pid == 0) {
            printf("Child %d: PID = %d, exiting with code %d\n", i+1, getpid(),
codes[i]);
            return codes[i];
        }
        pids[i] = pid;
    }

    // ожидание каждого дочернего процесса
    for (int i = 0; i < 3; i++) {
        int status;
        pid_t terminated = waitpid(pids[i], &status, 0);
        if (terminated == -1) {
            perror("waitpid");
            continue;
```

```
        }
        printf("Parent: child PID = %d terminated", terminated);
        if (WIFEXITED(status)) {
            printf(" with code %d\n", WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf(" by signal %d\n", WTERMSIG(status));
        }
    }
    printf("Parent: all children processed\n");
    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

```
gcc -Wall -g waitpid_demo.c -o waitpid_demo
./waitpid_demo
```

**Шаг 3. Вывод программы (пример)**

```
Child 1: PID = 1234, exiting with code 1
Child 2: PID = 1235, exiting with code 2
Child 3: PID = 1236, exiting with code 3
Parent: child PID = 1234 terminated with code 1
Parent: child PID = 1235 terminated with code 2
Parent: child PID = 1236 terminated with code 3
Parent: all children processed
```

**Вывод:** программа демонстрирует использование waitpid() для ожидания конкретных дочерних процессов. Код возврата каждого процесса

успешно извлечён с помощью WEXITSTATUS. Зомби не возникают, так как родитель вызывает waitpid() для каждого дочернего процесса.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >5– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Создать программу, порождающую 5 дочерних процессов. Родитель должен дождаться завершения всех детей с помощью wait() в цикле. Вывести PID каждого завершившегося процесса. |
| 2 | Создать процесс-зомби и продемонстрировать его наличие в системе через ps. Затем модифицировать программу, добавив wait(), и показать, что зомби исчез. |
| 3 | Реализовать программу, в которой родитель создаёт дочерний процесс, а затем завершается раньше него. Показать, что осиротевший процесс принимается init (PPID становится 1). |
| 4 | Написать программу, использующую неблокирующий режим WNOHANG. Родитель должен проверять статус дочернего процесса каждую секунду, выводя сообщение «still running», пока дочерний не завершится. |
| 5 | Создать обработчик сигнала SIGCHLD, который вызывает waitpid() с WNOHANG для очистки всех завершившихся дочерних процессов. Создать 3 дочерних процесса и убедиться, что зомби не остаётся. |

**Обратить внимание:**

– После завершения дочернего процесса обязательно вызывать wait() или waitpid(), чтобы предотвратить зомби.

– При использовании WNOHANG необходимо проверять возвращаемое значение: 0 – детей нет, >0 – завершившийся процесс.

– В обработчике сигнала SIGCHLD следует использовать только async-signal-safe функции (например, waitpid).

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – демонстрация exec без fork (замена образа);
   – использование fork() + exec() для запуска внешней программы;
   – передача аргументов командной строки;
   – работа с переменными окружения;
   – перенаправление вывода (если есть в варианте).
4. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание зомби и проверка через ps;
   – использование wait() для предотвращения зомби;
   – обработка кода возврата через WIFEXITED/WEXITSTATUS;
   – ожидание нескольких дочерних процессов;
   – неблокирующий режим с WNOHANG (если есть в варианте);
   – обработка SIGCHLD (если есть).
5. Скриншоты (обязательно):
   – вывод ps aux | grep Z с видимым зомби;
   – вывод no_zombie с кодом возврата;
   – вывод multi_wait или программы по варианту;
   – вывод nonblocking_wait (если применимо).
6. Исходный код программы (вставка текстом всех созданных файлов).
7. Выводы по работе (что освоено, почему возникают зомби, как их предотвратить, как анализировать статус завершения).
8. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое процесс-зомби? В каком состоянии он находится и почему он возникает?

2. Какой системный вызов используется для ожидания завершения дочернего процесса?

3. Чем отличается wait() от waitpid()?

4. Как получить код возврата дочернего процесса, завершившегося нормально?

5. Что означают макросы WIFEXITED и WEXITSTATUS?

6. Как узнать, что дочерний процесс был завершён сигналом? Какой макрос для этого используется?

7. Что произойдёт, если родительский процесс завершится, не дождавшись дочернего? Станет ли дочерний процесс зомби?

8. Для чего нужна опция WNOHANG в waitpid()?

9. Как обработать завершение дочерних процессов с помощью сигнала SIGCHLD?

10. Почему зомби не занимают память, но могут быть проблемой для системы?

# Лабораторная работа №10

# Ожидание завершения процессов. Зомби

**Цель работы:** освоить механизмы отправки и обработки сигналов в Linux, научиться устанавливать обработчики сигналов с использованием signal() и sigaction(), реализовать корректное завершение программы (graceful shutdown) по сигналу, изучить асинхронную природу сигналов и ограничения, накладываемые на функции, вызываемые из обработчика.

**Задачи:**

– Изучить основные сигналы Linux: SIGINT, SIGTERM, SIGKILL, SIGUSR1, SIGUSR2, SIGCHLD, SIGALRM.

– Освоить отправку сигналов с помощью системного вызова kill() и команды kill из командной строки.

– Научиться устанавливать обработчики сигналов с помощью signal() и sigaction().

– Реализовать программу, перехватывающую SIGINT (Ctrl+C) и выполняющую корректное завершение (освобождение ресурсов, сохранение состояния).

– Изучить блокировку сигналов с использованием sigprocmask().

– Освоить асинхронно-безопасные функции и ограничения для обработчиков сигналов.

## Теоретические сведения

1. **Понятие сигнала**

   Сигнал – это асинхронное уведомление, отправляемое процессу ядром или другим процессом, о наступлении какого-либо события. Сигналы могут быть сгенерированы:

   - аппаратными исключениями (деление на ноль, ошибка сегментации);

- пользователем с помощью kill (команда или системный вызов);
- ядром (например, SIGCHLD при завершении дочернего процесса);
- программно через raise() или abort().

2. **Основные сигналы**

| Сигнал | Номер | Действие по умолчанию | Описание |
|---|---|---|---|
| SIGINT | 2 | Завершение | Прерывание с клавиатуры (Ctrl+C) |
| SIGQUIT | 3 | Завершение + core dump | Выход с клавиатуры (Ctrl+) |
| SIGKILL | 9 | Завершение | Безусловное завершение (нельзя перехватить) |
| SIGTERM | 15 | Завершение | Сигнал завершения (по умолчанию для kill) |
| SIGUSR1 | 10 | Завершение | Пользовательский сигнал №1 |
| SIGUSR2 | 12 | Завершение | Пользовательский сигнал №2 |
| SIGCHLD | 17 | Игнорирование | Дочерний процесс завершился или остановился |
| SIGALRM | 14 | Завершение | Таймер, установленный alarm() |
| SIGSTOP | 19 | Остановка | Безусловная остановка (нельзя перехватить) |
| SIGCONT | 18 | Продолжение | Продолжение остановленного процесса |

3. **Функция signal()**

c

```
void (*signal(int signum, void (*handler)(int)))(int);
```

- Устанавливает обработчик для сигнала signum.
- handler может быть:
  - SIG_IGN – игнорировать сигнал;
  - SIG_DFL – действие по умолчанию;
  - указатель на функцию-обработчик (void handler(int sig)).
- Недостатки signal(): поведение отличается в разных версиях UNIX, некоторые сигналы после вызова обработчика сбрасываются в SIG_DFL.

4. **Функция sigaction()**

   c

   int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

   Более надёжный и переносимый способ установки обработчика.

   Структура struct sigaction:

   c

   struct sigaction {
   void (*sa_handler)(int); *// обработчик*
   void (*sa_sigaction)(int, siginfo_t *, void *); *// расширенный обработчик*
   sigset_t sa_mask; *// сигналы, блокируемые во время работы обработчика*
   int sa_flags; *// флаги (SA_RESTART, SA_SIGINFO и др.)*
   };

5. **Отправка сигналов**
   - Системный вызов kill(pid_t pid, int sig); – отправляет сигнал sig процессу с идентификатором pid.
   - raise(int sig); – отправляет сигнал текущему процессу.

- alarm(unsigned int seconds); – отправляет SIGALRM через указанное количество секунд.

6. **Блокировка сигналов**
   - sigprocmask(int how, const sigset_t *set, sigset_t *oldset); – изменяет маску заблокированных сигналов.
   - how: SIG_BLOCK (добавить), SIG_UNBLOCK (удалить), SIG_SETMASK (установить).
   - Блокированные сигналы не доставляются, а ставятся в очередь (кроме SIGKILL и SIGSTOP).

7. **Асинхронно-безопасные функции**

В обработчике сигнала можно вызывать только ограниченный набор функций (async-signal-safe):

- write(), read(), open(), close() (системные вызовы)
- _exit(), kill(), signal(), sigaction()
- Нельзя вызывать printf(), malloc(), fopen() (могут привести к deadlock).

  Рекомендуется в обработчике устанавливать флаг (volatile sig_atomic_t), а основная логика обрабатывает его вне обработчика.

8. **Гонка сигналов**

Если один и тот же сигнал генерируется несколько раз до того, как будет обработан, он может быть доставлен только один раз (некоторые реализации очереди сигналов). Для надёжности используется блокировка сигналов.

9. **Graceful shutdown**

Корректное завершение программы при получении сигнала (например, SIGTERM):

- Закрыть открытые файлы.
- Освободить динамическую память.
- Сохранить состояние в файл.
- Завершить дочерние процессы.
- Вызвать exit() (не _exit(), чтобы выполнить atexit-обработчики).

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   ```
   mkdir ~/linux_labs/lab10
   cd ~/linux_labs/lab10
   ```

2. **Простая обработка сигнала с signal()**

   2.1. Создайте файл sig_int.c:

   c

   ```
   #include <stdio.h>
   #include <signal.h>
   #include <unistd.h>

   volatile sig_atomic_t interrupted = 0;

   void handle_sigint(int sig) {
      interrupted = 1;
      write(STDOUT_FILENO, "\nCaught SIGINT!\n", 16);
   }

   int main() {
      signal(SIGINT, handle_sigint);
      printf("PID = %d. Press Ctrl+C to send SIGINT\n", getpid());
   ```

```
    while (!interrupted) {
        printf("Working...\n");
        sleep(1);
    }
    printf("Graceful shutdown\n");
    return 0;
}
```

2.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g sig_int.c -o sig_int
./sig_int
```

Нажмите Ctrl+C и наблюдайте, как программа перехватывает сигнал.

3. **Надёжная обработка сигналов с sigaction()**

3.1. Создайте файл sigaction_demo.c:

c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

volatile sig_atomic_t sigusr1_received = 0;

void handler(int sig) {
    if (sig == SIGUSR1) {
        sigusr1_received = 1;
        write(STDOUT_FILENO, "SIGUSR1 received\n", 17);
    }
}
```

```
int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;     // автоматически перезапускать прерванные системные вызовы

    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("sigaction");
        return 1;
    }

    printf("PID = %d. Send SIGUSR1: kill -USR1 %d\n", getpid(), getpid());
    while (!sigusr1_received) {
        printf("Waiting for SIGUSR1...\n");
        sleep(1);
    }
    printf("Exiting\n");
    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g sigaction_demo.c -o sigaction_demo
./sigaction_demo
```

В другом терминале отправьте сигнал:

bash

```
kill -USR1 <PID>
```

4. **Graceful shutdown: обработка SIGTERM с сохранением состояния**

4.1. Создайте файл graceful.c:

c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

volatile sig_atomic_t should_exit = 0;
int data_counter = 0;

void save_state() {
    int fd = open("state.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd != -1) {
        char buf[64];
        int len = snprintf(buf, sizeof(buf), "counter = %d\n", data_counter);
        write(fd, buf, len);
        close(fd);
        write(STDOUT_FILENO, "State saved\n", 12);
    }
}

void handle_sigterm(int sig) {
    write(STDOUT_FILENO, "SIGTERM received, saving state...\n", 35);
    save_state();
    should_exit = 1;
}
```

```
int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handle_sigterm;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGTERM, &sa, NULL);
    // игнорируем SIGINT, чтобы не завершаться по Ctrl+C
    signal(SIGINT, SIG_IGN);

    printf("PID = %d. Send SIGTERM: kill %d\n", getpid(), getpid());
    while (!should_exit) {
        printf("Working... counter = %d\n", data_counter++);
        sleep(1);
    }
    printf("Program terminated gracefully\n");
    return 0;
}
```

4.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g graceful.c -o graceful
./graceful
```

В другом терминале отправьте SIGTERM:

bash

```
kill <PID>
```

Проверьте, что файл state.txt создан с сохранённым значением счётчика.

5. **Блокировка сигналов с sigprocmask**

5.1. Создайте файл block_signals.c:

```
c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handler(int sig) {
    write(STDOUT_FILENO, "Signal received!\n", 17);
}

int main() {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGUSR1, &sa, NULL);

    sigset_t block_set, old_set;
    sigemptyset(&block_set);
    sigaddset(&block_set, SIGUSR1);
    sigprocmask(SIG_BLOCK, &block_set, &old_set);
    printf("SIGUSR1 is blocked for 10 seconds. Send kill -USR1 %d\n", getpid());
    sleep(10);
    printf("Unblocking SIGUSR1...\n");
    sigprocmask(SIG_SETMASK, &old_set, NULL);
    sleep(1);
    return 0;
}
```

5.2. Запустите программу и отправьте SIGUSR1 в течение 10 секунд – сигнал не будет обработан до разблокировки.

6. **Использование alarm() и SIGALRM**

6.1. Создайте файл alarm_demo.c:

c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handle_alarm(int sig) {
    write(STDOUT_FILENO, "ALARM! Time is up!\n", 19);
}

int main() {
    signal(SIGALRM, handle_alarm);
    printf("Setting alarm for 3 seconds...\n");
    alarm(3);
    pause();  // ждём сигнал
    printf("Exiting\n");
    return 0;
}
```

## Пример выполнения задания

**Задача варианта:** реализовать программу, которая перехватывает SIGINT (Ctrl+C) и SIGTERM. При получении SIGINT программа выводит сообщение и продолжает работу (не завершается). При получении SIGTERM программа выполняет graceful shutdown: закрывает файл лога, сохраняет статистику и завершается.

**Шаг 1. Реализация программы**

Файл signal_handler.c:

c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

volatile sig_atomic_t term_received = 0;
volatile sig_atomic_t int_count = 0;
int log_fd;

void save_stats() {
    char buf[128];
    int len = snprintf(buf, sizeof(buf), "SIGINT received %d times\n", int_count);
    write(log_fd, buf, len);
    close(log_fd);
}

void handle_sigint(int sig) {
    int_count++;
    write(STDOUT_FILENO, "SIGINT ignored. Use SIGTERM to exit.\n", 37);
}

void handle_sigterm(int sig) {
    write(STDOUT_FILENO, "SIGTERM received, cleaning up...\n", 34);
    save_stats();
    term_received = 1;
}
```

```
int main() {
    log_fd = open("stats.log", O_WRONLY | O_CREAT | O_APPEND, 0644);
    if (log_fd == -1) {
        perror("open log");
        return 1;
    }

    struct sigaction sa_int, sa_term;
    memset(&sa_int, 0, sizeof(sa_int));
    sa_int.sa_handler = handle_sigint;
    sigemptyset(&sa_int.sa_mask);
    sigaction(SIGINT, &sa_int, NULL);

    memset(&sa_term, 0, sizeof(sa_term));
    sa_term.sa_handler = handle_sigterm;
    sigemptyset(&sa_term.sa_mask);
    sigaction(SIGTERM, &sa_term, NULL);

    printf("PID = %d. Press Ctrl+C (SIGINT) – ignored. kill %d (SIGTERM) – exit\n", getpid(), getpid());

    while (!term_received) {
        printf("Working... SIGINT count = %d\n", int_count);
        sleep(1);
    }
    printf("Program terminated gracefully\n");
    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

gcc -Wall -g signal_handler.c -o signal_handler

./signal_handler

**Шаг 3. Демонстрация работы**

- Нажатие Ctrl+C несколько раз – программа продолжает работу, счётчик увеличивается.
- Отправка SIGTERM из другого терминала:

  bash

  kill <PID>
- Программа сохраняет статистику в stats.log и завершается.

**Вывод:** программа демонстрирует различие между SIGINT (игнорируется, только счётчик) и SIGTERM (graceful shutdown с сохранением состояния). Использованы надёжные обработчики с sigaction, асинхронно-безопасные вызовы (write вместо printf внутри обработчиков).

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать программу, перехватывающую SIGINT и SIGTERM. SIGINT только выводит сообщение и продолжает работу. SIGTERM выполняет graceful shutdown с сохранением статистики в файл. |
| 2 | Создать программу, которая устанавливает обработчик для SIGUSR1 и SIGUSR2. При получении SIGUSR1 программа увеличивает счётчик, при SIGUSR2 – выводит текущее значение счётчика. Обработчики должны быть установлены через sigaction. |
| 3 | Написать программу, которая игнорирует SIGINT, но корректно обрабатывает SIGQUIT (Ctrl+\), выполняя очистку и завершение. Использовать sigaction с флагом SA_RESTART. |
| 4 | Реализовать программу с функцией alarm(), которая каждые 2 секунды выводит сообщение «tick». Добавить обработчик SIGALRM, переустанавливающий таймер. |
| 5 | Создать программу, демонстрирующую блокировку сигналов с помощью sigprocmask. Заблокировать SIGUSR1 на 10 секунд, затем разблокировать. Отправленные во время блокировки сигналы должны быть доставлены после разблокировки. |
| 6 | Написать программу «демон», который записывает текущее время в файл каждую секунду. При получении SIGTERM программа должна корректно закрыть файл и завершиться. Использовать sigaction и alarm или setitimer. |

**Обратить внимание:**

- В обработчиках сигналов использовать только async-signal-safe функции (write, _exit, signal, sigaction).
- Для флагов, изменяемых в обработчике, использовать тип volatile sig_atomic_t.
- Предусмотреть корректное завершение (закрытие файлов, освобождение ресурсов).

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – установка обработчика с signal();
   – установка обработчика с sigaction();
   – graceful shutdown по сигналу;
   – блокировка сигналов;
   – использование alarm() (если есть в варианте).
4. Скриншоты (обязательно):
   – вывод sig_int при нажатии Ctrl+C;
   – отправка SIGUSR1 программе sigaction_demo;
   – graceful shutdown: вывод программы и содержимое state.txt или stats.log;
   – (по варианту) работа с блокировкой сигналов.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе (что освоено, в чём разница между signal и sigaction, какие ограничения на обработчики сигналов).
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое сигнал? Какие существуют способы генерации сигналов?
2. Какие сигналы нельзя перехватить или игнорировать? Почему?
3. В чём разница между SIGINT, SIGTERM и SIGKILL?
4. Каковы основные недостатки функции signal()? Почему рекомендуется использовать sigaction()?
5. Что такое асинхронно-безопасные функции? Почему нельзя вызывать printf() в обработчике сигнала?
6. Зачем используется тип volatile sig_atomic_t для флагов в обработчиках сигналов?
7. Что делает флаг SA_RESTART в sigaction?
8. Как заблокировать сигнал с помощью sigprocmask? Зачем это нужно?
9. Что произойдёт, если во время выполнения обработчика сигнала придет ещё один сигнал?
10. Как организовать graceful shutdown (корректное завершение) программы при получении сигнала?

# Лабораторная работа №11
# Создание и завершение потоков

**Цель работы:** освоить базовые механизмы работы с потоками (threads) с использованием библиотеки pthreads, научиться создавать потоки, передавать им параметры, ожидать завершения потоков и получать результаты их выполнения, изучить параллельное выполнение задач на примере суммирования элементов массива по частям.

**Задачи:**

– Изучить функции библиотеки pthreads: pthread_create, pthread_join, pthread_exit, pthread_self.

– Научиться создавать несколько потоков, выполняющих параллельные вычисления.

– Освоить передачу параметров в потоковую функцию (через void* аргумент).

– Научиться получать результат выполнения потока через pthread_join и возвращаемое значение.

– Изучить разделение задачи на независимые подзадачи (разделение массива на блоки).

– Понять разницу между процессами и потоками (общая память vs изолированная память).

## Теоретические сведения

1. **Потоки (threads) в Linux**

Поток (нить) – это наименьшая единица выполнения, которая может быть запланирована ядром. В отличие от процессов, потоки одного процесса разделяют:

- Адресное пространство (глобальные переменные, куча)
- Открытые файловые дескрипторы

- Сигналы
- Текущий рабочий каталог

Каждый поток имеет собственный:

- Стек (локальные переменные)
- Регистры
- Состояние (ожидание, выполнение)
- Идентификатор потока (TID)

2. **Библиотека pthreads**

POSIX threads (pthreads) – стандарт для многопоточного программирования в UNIX-системах. При компиляции необходимо указывать флаг -pthread:

bash

gcc program.c -o program -pthread

3. **Основные функции pthreads**

| **Функция** | **Назначение** |
|---|---|
| pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg) | Создание нового потока. Поток начинает выполнение с функции start_routine, которой передаётся arg. |
| pthread_join(pthread_t thread, void **retval) | Ожидание завершения указанного потока. retval – указатель на возвращаемое значение потоковой функции. |
| pthread_exit(void *retval) | Завершение текущего потока с возвратом значения retval. |
| pthread_self(void) | Возвращает идентификатор текущего потока. |
| pthread_detach(pthread_t thread) | Отсоединяет поток – его ресурсы будут автоматически освобождены при завершении (не требует pthread_join). |

4. **Структура потоковой функции**

   Потоковая функция должна иметь прототип:

   c

   void *thread_function(void *arg);

   - Принимает указатель void* (можно передать любые данные, приведя к указателю).
   - Возвращает указатель void* (можно вернуть результат, приведя к указателю).

5. **Передача параметров в поток**

Параметры передаются через последний аргумент pthread_create. Рекомендуется передавать указатель на структуру, содержащую все необходимые данные.

6. **Получение результата из потока**

Поток может вернуть результат с помощью return или pthread_exit. Родительский поток получает результат через второй аргумент pthread_join. Важно: возвращаемое значение должно быть доступно после завершения потока (не использовать указатели на локальные переменные).

7. **Разделение задачи между потоками**

При параллельных вычислениях задача разбивается на независимые части. Каждый поток обрабатывает свой фрагмент данных (например, блок массива). Результаты частичных вычислений затем объединяются.

8. **Особенности многопоточного программирования**

- Гонка данных (data race) – когда несколько потоков одновременно обращаются к одной памяти, и хотя бы один пишет. Требуется синхронизация.
- Общие ресурсы (глобальные переменные) доступны всем потокам.
- Ошибка: использование локальной переменной как возвращаемого значения (её стек уничтожается после завершения потока).

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   ```
   mkdir ~/linux_labs/lab11
   cd ~/linux_labs/lab11
   ```

2. **Создание простого потока**

   2.1. Создайте файл simple_thread.c:

   c

   ```
   #include <stdio.h>
   #include <pthread.h>
   #include <unistd.h>

   void *thread_func(void *arg) {
       int num = *(int*)arg;
       printf("Thread: received %d, TID = %lu\n", num, pthread_self());
       sleep(1);
       printf("Thread: finishing\n");
       return NULL;
   }
   ```

```
int main() {
    pthread_t thread;
    int value = 42;

    printf("Main: creating thread\n");
    if (pthread_create(&thread, NULL, thread_func, &value) != 0) {
        perror("pthread_create");
        return 1;
    }

    printf("Main: waiting for thread\n");
    pthread_join(thread, NULL);
    printf("Main: thread finished\n");
    return 0;
}
```

2.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g simple_thread.c -o simple_thread -pthread
./simple_thread
```

3. **Создание нескольких потоков**

3.1. Создайте файл multi_thread.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define NUM_THREADS 5
```

```
void *thread_work(void *arg) {
    int id = *(int*)arg;
    free(arg);  // освобождаем выделенную память для аргумента
    printf("Thread %d: TID = %lu, working...\n", id, pthread_self());
    pthread_exit((void*)(long)(id * 10));
}

int main() {
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        int *arg = malloc(sizeof(int));
        *arg = i;
        if (pthread_create(&threads[i], NULL, thread_work, arg) != 0) {
            perror("pthread_create");
            return 1;
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        void *result;
        pthread_join(threads[i], &result);
        printf("Main: thread %d finished with result %ld\n", i, (long)result);
    }

    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g multi_thread.c -o multi_thread -pthread
./multi_thread
```

4. **Параллельное суммирование массива**

4.1. Создайте файл array_sum.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define ARRAY_SIZE 1000000
#define NUM_THREADS 4

int array[ARRAY_SIZE];
long partial_sums[NUM_THREADS];

typedef struct {
    int start;
    int end;
    int thread_id;
} ThreadData;

void *sum_range(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    long sum = 0;
    for (int i = data->start; i < data->end; i++) {
        sum += array[i];
    }
    partial_sums[data->thread_id] = sum;
```

```
    printf("Thread %d: sum of indices [%d, %d) = %ld\n",
           data->thread_id, data->start, data->end, sum);
    return NULL;
}

int main() {
    // инициализация массива
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i + 1;
    }

    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = ARRAY_SIZE / NUM_THREADS;

    // создание потоков
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_data[i].start = i * chunk_size;
        thread_data[i].end = (i == NUM_THREADS - 1) ? ARRAY_SIZE : (i +
1) * chunk_size;
        thread_data[i].thread_id = i;
        pthread_create(&threads[i], NULL, sum_range, &thread_data[i]);
    }

    // ожидание завершения
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // суммирование результатов
```

```
    long total = 0;
    for (int i = 0; i < NUM_THREADS; i++) {
        total += partial_sums[i];
    }

    printf("Total sum = %ld\n", total);
    printf("Expected sum = %ld\n", (long)ARRAY_SIZE * (ARRAY_SIZE +
1) / 2);

    return 0;
}
```

4.2. Скомпилируйте и выполните:

```
bash
gcc -Wall -g array_sum.c -o array_sum -pthread
./array_sum
```

5. **Передача сложных структур в поток**

5.1. Создайте файл struct_thread.c:

```
c
#include <stdio.h>
#include <pthread.h>
#include <string.h>

typedef struct {
    char name[50];
    int iterations;
    int result;
} Task;
```

```
void *process_task(void *arg) {
    Task *task = (Task*)arg;
    printf("Processing task for: %s\n", task->name);
    task->result = 0;
    for (int i = 0; i < task->iterations; i++) {
        task->result += i;
    }
    printf("Task completed: %s result = %d\n", task->name, task->result);
    return task;
}

int main() {
    pthread_t thread;
    Task my_task;
    strcpy(my_task.name, "SumCalculator");
    my_task.iterations = 100;

    pthread_create(&thread, NULL, process_task, &my_task);
    pthread_join(thread, NULL);

    printf("Main: final result = %d\n", my_task.result);
    return 0;
}
```

## Пример выполнения задания

**Задача примера:** разработать программу для параллельного вычисления числа Пи методом Монте-Карло с использованием нескольких потоков. Каждый поток выполняет случайные бросания точки в квадрат и подсчитывает количество попаданий в четверть круга. Результаты суммируются для получения приближённого значения Пи.

**Шаг 1. Реализация программы**

Файл pi_monte_carlo.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

#define NUM_POINTS 10000000  // общее количество точек
#define NUM_THREADS 4

long hits_per_thread[NUM_THREADS];

typedef struct {
    int thread_id;
    long num_points;
    unsigned int seed;  // своё зерно для каждого потока
} ThreadData;

void *monte_carlo_pi(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    long hits = 0;

    for (long i = 0; i < data->num_points; i++) {
        double x = (double)rand_r(&data->seed) / RAND_MAX;
        double y = (double)rand_r(&data->seed) / RAND_MAX;
        if (x*x + y*y <= 1.0) {
            hits++;
        }
    }
```

```
    hits_per_thread[data->thread_id] = hits;
    printf("Thread %d: hits = %ld out of %ld\n",
        data->thread_id, hits, data->num_points);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    long points_per_thread = NUM_POINTS / NUM_THREADS;
    long total_hits = 0;

    // создание потоков
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_data[i].thread_id = i;
        thread_data[i].num_points = (i == NUM_THREADS - 1) ?
                NUM_POINTS - points_per_thread *
(NUM_THREADS - 1) :
                points_per_thread;
        thread_data[i].seed = time(NULL) ^ (i * 12345);  // уникальное зерно

        pthread_create(&threads[i], NULL, monte_carlo_pi, &thread_data[i]);
    }

    // ожидание завершения всех потоков
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
        total_hits += hits_per_thread[i];
    }
```

```
    // вычисление Пи
    double pi = 4.0 * total_hits / NUM_POINTS;
    printf("\nTotal points: %ld\n", NUM_POINTS);
    printf("Total hits: %ld\n", total_hits);
    printf("Estimated Pi = %f\n", pi);
    printf("Actual Pi = 3.14159265358979\n");

    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

```
gcc -Wall -g pi_monte_carlo.c -o pi_monte_carlo -pthread
./pi_monte_carlo
```

**Шаг 3. Пример вывода программы**

```
Thread 0: hits = 1963492 out of 2500000
Thread 1: hits = 1963501 out of 2500000
Thread 2: hits = 1963498 out of 2500000
Thread 3: hits = 1963505 out of 2500000

Total points: 10000000
Total hits: 7853996
Estimated Pi = 3.14159840
Actual Pi = 3.14159265358979
```

**Вывод**

Программа демонстрирует:

- Создание нескольких потоков с равномерным распределением нагрузки.
- Передачу каждому потоку уникальных параметров (идентификатор, количество точек, зерно для генератора случайных чисел).
- Использование rand_r() (реентерабельная версия rand) для потокобезопасности.
- Сбор результатов из каждого потока через глобальный массив и последующее объединение.
- Ускорение вычислений за счёт параллельной работы потоков на многоядерном процессоре.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Написать программу для вычисления суммы квадратов элементов массива (размер 1 000 000). Разделить массив на равные части между 4 потоками. Каждый поток вычисляет сумму квадратов своего фрагмента. Результаты суммируются в главном потоке. |
| 2 | Реализовать программу для поиска максимального элемента в массиве (размер 2 000 000) с использованием 4 потоков. Каждый поток находит максимум в своей части, затем главный поток определяет общий максимум. |
| 3 | Создать программу, в которой каждый поток вычисляет факториал числа (число передаётся как параметр). Потоки возвращают результат через pthread_exit. Главный поток выводит все результаты. |
| 4 | Написать программу для умножения матрицы на вектор. Матрица 1000×1000, вектор 1000. Разделить строки матрицы между 4 потоками. Каждый поток вычисляет свои элементы результирующего вектора. |
| 5 | Реализовать программу, которая создаёт N потоков (N задаётся аргументом командной строки). Каждый поток выводит свой номер и ждёт 1 секунду. Использовать pthread_join для синхронизации. |
| 6 | Написать программу для проверки, является ли число простым, используя параллельный перебор делителей. Разделить диапазон проверяемых делителей между 4 потоками. Если любой поток находит делитель, остальные должны завершиться (использовать флаг). |

**Обратить внимание:**

– При компиляции обязательно указывать флаг -pthread.

– Аргументы для потоков нельзя передавать как указатели на локальные переменные, которые могут разрушиться.

– Для возврата значения из потока использовать динамическое выделение памяти или глобальные переменные.

– Избегать гонок данных (несколько потоков не должны одновременно писать в одну переменную без синхронизации).

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание простого потока;
   – создание нескольких потоков;
   – параллельное суммирование массива;
   – передача параметров и получение результатов.
4. Скриншоты (обязательно):
   – вывод simple_thread;
   – вывод multi_thread;
   – вывод array_sum с проверкой правильности суммы;
   – (по варианту) вывод программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе (что освоено, как передаются параметры, как получать результаты, преимущества многопоточности).
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое поток? Чем поток отличается от процесса?

2. Какие ресурсы разделяются между потоками одного процесса, а какие являются уникальными для каждого потока?

3. Как создать новый поток с использованием библиотеки pthreads?

4. Какой флаг нужно указать при компиляции для использования pthreads?

5. Как передать данные в потоковую функцию?

6. Как получить результат выполнения потока после его завершения?

7. Для чего используется функция pthread_join? Что произойдёт, если не вызвать pthread_join?

8. Что такое pthread_detach? В каких случаях его используют?

9. Почему нельзя вернуть указатель на локальную переменную из потоковой функции?

10. Как правильно распределить работу между несколькими потоками для достижения максимальной производительности?

11. Что такое гонка данных (data race)? Приведите пример.

12. В чём преимущество использования потоков перед процессами для параллельных вычислений?

# Лабораторная работа №12

# Синхронизация потоков с мьютексами

**Цель работы:** изучить проблему гонок данных (data race) при параллельном доступе к разделяемым ресурсам, освоить использование мьютексов для синхронизации потоков, научиться предотвращать некорректные результаты при одновременном изменении общих переменных.

**Задачи:**

– Изучить проблему гонок данных на примере параллельного инкрементирования общего счётчика несколькими потоками.

– Продемонстрировать некорректные результаты при отсутствии синхронизации.

– Освоить использование мьютексов: pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_destroy.

– Научиться защищать критические секции с помощью мьютексов.

– Изучить влияние синхронизации на производительность.

– Понять понятие взаимной блокировки (deadlock) и способы её предотвращения.

## Теоретические сведения

1. **Гонка данных (data race)**

Гонка данных возникает, когда несколько потоков одновременно обращаются к одной области памяти, и хотя бы один из них выполняет запись. Результат таких операций непредсказуем и зависит от порядка выполнения потоков (недетерминирован). Гонки данных приводят к:

- Потере обновлений (lost update)
- Чтению неконсистентных данных
- Неправильным результатам вычислений
- Трудно воспроизводимым ошибкам

2. **Проблема параллельного инкрементирования**

Операция counter++ не является атомарной. Она включает три шага:

- Чтение значения из памяти в регистр
- Увеличение значения в регистре
- Запись значения обратно в память

Когда несколько потоков выполняют эти шаги одновременно, возможна ситуация:

- Поток A читает значение (например, 5)
- Поток B читает то же значение (5)
- Оба потока увеличивают до 6 и записывают обратно
- Результат – 6 вместо ожидаемых 7

3. **Мьютексы (mutexes)**

Мьютекс (mutual exclusion) – объект синхронизации, обеспечивающий исключительный доступ к разделяемому ресурсу. Только один поток может захватить мьютекс в любой момент времени.

| **Функция** | **Назначение** |
|---|---|
| pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) | Инициализация мьютекса |
| pthread_mutex_lock(pthread_mutex_t *mutex) | Захват мьютекса (блокирует, если уже захвачен) |
| pthread_mutex_trylock(pthread_mutex_t *mutex) | Попытка захвата (не блокирует, возвращает EBUSY) |
| pthread_mutex_unlock(pthread_mutex_t *mutex) | Освобождение мьютекса |
| pthread_mutex_destroy(pthread_mutex_t *mutex) | Уничтожение мьютекса |

4. **Критическая секция**

Критическая секция – участок кода, в котором происходит доступ к разделяемому ресурсу. Внутри критической секции не должно быть потоков, работающих с тем же ресурсом. Мьютекс защищает критическую секцию.

5. **Статическая и динамическая инициализация мьютексов**
    - Статическая (для глобальных мьютексов):

```
c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

    - Динамическая (для локальных и динамических):

```
c
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

6. **Взаимная блокировка (deadlock)**

Deadlock возникает, когда два или более потоков ожидают освобождения ресурсов, захваченных друг другом. Например:

- Поток A захватил мьютекс 1, ждёт мьютекс 2
- Поток B захватил мьютекс 2, ждёт мьютекс 1

Способы предотвращения:

- Всегда захватывать мьютексы в одинаковом порядке
- Использовать pthread_mutex_trylock с откатом
- Ограничивать время удержания мьютексов

7. **Атомарные операции**

Современные процессоры поддерживают атомарные операции (например, __sync_fetch_and_add в GCC). Однако мьютексы являются более универсальным и переносимым решением.

8. **Влияние синхронизации на производительность**

Мьютексы вносят накладные расходы и могут снижать степень параллелизма. Рекомендации:

- Минимизировать размер критической секции
- Не выполнять длительных операций внутри критической секции
- По возможности использовать локальные переменные

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   ```
   mkdir ~/linux_labs/lab12
   cd ~/linux_labs/lab12
   ```

2. **Демонстрация гонки данных (без синхронизации)**

   2.1. Создайте файл race_condition.c:

   c

   ```
   #include <stdio.h>
   #include <pthread.h>
   #include <stdlib.h>

   #define NUM_THREADS 10
   #define ITERATIONS 1000000

   long counter = 0;

   void *increment(void *arg) {
       for (int i = 0; i < ITERATIONS; i++) {
           counter++;
       }
   ```

```
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Expected counter: %ld\n", (long)NUM_THREADS * ITERATIONS);
    printf("Actual counter: %ld\n", counter);
    printf("Difference: %ld\n", (long)NUM_THREADS * ITERATIONS - counter);

    return 0;
}
```

2.2. Скомпилируйте и запустите несколько раз:

```
bash
gcc -Wall -g race_condition.c -o race_condition -pthread
./race_condition
./race_condition
./race_condition
```

Обратите внимание, что результаты различны и не совпадают с ожидаемым.

3. **Исправление с помощью мьютекса**

3.1. Создайте файл mutex_fix.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define NUM_THREADS 10
#define ITERATIONS 1000000

long counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
```

```
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Expected counter: %ld\n", (long)NUM_THREADS * ITERATIONS);
    printf("Actual counter: %ld\n", counter);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g mutex_fix.c -o mutex_fix -pthread
./mutex_fix
```

Теперь результат всегда правильный, но программа работает медленнее.

4. **Оптимизация: уменьшение критической секции**

4.1. Создайте файл mutex_optimized.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define NUM_THREADS 10
#define ITERATIONS 1000000
```

```
long counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *arg) {
    long local_sum = 0;

    // локальное накопление без блокировок
    for (int i = 0; i < ITERATIONS; i++) {
        local_sum++;
    }

    // однократная блокировка для добавления локальной суммы
    pthread_mutex_lock(&mutex);
    counter += local_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
    printf("Expected counter: %ld\n", (long)NUM_THREADS *
ITERATIONS);
    printf("Actual counter: %ld\n", counter);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

4.2. Сравните время выполнения mutex_fix и mutex_optimized:

```
bash
time ./mutex_fix
time ./mutex_optimized
```

5. **Использование pthread_mutex_trylock**

5.1. Создайте файл trylock_demo.c:

```
c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

void *worker1(void *arg) {
    pthread_mutex_lock(&mutex);
    printf("Worker1: locked mutex, working...\n");
    sleep(3);
    counter++;
    pthread_mutex_unlock(&mutex);
```

```
    printf("Worker1: unlocked\n");
    return NULL;
}

void *worker2(void *arg) {
    sleep(1);
    if (pthread_mutex_trylock(&mutex) == 0) {
        printf("Worker2: got lock immediately\n");
        counter++;
        pthread_mutex_unlock(&mutex);
    } else {
        printf("Worker2: mutex is locked, doing something else\n");
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, worker1, NULL);
    pthread_create(&t2, NULL, worker2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final counter: %d\n", counter);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

6. **Демонстрация взаимной блокировки (deadlock)**

   6.1. Создайте файл deadlock.c:

   c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *thread_a(void *arg) {
    pthread_mutex_lock(&mutex1);
    printf("Thread A: locked mutex1\n");
    sleep(1);
    printf("Thread A: waiting for mutex2\n");
    pthread_mutex_lock(&mutex2);
    printf("Thread A: locked mutex2\n");
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}

void *thread_b(void *arg) {
    pthread_mutex_lock(&mutex2);
    printf("Thread B: locked mutex2\n");
    sleep(1);
    printf("Thread B: waiting for mutex1\n");
    pthread_mutex_lock(&mutex1);
    printf("Thread B: locked mutex1\n");
    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);
    return NULL;
}
```

```
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_a, NULL);
    pthread_create(&t2, NULL, thread_b, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

(Программа зависнет – демонстрация deadlock)

## Пример выполнения задания

**Задача примера:** разработать программу для параллельного вычисления среднего арифметического массива чисел. Использовать мьютекс для безопасного накопления суммы. Продемонстрировать разницу между синхронизированной и несинхронизированной версиями.

**Шаг 1. Реализация программы с гонкой данных**

Файл avg_race.c:

```
c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define ARRAY_SIZE 1000000
#define NUM_THREADS 4

double array[ARRAY_SIZE];
double sum = 0;

typedef struct {
```

```
    int start;
    int end;
} ThreadData;

void init_array() {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i * 1.0;
    }
}

void *sum_range_race(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    for (int i = data->start; i < data->end; i++) {
        sum += array[i];  // критическая секция без синхронизации
    }
    return NULL;
}

int main() {
    init_array();
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = ARRAY_SIZE / NUM_THREADS;

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_data[i].start = i * chunk_size;
        thread_data[i].end = (i == NUM_THREADS - 1) ? ARRAY_SIZE : (i +
1) * chunk_size;
        pthread_create(&threads[i], NULL, sum_range_race, &thread_data[i]);
    }
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    double expected = (ARRAY_SIZE - 1) * ARRAY_SIZE / 2.0;
    printf("Expected sum: %.0f\n", expected);
    printf("Actual sum (race): %.0f\n", sum);
    printf("Difference: %.0f\n", expected - sum);

    return 0;
}
```

**Шаг 2. Реализация программы с мьютексом**

Файл avg_mutex.c:

```
c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define ARRAY_SIZE 1000000
#define NUM_THREADS 4

double array[ARRAY_SIZE];
double sum = 0;
pthread_mutex_t sum_mutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    int start;
    int end;
```

```
} ThreadData;

void init_array() {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i * 1.0;
    }
}

void *sum_range_mutex(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    double local_sum = 0;

    // локальное накопление (без блокировок)
    for (int i = data->start; i < data->end; i++) {
        local_sum += array[i];
    }

    // блокировка только для добавления локальной суммы
    pthread_mutex_lock(&sum_mutex);
    sum += local_sum;
    pthread_mutex_unlock(&sum_mutex);

    return NULL;
}

int main() {
    init_array();
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];
    int chunk_size = ARRAY_SIZE / NUM_THREADS;
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_data[i].start = i * chunk_size;
        thread_data[i].end = (i == NUM_THREADS - 1) ? ARRAY_SIZE : (i + 1) * chunk_size;
        pthread_create(&threads[i], NULL, sum_range_mutex, &thread_data[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    double expected = (ARRAY_SIZE - 1) * ARRAY_SIZE / 2.0;
    printf("Expected sum: %.0f\n", expected);
    printf("Actual sum (mutex): %.0f\n", sum);

    pthread_mutex_destroy(&sum_mutex);
    return 0;
}
```

**Шаг 3. Сравнение результатов**

```
bash
gcc -Wall -g avg_race.c -o avg_race -pthread
gcc -Wall -g avg_mutex.c -o avg_mutex -pthread

echo "=== Race condition ==="
./avg_race

echo "=== With mutex ==="
```

./avg_mutex

**Шаг 4. Пример вывода**

=== Race condition ===

Expected sum: 499999500000

Actual sum (race): 497234123456

Difference: 2765376544

=== With mutex ===

Expected sum: 499999500000

Actual sum (mutex): 499999500000

**Вывод:**

Программа наглядно демонстрирует:

- При отсутствии синхронизации результат суммирования неверен из-за гонки данных.
- Использование мьютекса для защиты критической секции даёт правильный результат.
- Оптимизация с локальным накоплением (один захват мьютекса на поток) минимизирует накладные расходы.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Написать программу, в которой 10 потоков одновременно увеличивают общий счётчик на 1 (каждый поток выполняет 100 000 итераций). Продемонстрировать некорректный результат. Исправить с помощью мьютекса. |
| 2 | Реализовать программу для параллельного вычисления суммы элементов матрицы 1000×1000. Показать гонку данных при суммировании в общую переменную. Исправить с использованием мьютекса. |
| 3 | Создать программу, в которой 5 потоков одновременно добавляют элементы в общий список (массив). Использовать мьютекс для защиты операций добавления. Сравнить время выполнения с несинхронизированной версией. |
| 4 | Написать программу для параллельного нахождения минимума в массиве. Продемонстрировать проблему гонки данных при обновлении общего минимума. Исправить с помощью мьютекса. |
| 5 | Реализовать программу, моделирующую работу банкоматов: 10 потоков одновременно снимают деньги с общего счёта. Использовать мьютекс для защиты баланса. Обеспечить проверку достаточности средств. |
| 6 | Создать программу, в которой 8 потоков обрабатывают очередь задач (задачи в глобальном массиве). Использовать мьютекс для защиты индекса текущей задачи. Каждый поток берёт следующую необработанную задачу. |

**Обратить внимание:**

- Всегда инициализировать мьютексы перед использованием.
- Освобождать мьютекс после выхода из критической секции.
- Использовать pthread_mutex_destroy для очистки ресурсов.
- Для измерения производительности использовать утилиту time.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – демонстрация гонки данных (несколько запусков, различные результаты);
   – исправление с помощью мьютекса (правильный результат);
   – оптимизация критической секции (локальные переменные);
   – демонстрация trylock;
   – (по желанию) демонстрация deadlock.
4. Скриншоты (обязательно):
   – вывод race_condition (разные результаты);
   – вывод mutex_fix (правильный результат);
   – сравнение времени выполнения mutex_fix и mutex_optimized;
   – (по варианту) вывод программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое гонка данных (data race)? Приведите пример.

2. Почему операция counter++ не является атомарной? Из каких шагов она состоит?

3. Какую проблему решают мьютексы?

4. Какие функции используются для работы с мьютексами в pthreads?

5. Что такое критическая секция? Как её защитить с помощью мьютекса?

6. В чём разница между pthread_mutex_lock и pthread_mutex_trylock?

7. Что такое взаимная блокировка (deadlock)? Как её избежать?

8. Почему использование мьютекса снижает производительность? Как минимизировать накладные расходы?

9. Как инициализировать мьютекс статически и динамически?

10. Что произойдёт, если поток попытается захватить уже захваченный им же мьютекс (без рекурсивной поддержки)?

11. Можно ли использовать мьютекс для защиты нескольких разных ресурсов?

12. Как правильно завершить работу с мьютексом?

# Лабораторная работа №13

# Условные переменные. Producer-Consumer

**Цель работы:** освоить использование условных переменных для координации потоков, реализовать классическую задачу «производитель-потребитель» с ограниченным буфером, изучить механизмы ожидания и уведомления между потоками.

**Задачи:**

– Изучить условные переменные и их роль в синхронизации потоков.

– Освоить

функции pthread_cond_init, pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_destroy.

– Реализовать задачу «производитель-потребитель» с кольцевым буфером фиксированного размера.

– Научиться корректно использовать условные переменные в связке с мьютексами.

– Изучить проблему ложных пробуждений (spurious wakeups) и способ её решения.

– Реализовать синхронизацию доступа к общему буферу с защитой от переполнения и чтения из пустого буфера.

## Теоретическая часть

1. **Условные переменные (condition variables)**

Условная переменная – механизм синхронизации, позволяющий потокам блокироваться в ожидании наступления некоторого условия и уведомлять другие потоки о его наступлении. В отличие от мьютексов, которые обеспечивают исключительный доступ, условные переменные позволяют потокам ждать изменения состояния.

2. **Задача «производитель-потребитель»**

   Классическая задача синхронизации:

   - **Производители** генерируют данные и помещают их в общий буфер.
   - **Потребители** извлекают данные из буфера и обрабатывают их.
   - **Ограниченный буфер** имеет фиксированный размер.
   - **Условия синхронизации:**
     - Производитель не должен добавлять данные в заполненный буфер.
     - Потребитель не должен извлекать данные из пустого буфера.
     - Доступ к буферу должен быть потокобезопасным.

3. **Основные функции условных переменных**

| **Функция** | **Назначение** |
|---|---|
| pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr) | Инициализация условной переменной |
| pthread_cond_destroy(pthread_cond_t *cond) | Уничтожение условной переменной |
| pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) | Блокирует поток до получения сигнала, атомарно отпускает мьютекс |
| pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime) | То же, но с таймаутом |
| pthread_cond_signal(pthread_cond_t *cond) | Разблокирует один ожидающий поток (если есть) |
| pthread_cond_broadcast(pthread_cond_t *cond) | Разблокирует все ожидающие потоки |

4. **Принцип работы условных переменных**

Условная переменная всегда используется вместе с мьютексом. Типичный паттерн:

```
c
// Ожидание условия
pthread_mutex_lock(&mutex);
while (условие_не_выполнено) {
   pthread_cond_wait(&cond, &mutex);
}
// выполнение действия (условие выполнено)
pthread_mutex_unlock(&mutex);

// Уведомление
pthread_mutex_lock(&mutex);
// изменение условия
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

5. **Ложные пробуждения (spurious wakeups)**

Функция pthread_cond_wait может возвращаться даже без вызова pthread_cond_signal (ложное пробуждение). Поэтому необходимо проверять условие в цикле while, а не в if.

6. **Отличие signal от broadcast**
   - signal – пробуждает **один** ожидающий поток (обычно для производитель-потребитель).
   - broadcast – пробуждает **все** ожидающие потоки (когда несколько потоков ожидают разных условий).

7. **Кольцевой буфер (ring buffer)**

Для реализации ограниченного буфера удобно использовать массив фиксированного размера с индексами записи и чтения, зацикленными по модулю размера буфера.

8. **Структура синхронизации для producer-consumer**
    - Один мьютекс для защиты доступа к буферу.
    - Две условные переменные: cond_not_full (буфер не заполнен) и cond_not_empty (буфер не пуст).
    - Счётчики: count (текущее количество элементов) или индексы in и out.

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab13

   cd ~/linux_labs/lab13

2. **Реализация простого producer-consumer с одним производителем и одним потребителем**

   2.1. Создайте файл producer_consumer.c:

   c

   #include <stdio.h>

   #include <pthread.h>

   #include <stdlib.h>

   #include <unistd.h>

   #define BUFFER_SIZE 10

```
#define NUM_ITEMS 30

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        pthread_mutex_lock(&mutex);

        // ждём, пока буфер не заполнен
        while (count == BUFFER_SIZE) {
            printf("Producer: buffer full, waiting...\n");
            pthread_cond_wait(&cond_not_full, &mutex);
        }

        // производим элемент
        buffer[in] = i;
        printf("Producer: produced item %d at index %d\n", i, in);
        in = (in + 1) % BUFFER_SIZE;
        count++;

        // уведомляем потребителя
        pthread_cond_signal(&cond_not_empty);
        pthread_mutex_unlock(&mutex);
```

```
        sleep(rand() % 2);  // имитация работы
    }
    return NULL;
}

void *consumer(void *arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        pthread_mutex_lock(&mutex);

        // ждём, пока буфер не пуст
        while (count == 0) {
            printf("Consumer: buffer empty, waiting...\n");
            pthread_cond_wait(&cond_not_empty, &mutex);
        }

        // потребляем элемент
        int item = buffer[out];
        printf("Consumer: consumed item %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;
        count--;

        // уведомляем производителя
        pthread_cond_signal(&cond_not_full);
        pthread_mutex_unlock(&mutex);

        sleep(rand() % 2);  // имитация работы
    }
    return NULL;
}
```

```
int main() {
    srand(time(NULL));
    pthread_t prod, cons;

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_not_full);
    pthread_cond_destroy(&cond_not_empty);

    return 0;
}
```

2.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g producer_consumer.c -o producer_consumer -pthread
./producer_consumer
```

3. **Реализация с несколькими производителями и потребителями**

3.1. Создайте файл multi_prod_cons.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define BUFFER_SIZE 8
#define NUM_ITEMS 20
#define NUM_PRODUCERS 3
#define NUM_CONSUMERS 3

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;
int produced_total = 0;
int consumed_total = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
    int id = *(int*)arg;
    while (1) {
        pthread_mutex_lock(&mutex);

        if (produced_total >= NUM_ITEMS) {
            pthread_mutex_unlock(&mutex);
            break;
        }

        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&cond_not_full, &mutex);
            if (produced_total >= NUM_ITEMS) {
```

```
            pthread_mutex_unlock(&mutex);
            return NULL;
        }
    }

    int item = produced_total++;
    buffer[in] = item;
    printf("Producer %d: produced item %d at index %d (count=%d)\n",
           id, item, in, count + 1);
    in = (in + 1) % BUFFER_SIZE;
    count++;

    pthread_cond_signal(&cond_not_empty);
    pthread_mutex_unlock(&mutex);
    usleep(rand() % 500000);
  }
  printf("Producer %d: finishing\n", id);
  return NULL;
}

void *consumer(void *arg) {
  int id = *(int*)arg;
  while (1) {
    pthread_mutex_lock(&mutex);

    if (consumed_total >= NUM_ITEMS) {
      pthread_mutex_unlock(&mutex);
      break;
    }
```

```
            while (count == 0) {
                pthread_cond_wait(&cond_not_empty, &mutex);
                if (consumed_total >= NUM_ITEMS) {
                    pthread_mutex_unlock(&mutex);
                    return NULL;
                }
            }

            int item = buffer[out];
            printf("Consumer %d: consumed item %d from index %d (count=%d)\n",
                   id, item, out, count - 1);
            out = (out + 1) % BUFFER_SIZE;
            count--;
            consumed_total++;

            pthread_cond_signal(&cond_not_full);
            pthread_mutex_unlock(&mutex);
            usleep(rand() % 500000);
        }
        printf("Consumer %d: finishing\n", id);
        return NULL;
    }

    int main() {
        srand(time(NULL));
        pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];
        int prod_ids[NUM_PRODUCERS], cons_ids[NUM_CONSUMERS];
```

```
    for (int i = 0; i < NUM_PRODUCERS; i++) {
        prod_ids[i] = i;
        pthread_create(&producers[i], NULL, producer, &prod_ids[i]);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        cons_ids[i] = i;
        pthread_create(&consumers[i], NULL, consumer, &cons_ids[i]);
    }

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
    }

    printf("All done! Produced: %d, Consumed: %d\n", produced_total, consumed_total);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_not_full);
    pthread_cond_destroy(&cond_not_empty);

    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

gcc -Wall -g multi_prod_cons.c -o multi_prod_cons -pthread

./multi_prod_cons

,

4. **Использование pthread_cond_broadcast**

4.1. Модифицируйте программу, заменив некоторые signal на broadcast для пробуждения всех ожидающих потоков.

5. **Изучение ложных пробуждений**

5.1. Попробуйте заменить while (count == BUFFER_SIZE) на if (count == BUFFER_SIZE) и объясните, почему это может привести к проблемам.

## Пример выполнения задания

**Задача примера:** реализовать программу для моделирования работы склада с ограниченной вместимостью. Поставщики (производители) доставляют товары на склад, покупатели (потребители) забирают товары со склада. Склад не может быть заполнен сверх вместимости, и покупатели не могут забрать товар с пустого склада. Каждый поставщик и покупатель работает в своём потоке.

**Шаг 1. Реализация программы**

Файл warehouse.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define WAREHOUSE_SIZE 5
#define NUM_SUPPLIERS 2
#define NUM_CUSTOMERS 3
```

```
#define TOTAL_ITEMS 15

typedef struct {
    int items[WAREHOUSE_SIZE];
    int in;
    int out;
    int count;
    int total_produced;
    int total_consumed;
    pthread_mutex_t mutex;
    pthread_cond_t cond_not_full;
    pthread_cond_t cond_not_empty;
} Warehouse;

Warehouse warehouse = {
    .in = 0,
    .out = 0,
    .count = 0,
    .total_produced = 0,
    .total_consumed = 0,
    .mutex = PTHREAD_MUTEX_INITIALIZER,
    .cond_not_full = PTHREAD_COND_INITIALIZER,
    .cond_not_empty = PTHREAD_COND_INITIALIZER
};

void *supplier(void *arg) {
    int id = *(int*)arg;

    while (1) {
        pthread_mutex_lock(&warehouse.mutex);
```

```
if (warehouse.total_produced >= TOTAL_ITEMS) {
    pthread_mutex_unlock(&warehouse.mutex);
    break;
}

// ждём, пока склад не заполнен
while (warehouse.count == WAREHOUSE_SIZE) {
    printf("Supplier %d: warehouse full, waiting...\n", id);
    pthread_cond_wait(&warehouse.cond_not_full, &warehouse.mutex);
    if (warehouse.total_produced >= TOTAL_ITEMS) {
        pthread_mutex_unlock(&warehouse.mutex);
        return NULL;
    }
}

int item = warehouse.total_produced + 1;
warehouse.items[warehouse.in] = item;
warehouse.total_produced++;
warehouse.count++;

printf("Supplier %d: delivered item %d, warehouse count = %d\n",
       id, item, warehouse.count);

warehouse.in = (warehouse.in + 1) % WAREHOUSE_SIZE;

pthread_cond_signal(&warehouse.cond_not_empty);
pthread_mutex_unlock(&warehouse.mutex);

sleep(rand() % 2);  // время на доставку
```

```
        }

        printf("Supplier %d: finished\n", id);
        return NULL;
    }

    void *customer(void *arg) {
        int id = *(int*)arg;

        while (1) {
            pthread_mutex_lock(&warehouse.mutex);

            if (warehouse.total_consumed >= TOTAL_ITEMS) {
                pthread_mutex_unlock(&warehouse.mutex);
                break;
            }

            // ждём, пока склад не пуст
            while (warehouse.count == 0) {
                printf("Customer %d: warehouse empty, waiting...\n", id);
                pthread_cond_wait(&warehouse.cond_not_empty,
&warehouse.mutex);
                if (warehouse.total_consumed >= TOTAL_ITEMS) {
                    pthread_mutex_unlock(&warehouse.mutex);
                    return NULL;
                }
            }

            int item = warehouse.items[warehouse.out];
            warehouse.total_consumed++;
```

```
            warehouse.count--;

            printf("Customer %d: took item %d, warehouse count = %d\n",
                id, item, warehouse.count);

            warehouse.out = (warehouse.out + 1) % WAREHOUSE_SIZE;

            pthread_cond_signal(&warehouse.cond_not_full);
            pthread_mutex_unlock(&warehouse.mutex);

            sleep(rand() % 3);  // время на обработку покупки
        }

        printf("Customer %d: finished\n", id);
        return NULL;
    }

    int main() {
        srand(time(NULL));
        pthread_t suppliers[NUM_SUPPLIERS], customers[NUM_CUSTOMERS];
        int supplier_ids[NUM_SUPPLIERS], customer_ids[NUM_CUSTOMERS];

        printf("=== Warehouse Simulation ===\n");
        printf("Warehouse capacity: %d\n", WAREHOUSE_SIZE);
        printf("Total items to process: %d\n\n", TOTAL_ITEMS);

        for (int i = 0; i < NUM_SUPPLIERS; i++) {
            supplier_ids[i] = i + 1;
```

```
        pthread_create(&suppliers[i], NULL, supplier, &supplier_ids[i]);
    }

    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        customer_ids[i] = i + 1;
        pthread_create(&customers[i], NULL, customer, &customer_ids[i]);
    }

    for (int i = 0; i < NUM_SUPPLIERS; i++) {
        pthread_join(suppliers[i], NULL);
    }

    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        pthread_join(customers[i], NULL);
    }

    printf("\n=== Simulation finished ===\n");
    printf("Total delivered: %d\n", warehouse.total_produced);
    printf("Total taken: %d\n", warehouse.total_consumed);

    pthread_mutex_destroy(&warehouse.mutex);
    pthread_cond_destroy(&warehouse.cond_not_full);
    pthread_cond_destroy(&warehouse.cond_not_empty);

    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

```
gcc -Wall -g warehouse.c -o warehouse -pthread
```

./warehouse

**Шаг 3. Пример вывода программы**

=== Warehouse Simulation ===

Warehouse capacity: 5

Total items to process: 15

Supplier 1: delivered item 1, warehouse count = 1

Supplier 2: delivered item 2, warehouse count = 2

Customer 1: took item 1, warehouse count = 1

Supplier 1: delivered item 3, warehouse count = 2

Customer 2: took item 2, warehouse count = 1

Supplier 2: delivered item 4, warehouse count = 2

Customer 3: took item 3, warehouse count = 1

Supplier 1: delivered item 5, warehouse count = 2

Supplier 2: delivered item 6, warehouse count = 3

Customer 1: took item 4, warehouse count = 2

...

Supplier 1: finished

Supplier 2: finished

Customer 1: finished

Customer 2: finished

Customer 3: finished

=== Simulation finished ===

Total delivered: 15

Total taken: 15

**Вывод:**

Программа демонстрирует:

- Корректную синхронизацию нескольких производителей и потребителей.
- Использование условных переменных для ожидания заполнения/опустошения буфера.
- Защиту от переполнения буфера и чтения из пустого буфера.
- Грациозное завершение всех потоков после обработки всех элементов.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы. Для вариантов >6 – цикл повторяется.

| Вариант | Задание |
|---|---|
| 1 | Реализовать задачу «производитель-потребитель» для обработки строк. Производитель генерирует строки (например, "Task X"), потребитель выводит их на экран. Буфер – массив строк фиксированного размера. |
| 2 | Создать программу для моделирования работы кассы в магазине. Покупатели (потребители) подходят к кассе, кассир (один производитель) обслуживает их. Ограниченная очередь ожидания (буфер). |
| 3 | Написать программу для параллельной обработки файлов. Производители читают строки из файлов и помещают в буфер. Потребители обрабатывают строки (например, подсчитывают длину). |
| 4 | Реализовать программу для моделирования работы принтера. Несколько потоков-пользователей отправляют задания на печать, один поток-принтер печатает их. Ограниченная очередь заданий. |
| 5 | Создать программу для параллельного вычисления чисел Фибоначчи. Производитель генерирует запросы (индексы), потребители вычисляют значения. Использовать буфер для хранения запросов. |
| 6 | Реализовать программу «телефонный справочник»: производители добавляют записи в буфер, потребители сохраняют их в файл. Буфер ограниченного размера. |

**Обратить внимание:**

– Всегда использовать цикл while для проверки условия после pthread_cond_wait.

– Не забывать проверять условие завершения перед ожиданием.

– Освобождать мьютекс перед выходом из потока.

– Уничтожать условные переменные и мьютексы после использования.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):

– реализация producer-consumer с одним производителем и потребителем;

– реализация с несколькими производителями и потребителями;

– объяснение использования while вместо if;

– демонстрация работы условных переменных.

4. Скриншоты (обязательно):

– вывод producer_consumer;

– вывод multi_prod_cons;

– (по варианту) вывод программы по индивидуальному заданию.

5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое условная переменная? Для чего она используется?
2. Почему условная переменная всегда используется вместе с мьютексом?
3. В чём разница между pthread_cond_signal и pthread_cond_broadcast?
4. Что такое ложное пробуждение (spurious wakeup)? Как с ним бороться?
5. Почему условие в pthread_cond_wait проверяется в цикле while, а не в if?
6. Что происходит с мьютексом внутри pthread_cond_wait?
7. Как решается проблема «производитель-потребитель» с ограниченным буфером?
8. Какие две условные переменные обычно используются в задаче producer-consumer?
9. Можно ли использовать одну условную переменную для ожидания и заполнения, и опустошения буфера? Почему?
10. Что произойдёт, если вызвать pthread_cond_signal до того, как какой-либо поток вошёл в pthread_cond_wait?
11. Как корректно завершить потоки в программе с producer-consumer?
12. Что такое кольцевой буфер? Какие преимущества он даёт?

# Лабораторная работа №14
# Семафоры POSIX

**Цель работы:** освоить использование семафоров POSIX для синхронизации потоков и процессов, реализовать задачу «читатели-писатели» с использованием семафоров, сравнить различные подходы к синхронизации, изучить именованные и неименованные семафоры.

**Задачи:**

– Изучить семафоры POSIX как средство синхронизации.

– Освоить функции работы с неименованными семафорами: sem_init, sem_wait, sem_post, sem_destroy.

– Изучить именованные семафоры: sem_open, sem_close, sem_unlink.

– Реализовать задачу «читатели-писатели» с использованием семафоров.

– Сравнить решение на семафорах с решением на мьютексах и условных переменных.

– Реализовать синхронизацию между процессами с помощью именованных семафоров.

## Теоретические сведения

1. **Семафоры POSIX**

Семафор – счётный механизм синхронизации, позволяющий ограничивать количество потоков (процессов), одновременно имеющих доступ к ресурсу. Семафор может принимать неотрицательные целые значения. Основные операции:

- sem_wait (P-операция) – уменьшает значение семафора; если значение равно 0, поток блокируется.
- sem_post (V-операция) – увеличивает значение семафора; если есть заблокированные потоки, один из них разблокируется.

2. **Неименованные vs именованные семафоры**

| **Характеристика** | **Неименованные** | **Именованные** |
|---|---|---|
| Применение | Потоки одного процесса | Потоки + разные процессы |
| Идентификация | Адрес переменной | Имя (строка, начинается с /) |
| Создание | sem_init | sem_open |
| Удаление | sem_destroy | sem_unlink |
| Хранение | В памяти процесса | В файловой системе (/dev/shm/) |

3. **Основные функции для неименованных семафоров**

| **Функция** | **Назначение** |
|---|---|
| sem_init(sem_t *sem, int pshared, unsigned int value) | Инициализация семафора. pshared = 0 – для потоков, pshared != 0 – для процессов |
| sem_wait(sem_t *sem) | Уменьшение семафора (блокирующая) |
| sem_trywait(sem_t *sem) | Уменьшение семафора (неблокирующая) |
| sem_timedwait(sem_t *sem, const struct timespec *abs_timeout) | Уменьшение с таймаутом |
| sem_post(sem_t *sem) | Увеличение семафора |
| sem_destroy(sem_t *sem) | Уничтожение семафора |

4. **Основные функции для именованных семафоров**

| Функция | Назначение |
|---|---|
| sem_open(const char *name, int oflag, mode_t mode, unsigned int value) | Открывает/создаёт именованный семафор |
| sem_close(sem_t *sem) | Закрывает семафор |
| sem_unlink(const char *name) | Удаляет семафор из системы |
| Остальные (sem_wait, sem_post и др.) | Аналогичны неименованным |

5. **Задача «читатели-писатели»**

Классическая задача синхронизации, где:

- **Читатели** только читают данные, могут делать это одновременно.
- **Писатели** изменяют данные, требуют исключительного доступа.

**Условия:**

- Несколько читателей могут читать одновременно.
- Писатель должен иметь эксклюзивный доступ.
- Пока писатель работает, читатели не должны читать.
- Пока читатели читают, писатель не должен писать.

6. **Реализация задачи «читатели-писатели» с семафорами**

```
c

sem_t mutex;      // защита счётчика читателей
sem_t write_lock; // доступ для писателей
int readers = 0;


// Читатель
sem_wait(&mutex);
readers++;
if (readers == 1) sem_wait(&write_lock);
sem_post(&mutex);
```

```
// Чтение

sem_wait(&mutex);
readers--;
if (readers == 0) sem_post(&write_lock);
sem_post(&mutex);

// Писатель
sem_wait(&write_lock);
// Запись
sem_post(&write_lock);
```

7. **Семафор как бинарный мьютекс**

Семафор с начальным значением 1 может использоваться как мьютекс:

```
c
sem_wait(&mutex);
// критическая секция
sem_post(&mutex);
```

8. **Сравнение мьютексов и семафоров**
    - Мьютекс может быть разблокирован только тем потоком, который его заблокировал.
    - Семафор может быть увеличен любым потоком (удобно для producer-consumer).
    - Мьютексы легче (меньше накладных расходов).
    - Семафоры поддерживают счёт (более сложные паттерны синхронизации).

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab14

   cd ~/linux_labs/lab14

2. **Использование семафора как мьютекса**

   2.1. Создайте файл sem_mutex.c:

   c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>

#define NUM_THREADS 10
#define ITERATIONS 1000000

long counter = 0;
sem_t mutex;

void *increment(void *arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        sem_wait(&mutex);
        counter++;
        sem_post(&mutex);
    }
    return NULL;
}
```

```
int main() {
    sem_init(&mutex, 0, 1);   // pshared=0 (только потоки), начальное значение=1
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, increment, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Expected: %ld\n", (long)NUM_THREADS * ITERATIONS);
    printf("Actual: %ld\n", counter);

    sem_destroy(&mutex);
    return 0;
}
```

2.2. Скомпилируйте и выполните (требуется линковка с -pthread и -lrt для некоторых систем):

```
bash
gcc -Wall -g sem_mutex.c -o sem_mutex -pthread
./sem_mutex
```

3. **Реализация producer-consumer с семафорами**

3.1. Создайте файл sem_prod_cons.c:

```
c
#include <stdio.h>
```

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE 10
#define NUM_ITEMS 30

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t empty;   // счётчик свободных мест
sem_t full;    // счётчик заполненных мест
sem_t mutex;   // защита индексов

void *producer(void *arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&empty);      // ждём свободное место
        sem_wait(&mutex);

        buffer[in] = i;
        printf("Producer: produced %d at index %d\n", i, in);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);       // увеличиваем счётчик заполненных

        sleep(rand() % 2);
    }
```

```
    return NULL;
}

void *consumer(void *arg) {
    for (int i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full);       // ждём данные
        sem_wait(&mutex);

        int item = buffer[out];
        printf("Consumer: consumed %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);      // увеличиваем счётчик свободных

        sleep(rand() % 2);
    }
    return NULL;
}

int main() {
    srand(time(NULL));

    sem_init(&empty, 0, BUFFER_SIZE); // BUFFER_SIZE свободных мест
    sem_init(&full, 0, 0);            // 0 заполненных мест
    sem_init(&mutex, 0, 1);           // мьютекс

    pthread_t prod, cons;
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
```

```
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g sem_prod_cons.c -o sem_prod_cons -pthread
./sem_prod_cons
```

4. **Реализация задачи «читатели-писатели» с семафорами**

4.1. Создайте файл readers_writers.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_READERS 5
#define NUM_WRITERS 2
#define NUM_OPERATIONS 10

int shared_data = 0;
```

```
int readers_count = 0;

sem_t mutex;        // защита readers_count
sem_t write_lock;   // эксклюзивный доступ для писателей

void *reader(void *arg) {
    int id = *(int*)arg;
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        sem_wait(&mutex);
        readers_count++;
        if (readers_count == 1) {
            sem_wait(&write_lock);  // первый читатель блокирует писателей
        }
        sem_post(&mutex);

        // чтение данных
        printf("Reader %d: read value = %d\n", id, shared_data);
        usleep(rand() % 500000);

        sem_wait(&mutex);
        readers_count--;
        if (readers_count == 0) {
            sem_post(&write_lock);      // последний читатель разрешает писателям
        }
        sem_post(&mutex);

        usleep(rand() % 300000);
    }
    return NULL;
```

```
}

void *writer(void *arg) {
    int id = *(int*)arg;
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        sem_wait(&write_lock);  // эксклюзивный доступ

        // запись данных
        shared_data++;
        printf("Writer %d: wrote value = %d\n", id, shared_data);
        usleep(rand() % 500000);

        sem_post(&write_lock);

        usleep(rand() % 300000);
    }
    return NULL;
}

int main() {
    srand(time(NULL));

    sem_init(&mutex, 0, 1);
    sem_init(&write_lock, 0, 1);

    pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
    int reader_ids[NUM_READERS], writer_ids[NUM_WRITERS];

    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i + 1;
```

```
        pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
    }

    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&write_lock);

    return 0;
}
```

4.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g readers_writers.c -o readers_writers -pthread
./readers_writers
```

5. **Именованные семафоры для синхронизации процессов**

5.1. Создайте файл named_sem_proc1.c (процесс-производитель):

c

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

#define SEM_NAME "/my_semaphore"

int main() {
    sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0644, 0);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        return 1;
    }

    printf("Process 1: sending signal...\n");
    sleep(2);
    sem_post(sem);
    printf("Process 1: signal sent\n");

    sem_close(sem);
    return 0;
}
```

5.2. Создайте файл named_sem_proc2.c (процесс-потребитель):

c

```
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>
```

```
#define SEM_NAME "/my_semaphore"

int main() {
    sem_t *sem = sem_open(SEM_NAME, 0);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        return 1;
    }

    printf("Process 2: waiting for signal...\n");
    sem_wait(sem);
    printf("Process 2: signal received!\n");

    sem_close(sem);
    sem_unlink(SEM_NAME);
    return 0;
}
```

5.3. Скомпилируйте и запустите (в двух разных терминалах):

```
bash
gcc -Wall -g named_sem_proc2.c -o proc2 -pthread
gcc -Wall -g named_sem_proc1.c -o proc1 -pthread

# Терминал 1:
./proc2

# Терминал 2:
./proc1
```

## Пример выполнения задания

**Задача примера:** реализовать программу для моделирования работы парковки с ограниченной вместимостью. Автомобили (потоки) прибывают на парковку и покидают её. Когда парковка заполнена, прибывающие автомобили ждут освобождения места. Решить задачу с использованием семафоров.

**Шаг 1. Реализация программы**

Файл parking.c:

c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define PARKING_SIZE 5
#define NUM_CARS 15

sem_t parking_spaces;  // счётчик свободных мест
sem_t mutex;           // защита вывода на экран

void *car(void *arg) {
    int id = *(int*)arg;

    // случайное время прибытия
    usleep(rand() % 1000000);

    // ожидание свободного места на парковке
    sem_wait(&parking_spaces);
```

```
    sem_wait(&mutex);
    int value;
    sem_getvalue(&parking_spaces, &value);
    printf("Car %d: PARKED. Free spaces: %d\n", id, value);
    sem_post(&mutex);

    // автомобиль на парковке
    int parking_time = rand() % 5 + 1;
    sleep(parking_time);

    sem_wait(&mutex);
    sem_getvalue(&parking_spaces, &value);
    printf("Car %d: LEAVING after %d sec. Free spaces: %d\n",
           id, parking_time, value + 1);
    sem_post(&mutex);

    // освобождение места
    sem_post(&parking_spaces);

    return NULL;
}

int main() {
    srand(time(NULL));

    sem_init(&parking_spaces, 0, PARKING_SIZE);
    sem_init(&mutex, 0, 1);

    pthread_t cars[NUM_CARS];
    int car_ids[NUM_CARS];
```

```
    printf("=== Parking Simulation ===\n");
    printf("Parking capacity: %d spaces\n", PARKING_SIZE);
    printf("Number of cars: %d\n\n", NUM_CARS);

    for (int i = 0; i < NUM_CARS; i++) {
        car_ids[i] = i + 1;
        pthread_create(&cars[i], NULL, car, &car_ids[i]);
    }

    for (int i = 0; i < NUM_CARS; i++) {
        pthread_join(cars[i], NULL);
    }

    printf("\n=== Simulation finished ===\n");

    sem_destroy(&parking_spaces);
    sem_destroy(&mutex);

    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

```
gcc -Wall -g parking.c -o parking -pthread
./parking
```

**Шаг 3. Пример вывода программы**

```
=== Parking Simulation ===
Parking capacity: 5 spaces
```

Number of cars: 15

Car 1: PARKED. Free spaces: 4

Car 3: PARKED. Free spaces: 3

Car 2: PARKED. Free spaces: 2

Car 5: PARKED. Free spaces: 1

Car 4: PARKED. Free spaces: 0

Car 6: waiting for parking...

Car 7: waiting for parking...

Car 1: LEAVING after 3 sec. Free spaces: 1

Car 6: PARKED. Free spaces: 0

Car 8: waiting for parking...

Car 2: LEAVING after 4 sec. Free spaces: 1

Car 7: PARKED. Free spaces: 0

...

**Вывод:**

Программа демонстрирует:

- Использование семафора как счётчика для ограничения доступа к ресурсу.
- Корректную синхронизацию при заполнении парковки.
- Автоматическую блокировку автомобилей при отсутствии свободных мест.
- Освобождение места и пробуждение ожидающих автомобилей.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать задачу «читатели-писатели» с приоритетом писателей (писатели не должны ждать, если есть читатели). Использовать семафоры. |
| 2 | Создать программу для моделирования работы лифта. Пассажиры вызывают лифт с разных этажей, лифт обрабатывает запросы. Использовать семафоры для ограничения вместимости. |
| 3 | Реализовать задачу «обедающие философы» с использованием семафоров (5 философов, 5 вилок). Избежать deadlock. |
| 4 | Написать программу для синхронизации двух процессов с использованием именованного семафора. Один процесс генерирует числа, второй выводит их квадраты. |
| 5 | Реализовать пул потоков (thread pool) с использованием семафоров. Потоки ожидают задания в семафоре, главный поток добавляет задания в очередь. |
| 6 | Создать программу для моделирования работы барьера (barrier) с использованием семафоров. N потоков должны достичь барьера одновременно, прежде чем продолжить выполнение. |

**Обратить внимание:**

– Для именованных семафоров имена должны начинаться с символа /.

– Именованные семафоры не удаляются автоматически при завершении процесса (использовать sem_unlink).

– При использовании семафоров в разных процессах необходимо установить pshared = 1.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – использование семафора как мьютекса;
   – реализация producer-consumer с семафорами;
   – реализация задачи «читатели-писатели»;
   – работа с именованными семафорами.
4. Скриншоты (обязательно):
   – вывод sem_mutex (правильный результат);
   – вывод sem_prod_cons;
   – вывод readers_writers;
   – работа именованных семафоров (два терминала).
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое семафор? Чем семафор отличается от мьютекса?
2. Какие операции определены над семафором? Что означают P- и V-операции?
3. Как создать неименованный семафор для синхронизации потоков? Для синхронизации процессов?
4. Как создать именованный семафор? В чём преимущество именованных семафоров?
5. Для чего нужен параметр pshared в sem_init?
6. Как реализовать мьютекс на основе семафора?
7. В чём отличие sem_wait от sem_trywait?
8. Как решается задача «читатели-писатели» с помощью семафоров?
9. Почему в реализации «читатели-писатели» используется семафор mutex для защиты счётчика читателей?
10. Что произойдёт, если забыть удалить именованный семафор с помощью sem_unlink?
11. Можно ли использовать семафор для синхронизации более двух потоков? Приведите пример.
12. Как получить текущее значение семафора?

# Лабораторная работа №15
# Неименованные каналы (pipe)

**Цель работы:** освоить межпроцессное взаимодействие с использованием неименованных каналов (pipe), научиться передавать данные между родственными процессами, реализовать цепочку процессов с перенаправлением стандартных потоков ввода-вывода, эмулируя работу конвейера командной оболочки.

**Задачи:**

– Изучить системный вызов pipe() для создания неименованного канала.

– Научиться передавать данные от родительского процесса к дочернему через канал.

– Освоить перенаправление стандартных потоков (stdin, stdout) с помощью dup2().

– Реализовать цепочку из нескольких процессов, соединённых каналами (аналог конвейера | в shell).

– Изучить особенности работы с каналами (блокирующее чтение/запись, закрытие ненужных дескрипторов).

– Реализовать эмуляцию команды ls | grep | wc с использованием нескольких каналов.

## Теоретические сведения

1. **Неименованные каналы (pipe)**

   Канал (pipe) – простейшее средство IPC, позволяющее передавать поток данных от одного процесса к другому. Канал работает по принципу FIFO (first in, first out). Неименованные каналы существуют только во время выполнения процессов и не имеют имени в файловой системе.

2. **Системный вызов pipe()**

   с

int pipe(int pipefd[2]);

- Создаёт канал и возвращает два файловых дескриптора:
  - pipefd[0] – конец для чтения (read end)
  - pipefd[1] – конец для записи (write end)
- Данные, записанные в pipefd[1], могут быть прочитаны из pipefd[0].
- Возвращает 0 при успехе, -1 при ошибке.

3. **Направление передачи данных**

Процесс A (запись) → pipefd[1] → [канал] → pipefd[0] → Процесс B (чтение)

4. **Использование pipe для связи родитель-потомок**

Типичный паттерн:

- Родитель создаёт канал с помощью pipe().
- Вызывает fork() для создания дочернего процесса.
- Родитель закрывает ненужный конец канала (например, для записи, если он только читает).
- Потомок закрывает свой ненужный конец.
- Процессы обмениваются данными через оставшиеся дескрипторы.

5. **Перенаправление потоков с dup2()**

c

int dup2(int oldfd, int newfd);

- Делает newfd копией oldfd. После этого newfd указывает на тот же файл/канал.
- Часто используется для перенаправления stdin (0) или stdout (1) на канал.

6. **Конвейер (pipe chain) из нескольких процессов**

Для создания цепочки cmd1 | cmd2 | cmd3:

- Создаётся первый канал между cmd1 и cmd2.
- Создаётся второй канал между cmd2 и cmd3.

- cmd1 пишет в свой канал, cmd2 читает из него и пишет в следующий, cmd3 читает.

7. **Важные особенности работы с каналами**
   - Чтение из канала, у которого закрыт конец записи, возвращает 0 (конец файла).
   - Запись в канал, у которого закрыт конец чтения, вызывает сигнал SIGPIPE.
   - Операции чтения/записи блокируются, если нет данных/места (по умолчанию).
   - Размер буфера канала ограничен (обычно 4–64 КБ, можно изменить через fcntl).
8. **Стандартные команды для демонстрации**
   - ls – выводит список файлов (пишет в stdout).
   - grep – фильтрует строки по шаблону (читает из stdin, пишет в stdout).
   - wc – подсчитывает строки, слова, байты (читает из stdin, пишет в stdout).

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab15

   cd ~/linux_labs/lab15

2. **Простейший пример: передача строки от родителя к потомку**

   2.1. Создайте файл simple_pipe.c:

   c

   #include <stdio.h>

   #include <unistd.h>

```
#include <string.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];
    char buffer[256];

    if (pipe(pipefd) == -1) {
        perror("pipe");
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid == 0) {
        // Дочерний процесс: читает из канала
        close(pipefd[1]);  // закрываем запись
        read(pipefd[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
        close(pipefd[0]);
    } else {
        // Родительский процесс: пишет в канал
        close(pipefd[0]);  // закрываем чтение
        char *message = "Hello from parent!";
        write(pipefd[1], message, strlen(message) + 1);
        close(pipefd[1]);
```

```
        wait(NULL);
    }
    return 0;
}
```

2.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g simple_pipe.c -o simple_pipe
./simple_pipe
```

3. **Двусторонняя связь: два канала для обмена данными**

3.1. Создайте файл bidirectional_pipe.c:

c

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

int main() {
    int parent_to_child[2];
    int child_to_parent[2];

    pipe(parent_to_child);
    pipe(child_to_parent);

    pid_t pid = fork();
    if (pid == 0) {
        // Дочерний процесс
        close(parent_to_child[1]);
        close(child_to_parent[0]);
```

```
        char buffer[256];
        read(parent_to_child[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);

        char *response = "Hello from child!";
        write(child_to_parent[1], response, strlen(response) + 1);

        close(parent_to_child[0]);
        close(child_to_parent[1]);
    } else {
        // Родительский процесс
        close(parent_to_child[0]);
        close(child_to_parent[1]);

        char *message = "Hello from parent!";
        write(parent_to_child[1], message, strlen(message) + 1);

        char buffer[256];
        read(child_to_parent[0], buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);

        close(parent_to_child[1]);
        close(child_to_parent[0]);
        wait(NULL);
    }
    return 0;
}
```

3.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g bidirectional_pipe.c -o bidirectional_pipe
./bidirectional_pipe
```

4. **Эмуляция команды** ls | wc -l **(один канал)**

4.1. Создайте файл ls_wc.c:

c

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

int main() {
    int pipefd[2];
    pipe(pipefd);

    pid_t pid1 = fork();
    if (pid1 == 0) {
        // Первый потомок: выполняет ls
        close(pipefd[0]);          // закрываем чтение
        dup2(pipefd[1], STDOUT_FILENO);  // перенаправляем stdout в канал
        close(pipefd[1]);          // закрываем оригинальный дескриптор
        execlp("ls", "ls", "-l", NULL);
        perror("execlp ls");
        return 1;
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {
        // Второй потомок: выполняет wc -l
        close(pipefd[1]);          // закрываем запись
```

```
        dup2(pipefd[0], STDIN_FILENO);   // перенаправляем stdin из канала
        close(pipefd[0]);          // закрываем оригинальный дескриптор
        execlp("wc", "wc", "-l", NULL);
        perror("execlp wc");
        return 1;
    }

    // Родитель: закрывает оба конца канала и ждёт потомков
    close(pipefd[0]);
    close(pipefd[1]);
    wait(NULL);
    wait(NULL);

    return 0;
}
```

4.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g ls_wc.c -o ls_wc
./ls_wc
```

Сравните с результатом ls -l | wc -l в командной строке.

5. **Эмуляция цепочки из трёх процессов:** ls | grep ".c" | wc -l

5.1. Создайте файл ls_grep_wc.c:

c

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

```
int main() {
    int pipe1[2];  // между ls и grep
    int pipe2[2];  // между grep и wc

    pipe(pipe1);
    pipe(pipe2);

    // Процесс 1: ls -l
    if (fork() == 0) {
        close(pipe1[0]);
        dup2(pipe1[1], STDOUT_FILENO);
        close(pipe1[1]);
        close(pipe2[0]);
        close(pipe2[1]);
        execlp("ls", "ls", "-l", NULL);
        perror("execlp ls");
        return 1;
    }

    // Процесс 2: grep ".c"
    if (fork() == 0) {
        close(pipe1[1]);
        dup2(pipe1[0], STDIN_FILENO);
        close(pipe1[0]);

        close(pipe2[0]);
        dup2(pipe2[1], STDOUT_FILENO);
        close(pipe2[1]);

        execlp("grep", "grep", "\\.c", NULL);
```

```
        perror("execlp grep");
        return 1;
    }

    // Процесс 3: wc -l
    if (fork() == 0) {
        close(pipe1[0]);
        close(pipe1[1]);
        close(pipe2[1]);
        dup2(pipe2[0], STDIN_FILENO);
        close(pipe2[0]);

        execlp("wc", "wc", "-l", NULL);
        perror("execlp wc");
        return 1;
    }

    // Родитель закрывает все дескрипторы и ждёт завершения
    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[0]);
    close(pipe2[1]);

    for (int i = 0; i < 3; i++) {
        wait(NULL);
    }

    return 0;
}
```

5.2. Скомпилируйте и выполните:

bash

```
gcc -Wall -g ls_grep_wc.c -o ls_grep_wc
./ls_grep_wc
```

6. **Передача большого объёма данных через канал**

6.1. Создайте файл big_data_pipe.c для демонстрации буферизации:

c

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

#define BUFFER_SIZE 1024

int main() {
    int pipefd[2];
    pipe(pipefd);

    pid_t pid = fork();
    if (pid == 0) {
        close(pipefd[1]);
        char buffer[BUFFER_SIZE];
        ssize_t total = 0;
        ssize_t bytes;
        while ((bytes = read(pipefd[0], buffer, BUFFER_SIZE)) > 0) {
            total += bytes;
        }
        printf("Child received %ld bytes\n", total);
        close(pipefd[0]);
```

```
    } else {
        close(pipefd[0]);
        char buffer[BUFFER_SIZE];
        memset(buffer, 'A', BUFFER_SIZE);
        int total = 0;
        for (int i = 0; i < 1000; i++) {
            write(pipefd[1], buffer, BUFFER_SIZE);
            total += BUFFER_SIZE;
        }
        printf("Parent sent %d bytes\n", total);
        close(pipefd[1]);
        wait(NULL);
    }
    return 0;
}
```

## Пример выполнения задания

**Задача примера:** реализовать программу, эмулирующую конвейер ps aux | grep "root" | sort -r | head -5 с использованием четырёх процессов и трёх каналов. Программа должна выводить первые 5 процессов пользователя root в обратном порядке сортировки.

**Шаг 1. Реализация программы**

Файл ps_grep_sort_head.c:

c

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

int main() {
    int pipe1[2];  // ps aux → grep root
```

```
int pipe2[2];  // grep root → sort -r
int pipe3[2];  // sort -r → head -5

pipe(pipe1);
pipe(pipe2);
pipe(pipe3);

// Процесс 1: ps aux
if (fork() == 0) {
    dup2(pipe1[1], STDOUT_FILENO);
    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[0]);
    close(pipe2[1]);
    close(pipe3[0]);
    close(pipe3[1]);
    execlp("ps", "ps", "aux", NULL);
    perror("execlp ps");
    return 1;
}

// Процесс 2: grep "root"
if (fork() == 0) {
    dup2(pipe1[0], STDIN_FILENO);
    dup2(pipe2[1], STDOUT_FILENO);
    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[0]);
    close(pipe2[1]);
    close(pipe3[0]);
```

```
    close(pipe3[1]);
    execlp("grep", "grep", "root", NULL);
    perror("execlp grep");
    return 1;
}

// Процесс 3: sort -r
if (fork() == 0) {
    dup2(pipe2[0], STDIN_FILENO);
    dup2(pipe3[1], STDOUT_FILENO);
    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[0]);
    close(pipe2[1]);
    close(pipe3[0]);
    close(pipe3[1]);
    execlp("sort", "sort", "-r", NULL);
    perror("execlp sort");
    return 1;
}

// Процесс 4: head -5
if (fork() == 0) {
    dup2(pipe3[0], STDIN_FILENO);
    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[0]);
    close(pipe2[1]);
    close(pipe3[0]);
    close(pipe3[1]);
```

```
        execlp("head", "head", "-5", NULL);
        perror("execlp head");
        return 1;
    }

    // Родитель закрывает все дескрипторы
    close(pipe1[0]);
    close(pipe1[1]);
    close(pipe2[0]);
    close(pipe2[1]);
    close(pipe3[0]);
    close(pipe3[1]);

    // Ожидание завершения всех четырёх потомков
    for (int i = 0; i < 4; i++) {
        wait(NULL);
    }

    return 0;
}
```

**Шаг 2. Компиляция и запуск**

```
bash
gcc -Wall -g ps_grep_sort_head.c -o ps_grep_sort_head
./ps_grep_sort_head
```

**Шаг 3. Пример вывода программы**

```
root         1  0.0  0.1  16852  5632 ?        Ss  10:00   0:01 /sbin/init
root       123  0.0  0.0   5184  2048 ?        S   10:01   0:00 /sbin/agetty
```

```
root       456  0.0  0.1  23456  8192 ?        Sl  10:02   0:02 /usr/bin/ssh-agent
root       789  0.0  0.0   4096  1024 ?        S   10:03   0:00 /bin/sh /usr/bin/startx
root       999  0.0  0.2  45678 16384 ?        S   10:04   0:03 /usr/bin/Xorg
```

**Вывод:**

Программа демонстрирует:

- Создание цепочки из четырёх процессов, соединённых тремя каналами.
- Корректное перенаправление stdin и stdout с помощью dup2.
- Закрытие всех неиспользуемых дескрипторов в каждом процессе.
- Ожидание родителем завершения всех дочерних процессов.
- Эмуляцию сложного конвейера командной оболочки.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать конвейер ls -la \| sort -k5 -n \| tail -3 (вывести 3 самых больших файла). Использовать три процесса и два канала. |
| 2 | Создать программу, в которой родительский процесс отправляет дочернему массив чисел через канал. Дочерний процесс вычисляет сумму и возвращает результат через второй канал. |
| 3 | Реализовать конвейер cat /etc/passwd \| cut -d: -f1 \| sort \| uniq -c \| sort -nr. Использовать 5 процессов и 4 канала. |
| 4 | Написать программу, эмулирующую работу команды find . -name "*.c" \| xargs grep -l "main". Использовать два канала и три процесса. |
| 5 | Создать программу для параллельного вычисления: родитель создаёт 2 дочерних процесса, каждый вычисляет свою часть (например, сумму чисел), а третий дочерний процесс собирает результаты через каналы. |
| 6 | Реализовать конвейер dmesg \| grep -i error \| wc -l. Использовать два канала и три процесса. Вывести количество строк с ошибками в ядре. |

**Обратить внимание:**

- Всегда закрывать неиспользуемые концы каналов в каждом процессе.
- После перенаправления dup2 закрывать оригинальные дескрипторы.
- Проверять возвращаемые значения pipe(), fork(), dup2().
- При использовании exec в дочерних процессах не забывать про обработку ошибок.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание простого канала между родителем и потомком;
   – двусторонняя связь через два канала;
   – эмуляция ls | wc -l;
   – эмуляция ls | grep | wc -l;
   – (по варианту) эмуляция более сложного конвейера.
4. Скриншоты (обязательно):
   – вывод simple_pipe;
   – вывод bidirectional_pipe;
   – вывод ls_wc (сравнение с результатом shell);
   – вывод ls_grep_wc;
   – (по варианту) вывод программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое неименованный канал (pipe)? Как он создаётся?
2. Какие два дескриптора возвращает вызов pipe()? Для чего используется каждый?
3. Почему необходимо закрывать неиспользуемые концы канала?
4. Что произойдёт, если читать из канала, у которого закрыт конец записи?
5. Что произойдёт, если писать в канал, у которого закрыт конец чтения?
6. Как перенаправить stdout процесса в канал с помощью dup2()?
7. Как организовать конвейер из трёх процессов (cmd1 | cmd2 | cmd3)?
8. Почему после fork() оба процесса имеют копии дескрипторов канала?
9. Что такое SIGPIPE и когда он возникает?
10. Каков размер буфера канала по умолчанию? Как его можно изменить?
11. Чем отличаются неименованные каналы от именованных (FIFO)?
12. Можно ли использовать один канал для двусторонней связи между двумя процессами? Почему?

# Лабораторная работа №16
# Именованные каналы (FIFO)

**Цель работы:** освоить использование именованных каналов (FIFO) для организации межпроцессного взаимодействия между неродственными процессами, научиться создавать клиент-серверные приложения с обменом данными через FIFO, реализовать простой чат или систему передачи команд.

**Задачи:**

– Изучить именованные каналы (FIFO) как средство IPC для любых процессов.

– Освоить создание FIFO с помощью системного вызова mkfifo() и команды mkfifo.

– Научиться открывать FIFO для чтения и записи (блокирующий и неблокирующий режимы).

– Реализовать серверную программу, читающую данные из FIFO.

– Реализовать клиентскую программу, пишущую данные в FIFO.

– Организовать двусторонний обмен данными с использованием двух FIFO.

– Изучить особенности работы с FIFO (блокировка при открытии, атомарность записи).

## Теоретические сведения

1. **Именованные каналы (FIFO)**

   FIFO (First In, First Out) – поименованный канал, который существует в файловой системе как специальный файл. В отличие от неименованных каналов (pipe), FIFO может использоваться для обмена данными между любыми процессами, не только родственными.

2. **Создание FIFO**
   - Системный вызов: int mkfifo(const char *pathname, mode_t mode);

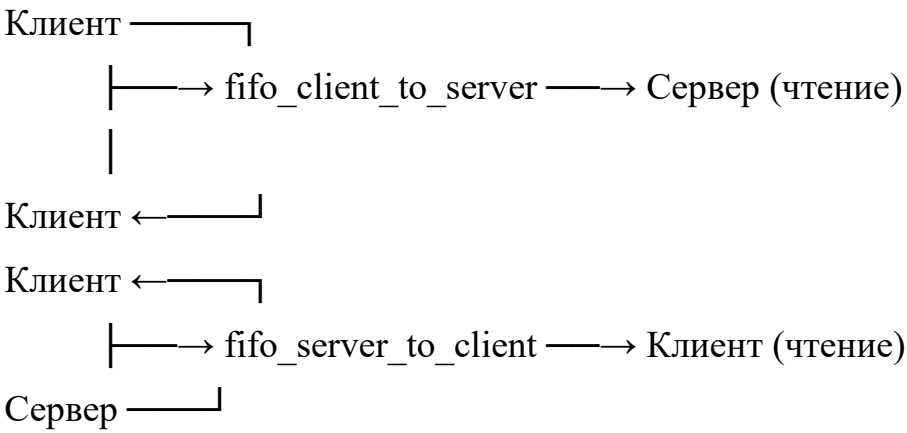
- Команда оболочки: mkfifo <имя>

3. **Свойства FIFO**
   - Данные записываются в FIFO одним процессом и читаются другим (однонаправленный канал).
   - FIFO работает по принципу "первый записан – первый прочитан".
   - Размер буфера FIFO ограничен (обычно 4–64 КБ).
   - Операции чтения/записи блокируются, если нет данных/места.
4. **Блокирующее и неблокирующее открытие FIFO**

   При открытии FIFO с флагом O_RDONLY процесс блокируется до тех пор, пока другой процесс не откроет FIFO для записи.
   При открытии FIFO с флагом O_WRONLY процесс блокируется до тех пор, пока другой процесс не откроет FIFO для чтения.
   Для неблокирующего режима используется флаг O_NONBLOCK.
5. **Атомарность записи**

   Запись в FIFO объёмом до PIPE_BUF байт (обычно 4096 или 5120) является атомарной – данные от разных процессов не перемешиваются.
6. **Организация чата с использованием двух FIFO**

   Для двустороннего обмена данными между двумя процессами используются два FIFO:
   - fifo_server_to_client – для передачи данных от сервера к клиенту.
   - fifo_client_to_server – для передачи данных от клиента к серверу.
7. **Типичная схема клиент-сервер**




8. **Преимущества и недостатки FIFO**

- **Преимущества:** простота, работа между любыми процессами, использование файловой системы.
- **Недостатки:** однонаправленность (нужны два канала для двусторонней связи), блокирующее поведение, ограниченный буфер.

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   ```
   mkdir ~/linux_labs/lab16
   cd ~/linux_labs/lab16
   ```

2. **Создание FIFO и простейшая передача данных**

   2.1. Создайте FIFO с помощью команды:

   bash

   ```
   mkfifo myfifo
   ls -la myfifo  # просмотр типа файла (p)
   ```

   2.2. Создайте файл fifo_writer.c (записывающая программа):

   c

   ```
   #include <stdio.h>
   #include <fcntl.h>
   #include <unistd.h>
   #include <string.h>

   #define FIFO_NAME "/tmp/myfifo"

   int main() {
      int fd = open(FIFO_NAME, O_WRONLY);
      if (fd == -1) {
         perror("open");
   ```

```
        return 1;
    }

    char *message = "Hello from writer!";
    write(fd, message, strlen(message) + 1);
    printf("Writer: message sent\n");

    close(fd);
    return 0;
}
```

2.3. Создайте файл fifo_reader.c (читающая программа):

```
c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define FIFO_NAME "/tmp/myfifo"

int main() {
    int fd = open(FIFO_NAME, O_RDONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    char buffer[256];
    ssize_t bytes = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes > 0) {
        buffer[bytes] = '\0';
        printf("Reader: received: %s\n", buffer);
```

```
    }

    close(fd);
    return 0;
}
```

2.4. Скомпилируйте программы и запустите в разных терминалах:

```
bash
gcc -Wall -g fifo_reader.c -o reader
gcc -Wall -g fifo_writer.c -o writer

# Терминал 1 (запустить сначала):
./reader

# Терминал 2:
./writer
```

3. **Сервер и клиент: передача команд через FIFO**

3.1. Создайте файл server.c:

```
c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define FIFO_NAME "/tmp/command_fifo"

int main() {
    // Создание FIFO (если не существует)
    mkfifo(FIFO_NAME, 0666);
```

```
    printf("Server started. Waiting for commands...\n");

    while (1) {
        int fd = open(FIFO_NAME, O_RDONLY);
        if (fd == -1) {
            perror("open");
            continue;
        }

        char command[256];
        ssize_t bytes = read(fd, command, sizeof(command) - 1);
        if (bytes > 0) {
            command[bytes] = '\0';
            printf("Server received command: %s\n", command);

            // Выполнение команды
            int result = system(command);
            printf("Command executed with status: %d\n", result);
        }

        close(fd);
    }

    return 0;
}
```

3.2. Создайте файл client.c:

```
c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <string.h>

#define FIFO_NAME "/tmp/command_fifo"

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <command>\n", argv[0]);
        return 1;
    }

    // Составляем команду из аргументов
    char command[256] = "";
    for (int i = 1; i < argc; i++) {
        strcat(command, argv[i]);
        if (i < argc - 1) strcat(command, " ");
    }

    int fd = open(FIFO_NAME, O_WRONLY);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    write(fd, command, strlen(command) + 1);
    printf("Client: command sent\n");

    close(fd);
    return 0;
}
```

3.3. Скомпилируйте и запустите:

bash

```
gcc -Wall -g server.c -o server
gcc -Wall -g client.c -o client

# Терминал 1:
./server

# Терминал 2:
./client "ls -la"
./client "date"
./client "whoami"
```

4. **Простой чат с использованием двух FIFO**

4.1. Создайте FIFO для двусторонней связи:

bash

```
mkfifo fifo_user1_to_user2
mkfifo fifo_user2_to_user1
```

4.2. Создайте файл chat_user1.c:

c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>

#define FIFO_SEND "/tmp/fifo_user1_to_user2"
#define FIFO_RECV "/tmp/fifo_user2_to_user1"

void *receive_messages(void *arg) {
    int fd = open(FIFO_RECV, O_RDONLY);
    if (fd == -1) {
```

```
        perror("open receive");
        return NULL;
    }

    char buffer[256];
    while (1) {
        ssize_t bytes = read(fd, buffer, sizeof(buffer) - 1);
        if (bytes > 0) {
            buffer[bytes] = '\0';
            printf("\nUser2: %s\nYou: ", buffer);
            fflush(stdout);
        }
    }
    close(fd);
    return NULL;
}

int main() {
    pthread_t recv_thread;
    pthread_create(&recv_thread, NULL, receive_messages, NULL);

    int fd = open(FIFO_SEND, O_WRONLY);
    if (fd == -1) {
        perror("open send");
        return 1;
    }

    printf("=== Chat (User1). Type messages, 'exit' to quit ===\n");

    char message[256];
```

```
    while (1) {
        printf("You: ");
        fflush(stdout);
        fgets(message, sizeof(message), stdin);
        message[strcspn(message, "\n")] = '\0';

        if (strcmp(message, "exit") == 0) {
            break;
        }

        write(fd, message, strlen(message) + 1);
    }

    close(fd);
    return 0;
}
```

4.3. Создайте файл chat_user2.c (аналогично, с заменой FIFO_SEND и FIFO_RECV):

```
c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>

#define FIFO_SEND "/tmp/fifo_user2_to_user1"
#define FIFO_RECV "/tmp/fifo_user1_to_user2"

void *receive_messages(void *arg) {
    int fd = open(FIFO_RECV, O_RDONLY);
```

```
    if (fd == -1) {
        perror("open receive");
        return NULL;
    }

    char buffer[256];
    while (1) {
        ssize_t bytes = read(fd, buffer, sizeof(buffer) - 1);
        if (bytes > 0) {
            buffer[bytes] = '\0';
            printf("\nUser1: %s\nYou: ", buffer);
            fflush(stdout);
        }
    }
    close(fd);
    return NULL;
}

int main() {
    pthread_t recv_thread;
    pthread_create(&recv_thread, NULL, receive_messages, NULL);

    int fd = open(FIFO_SEND, O_WRONLY);
    if (fd == -1) {
        perror("open send");
        return 1;
    }

    printf("=== Chat (User2). Type messages, 'exit' to quit ===\n");
```

```
    char message[256];
    while (1) {
        printf("You: ");
        fflush(stdout);
        fgets(message, sizeof(message), stdin);
        message[strcspn(message, "\n")] = '\0';

        if (strcmp(message, "exit") == 0) {
            break;
        }

        write(fd, message, strlen(message) + 1);
    }

    close(fd);
    return 0;
}
```

4.4. Скомпилируйте и запустите в разных терминалах:

```
bash
gcc -Wall -g chat_user1.c -o user1 -pthread
gcc -Wall -g chat_user2.c -o user2 -pthread

# Терминал 1:
./user1

# Терминал 2:
./user2
```

5. **Неблокирующий режим работы с FIFO**

5.1. Создайте файл nonblocking_reader.c:

```
c
```

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define FIFO_NAME "/tmp/nonblocking_fifo"

int main() {
    mkfifo(FIFO_NAME, 0666);

    // Открытие в неблокирующем режиме
    int fd = open(FIFO_NAME, O_RDONLY | O_NONBLOCK);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    char buffer[256];
    while (1) {
        ssize_t bytes = read(fd, buffer, sizeof(buffer) - 1);
        if (bytes == -1) {
            if (errno == EAGAIN) {
                printf("No data available, doing other work...\n");
                sleep(1);
                continue;
            } else {
                perror("read");
                break;
            }
        } else if (bytes > 0) {
```

```
            buffer[bytes] = '\0';
            printf("Received: %s\n", buffer);
        }
    }

    close(fd);
    return 0;
}
```

## Пример выполнения задания

**Задача примера:** реализовать систему «сервер логов», где несколько клиентов могут отправлять сообщения для записи в общий файл лога через FIFO. Сервер принимает сообщения от клиентов, добавляет временную метку и сохраняет в файл server.log. Клиенты могут отправлять сообщения произвольного содержимого.

**Шаг 1. Реализация сервера**

Файл log_server.c:

c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <signal.h>

#define FIFO_NAME "/tmp/log_fifo"
#define LOG_FILE "server.log"

int running = 1;
```

```
void handle_sigint(int sig) {
    running = 0;
}

int main() {
    signal(SIGINT, handle_sigint);

    // Создание FIFO
    mkfifo(FIFO_NAME, 0666);

    // Открытие файла лога
    int log_fd = open(LOG_FILE, O_WRONLY | O_CREAT | O_APPEND,
0644);
    if (log_fd == -1) {
        perror("open log file");
        return 1;
    }

    printf("Log server started. PID = %d\n", getpid());
    printf("FIFO: %s\n", FIFO_NAME);
    printf("Log file: %s\n", LOG_FILE);
    printf("Press Ctrl+C to stop\n\n");

    while (running) {
        int fifo_fd = open(FIFO_NAME, O_RDONLY);
        if (fifo_fd == -1) {
            if (running) perror("open fifo");
            continue;
        }
```

```
            char buffer[1024];
            ssize_t bytes = read(fifo_fd, buffer, sizeof(buffer) - 1);

            if (bytes > 0) {
                buffer[bytes] = '\0';

                // Добавление временной метки
                time_t now = time(NULL);
                struct tm *tm_info = localtime(&now);
                char timestamp[64];
                strftime(timestamp, sizeof(timestamp), "[%Y-%m-%d
%H:%M:%S]", tm_info);

                // Запись в лог-файл
                dprintf(log_fd, "%s %s\n", timestamp, buffer);
                fflush(stdout);
                printf("Logged: %s %s\n", timestamp, buffer);
            }

            close(fifo_fd);
        }

        printf("\nServer shutting down...\n");
        close(log_fd);
        unlink(FIFO_NAME);  // удаление FIFO

        return 0;
    }
```

**Шаг 2. Реализация клиента**

Файл log_client.c:

c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_NAME "/tmp/log_fifo"

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <message>\n", argv[0]);
        return 1;
    }

    // Составление сообщения из аргументов
    char message[1024] = "";
    for (int i = 1; i < argc; i++) {
        strcat(message, argv[i]);
        if (i < argc - 1) strcat(message, " ");
    }

    int fd = open(FIFO_NAME, O_WRONLY);
    if (fd == -1) {
        perror("open fifo");
        printf("Make sure the server is running\n");
        return 1;
    }

    write(fd, message, strlen(message) + 1);
    printf("Message sent: %s\n", message);
```

```
    close(fd);
    return 0;
}
```

**Шаг 3. Компиляция и запуск**

```
bash
gcc -Wall -g log_server.c -o log_server
gcc -Wall -g log_client.c -o log_client

# Терминал 1 (сервер):
./log_server

# Терминал 2 (клиент):
./log_client "User1 logged in"
./log_client "Database connection established"
./log_client "Error: connection timeout"

# Несколько клиентов одновременно (ещё терминалы):
./log_client "Another client message"
```

**Шаг 4. Пример вывода**

**Сервер:**

```
Log server started. PID = 12345
FIFO: /tmp/log_fifo
Log file: server.log
Press Ctrl+C to stop

Logged: [2024-01-15 14:30:01] User1 logged in
Logged: [2024-01-15 14:30:05] Database connection established
Logged: [2024-01-15 14:30:10] Error: connection timeout
```

**Клиент:**

Message sent: User1 logged in

**Файл** server.log**:**

[2024-01-15 14:30:01] User1 logged in

[2024-01-15 14:30:05] Database connection established

[2024-01-15 14:30:10] Error: connection timeout

**Вывод**

Программа демонстрирует:

- Создание сервера, ожидающего сообщения от клиентов через FIFO.
- Возможность подключения нескольких клиентов (последовательно, FIFO работает как очередь).
- Добавление временной метки к каждому сообщению.
- Сохранение логов в файл.
- Корректное завершение сервера по сигналу SIGINT с очисткой ресурсов.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать систему «удалённое выполнение команд». Сервер принимает команды через FIFO, выполняет их и возвращает результат клиенту через второй FIFO. |
| 2 | Создать простой калькулятор: клиент отправляет выражение (например, "2 + 3"), сервер вычисляет результат и возвращает через второй FIFO. |
| 3 | Реализовать чат для трёх пользователей с использованием FIFO (каждый пользователь имеет свой канал для отправки и общий канал для приёма). |
| 4 | Написать программу-монитор, которая отслеживает системные события. Клиенты отправляют события в FIFO, сервер записывает их в файл с указанием времени и типа события. |
| 5 | Создать систему «очередь задач». Сервер принимает задачи от клиентов, обрабатывает их по очереди и сохраняет результаты в файл. Клиент может отправить задачу и получить её идентификатор. |
| 6 | Реализовать примитивный мессенджер с использованием двух FIFO для каждого клиента. Сервер управляет подключениями и пересылает сообщения адресатам. |

**Обратить внимание:**

– При создании FIFO проверять код возврата mkfifo.

– Удалять FIFO после завершения работы с помощью unlink().

– Учитывать блокирующий характер операций открытия FIFO.

– При работе с несколькими клиентами сервер должен открывать FIFO заново для каждого сообщения или использовать неблокирующий режим.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание FIFO и простейшая передача данных;
   – реализация сервера и клиента для передачи команд;
   – реализация простого чата с двумя FIFO;
   – (по варианту) реализация системы по индивидуальному заданию.
4. Скриншоты (обязательно):
   – работа reader и writer в разных терминалах;
   – работа сервера и клиента команд;
   – работа чата между двумя пользователями;
   – (по варианту) работа программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое именованный канал (FIFO)? Чем он отличается от неименованного канала (pipe)?

2. Как создать FIFO программно? Какой системный вызов для этого используется?

3. Какие права доступа можно установить при создании FIFO?

4. Что происходит при открытии FIFO для чтения, если нет процесса, открывшего его для записи?

5. Что происходит при открытии FIFO для записи, если нет процесса, открывшего его для чтения?

6. Как сделать открытие FIFO неблокирующим?

7. Как организовать двустороннюю связь между двумя процессами с помощью FIFO?

8. Что такое атомарность записи в FIFO? Каков максимальный размер атомарной записи?

9. Как удалить FIFO из файловой системы?

10. Можно ли использовать один FIFO для связи между несколькими клиентами и одним сервером? Как при этом избежать перемешивания данных?

11. Какие преимущества имеют FIFO перед неименованными каналами?

12. Какие ограничения есть у FIFO по сравнению с сокетами?

# Лабораторная работа №17
# Очереди сообщений System V/POSIX

**Цель работы:** освоить использование именованных каналов (FIFO) для организации межпроцессного взаимодействия между неродственными процессами, научиться создавать клиент-серверные приложения с обменом данными через FIFO, реализовать простой чат или систему передачи команд.

**Задачи:**

– Изучить очереди сообщений System V и POSIX как средство IPC.

– Освоить функции System V: msgget, msgsnd, msgrcv, msgctl.

– Изучить функции POSIX: mq_open, mq_send, mq_receive, mq_close, mq_unlink.

– Научиться отправлять структурированные данные различных типов.

– Реализовать клиент-серверное взаимодействие с использованием очередей сообщений.

– Освоить работу с приоритетами сообщений (типы сообщений).

– Изучить отличия между очередями System V и POSIX.

## Теоретические сведения

1. **Очереди сообщений System V**

   Очередь сообщений – это структура данных в ядре, которая хранит сообщения, отправленные разными процессами. Каждое сообщение имеет тип (положительное целое число) и данные. Сообщения могут быть прочитаны выборочно по типу.
2. **Основные функции System V**

| Функция | Назначение |
|---|---|
| msgget(key_t key, int msgflg) | Создание или получение идентификатора очереди сообщений |
| msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg) | Отправка сообщения в очередь |
| msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg) | Приём сообщения из очереди |
| msgctl(int msqid, int cmd, struct msqid_ds *buf) | Управление очередью (удаление, получение информации) |

3. **Структура сообщения System V**

   Каждое сообщение должно начинаться с поля long mtype (тип сообщения, >0). Далее могут следовать любые данные:

   c

   ```
   struct mymsg {
      long mtype;
      char m[100];
      // или другие поля
   };
   ```

4. **Параметры функций System V**
   - key – ключ IPC (можно получить через ftok() или задать константой).
   - msgflg – права доступа и флаги (IPC_CREAT, IPC_EXCL).
   - msgtyp – тип сообщения для чтения:
     - >0 – читать сообщение с указанным типом.
     - ==0 – читать первое сообщение в очереди.
     - <0 – читать сообщение с наименьшим типом, меньшим или равным |msgtyp|.

- msgflg для msgsnd/msgrcv – IPC_NOWAIT (неблокирующий режим).

5. **Очереди сообщений POSIX**

   POSIX предоставляет альтернативный интерфейс для работы с очередями сообщений, который более современен и лучше интегрируется с потоками.

6. **Основные функции POSIX**

| **Функция** | **Назначение** |
|---|---|
| mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr) | Открытие/создание очереди |
| mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio) | Отправка сообщения (с приоритетом) |
| mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio) | Приём сообщения |
| mq_close(mqd_t mqdes) | Закрытие очереди |
| mq_unlink(const char *name) | Удаление очереди из системы |

7. **Отличия System V от POSIX**

| **Характеристика** | **System V** | **POSIX** |
|---|---|---|
| Идентификация | Целочисленный ключ (key) | Строковое имя (например, /myqueue) |
| Типы сообщений | Произвольный long тип | Только приоритет (0-31) |
| Максимальный размер сообщения | Зависит от системы | Задаётся при создании |
| Управление | msgctl | mq_getattr/mq_setattr |
| Потокобезопасность | Нет | Да (может использоваться с потоками) |

8. **Клиент-серверная архитектура на очередях сообщений**
    - Сервер создаёт очередь с известным ключом/именем.
    - Клиенты отправляют запросы в эту очередь, указывая свой PID или специальный тип для ответов.
    - Сервер обрабатывает запросы и отправляет ответы в очередь клиента (или в общую очередь с типом = PID клиента).

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   ```
   mkdir ~/linux_labs/lab17
   cd ~/linux_labs/lab17
   ```

2. **Очереди сообщений System V: простая отправка и приём**

   2.1. Создайте файл sysv_sender.c:

   c

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   #include <sys/ipc.h>
   #include <sys/msg.h>

   #define KEY_PATH "/tmp"
   #define KEY_ID 1234
   #define MSG_SIZE 256

   struct message {
      long mtype;
      char m[MSG_SIZE];
   };
   ```

```
int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    if (key == -1) {
        perror("ftok");
        return 1;
    }

    int msqid = msgget(key, IPC_CREAT | 0666);
    if (msqid == -1) {
        perror("msgget");
        return 1;
    }

    struct message msg;
    msg.mtype = 1;
    strcpy(msg.m, "Hello from sender!");

    if (msgsnd(msqid, &msg, strlen(msg.m) + 1, 0) == -1) {
        perror("msgsnd");
        return 1;
    }

    printf("Sender: message sent\n");
    return 0;
}
```

2.2. Создайте файл sysv_receiver.c:

```
c
#include <stdio.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

#define KEY_PATH "/tmp"
#define KEY_ID 1234
#define MSG_SIZE 256

struct message {
    long mtype;
    char m[MSG_SIZE];
};

int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    if (key == -1) {
        perror("ftok");
        return 1;
    }

    int msqid = msgget(key, IPC_CREAT | 0666);
    if (msqid == -1) {
        perror("msgget");
        return 1;
    }

    struct message msg;
    if (msgrcv(msqid, &msg, MSG_SIZE, 1, 0) == -1) {
        perror("msgrcv");
        return 1;
    }
```

```
    printf("Receiver: received: %s\n", msg.m);

    // Удаление очереди
    msgctl(msqid, IPC_RMID, NULL);

    return 0;
}
```

2.3. Скомпилируйте и выполните:

```
bash
gcc -Wall -g sysv_sender.c -o sysv_sender
gcc -Wall -g sysv_receiver.c -o sysv_receiver

# Терминал 1 (запустить сначала):
./sysv_receiver

# Терминал 2:
./sysv_sender
```

3. **Отправка структурированных данных через System V**

3.1. Создайте файл sysv_struct_sender.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define KEY_PATH "/tmp"
#define KEY_ID 5678

typedef struct {
```

```
    int id;
    char name[50];
    float value;
} Data;

struct message {
    long mtype;
    Data data;
};

int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    int msqid = msgget(key, IPC_CREAT | 0666);

    struct message msg;
    msg.mtype = 1;
    msg.data.id = 100;
    strcpy(msg.data.name, "Test Data");
    msg.data.value = 3.14159;

    msgsnd(msqid, &msg, sizeof(Data), 0);
    printf("Struct sender: data sent (id=%d, name=%s, value=%f)\n",
           msg.data.id, msg.data.name, msg.data.value);

    return 0;
}
```

3.2. Создайте файл sysv_struct_receiver.c:

```
c
#include <stdio.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

#define KEY_PATH "/tmp"
#define KEY_ID 5678

typedef struct {
    int id;
    char name[50];
    float value;
} Data;

struct message {
    long mtype;
    Data data;
};

int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    int msqid = msgget(key, IPC_CREAT | 0666);

    struct message msg;
    msgrcv(msqid, &msg, sizeof(Data), 1, 0);

    printf("Struct receiver: received (id=%d, name=%s, value=%f)\n",
          msg.data.id, msg.data.name, msg.data.value);

    msgctl(msqid, IPC_RMID, NULL);

    return 0;
}
```

4. **Очереди сообщений POSIX**

4.1. Создайте файл posix_sender.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

#define QUEUE_NAME "/myqueue"
#define MAX_SIZE 256

int main() {
    mqd_t mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644,
NULL);
    if (mq == (mqd_t)-1) {
        perror("mq_open");
        return 1;
    }

    char buffer[MAX_SIZE] = "Hello from POSIX sender!";
    if (mq_send(mq, buffer, strlen(buffer) + 1, 1) == -1) {
        perror("mq_send");
        return 1;
    }

    printf("POSIX sender: message sent\n");
    mq_close(mq);
```

```
    return 0;
}
```

4.2. Создайте файл posix_receiver.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

#define QUEUE_NAME "/myqueue"
#define MAX_SIZE 256

int main() {
    mqd_t mq = mq_open(QUEUE_NAME, O_RDONLY);
    if (mq == (mqd_t)-1) {
        perror("mq_open");
        return 1;
    }

    char buffer[MAX_SIZE];
    unsigned int priority;

    if (mq_receive(mq, buffer, MAX_SIZE, &priority) == -1) {
        perror("mq_receive");
        return 1;
    }

    printf("POSIX receiver: received: %s (priority=%u)\n", buffer, priority);
```

```
    mq_close(mq);
    mq_unlink(QUEUE_NAME);

    return 0;
}
```

4.3. Компиляция (требуется линковка с -lrt):

```
bash
gcc -Wall -g posix_sender.c -o posix_sender -lrt
gcc -Wall -g posix_receiver.c -o posix_receiver -lrt

# Терминал 1:
./posix_receiver

# Терминал 2:
./posix_sender
```

5. **Клиент-сервер с приоритетами сообщений (System V)**

5.1. Создайте файл priority_server.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY_PATH "/tmp"
#define KEY_ID 9999
#define MSG_SIZE 256

struct message {
```

```
    long mtype;
    char m[MSG_SIZE];
};

int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    int msqid = msgget(key, IPC_CREAT | 0666);

    printf("Priority server started. Waiting for messages...\n");

    while (1) {
        struct message msg;
        // Читаем сообщения с наивысшим приоритетом (меньший тип)
        if (msgrcv(msqid, &msg, MSG_SIZE, -5, 0) == -1) {
            perror("msgrcv");
            break;
        }

        printf("Server received [type=%ld]: %s\n", msg.mtype, msg.m);

        // Эхо-ответ
        struct message resp;
        resp.mtype = msg.mtype;
        snprintf(resp.m, MSG_SIZE, "Echo: %s", msg.m);
        msgsnd(msqid, &resp, strlen(resp.m) + 1, 0);
    }

    msgctl(msqid, IPC_RMID, NULL);
    return 0;
}
```

5.2. Создайте файл priority_client.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY_PATH "/tmp"
#define KEY_ID 9999
#define MSG_SIZE 256

struct message {
    long mtype;
    char m[MSG_SIZE];
};

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

    long type = atol(argv[1]);
    char * = argv[2];

    key_t key = ftok(KEY_PATH, KEY_ID);
    int msqid = msgget(key, 0666);
```

```
    struct message msg;
    msg.mtype = type;
    strcpy(msg.m, );

    msgsnd(msqid, &msg, strlen(msg.m) + 1, 0);
    printf("Client: sent [type=%ld]: %s\n", type, );

    // Ожидание ответа
    struct message resp;
    msgrcv(msqid, &resp, MSG_SIZE, type, 0);
    printf("Client: received: %s\n", resp.m);

    return 0;
}
```

5.3. Запустите сервер и несколько клиентов с разными приоритетами:

```
bash
gcc -Wall -g priority_server.c -o pserver
gcc -Wall -g priority_client.c -o pclient

# Терминал 1:
./pserver

# Терминал 2:
./pclient 1 "Low priority message"
./pclient 5 "High priority message"
./pclient 3 "Normal priority message"
```

## Пример выполнения задания

**Задача примера:** реализовать систему управления задачами (task manager) с использованием очереди сообщений System V. Клиенты могут отправлять запросы разных типов: добавление задачи (тип 1), получение списка задач (тип 2), удаление задачи (тип 3). Сервер обрабатывает запросы с учётом приоритета (тип сообщения = приоритет задачи) и возвращает результаты.

**Шаг 1. Реализация сервера**

Файл task_server.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY_PATH "/tmp"
#define KEY_ID 7777
#define MAX_TASKS 100
#define MSG_SIZE 512

typedef struct {
    int id;
    char description[200];
    int priority;
} Task;

struct message {
    long mtype;         // тип запроса/ответа
```

```
    int client_pid;      // PID клиента для ответа
    Task task;           // данные задачи
    char response[MSG_SIZE];
};

Task tasks[MAX_TASKS];
int task_count = 0;
int next_id = 1;

void add_task(int priority, const char *desc) {
    if (task_count < MAX_TASKS) {
        tasks[task_count].id = next_id++;
        strcpy(tasks[task_count].description, desc);
        tasks[task_count].priority = priority;
        task_count++;
        printf("Task added: ID=%d, priority=%d\n", tasks[task_count-1].id,
priority);
    }
}

void list_tasks(char *response) {
    response[0] = '\0';
    for (int i = 0; i < task_count; i++) {
        char line[256];
        snprintf(line, sizeof(line), "ID=%d, Priority=%d, Desc=%s\n",
                tasks[i].id, tasks[i].priority, tasks[i].description);
        strcat(response, line);
    }
    if (task_count == 0) {
        strcpy(response, "No tasks available.\n");
```

```
    }
}

void delete_task(int id, char *response) {
    for (int i = 0; i < task_count; i++) {
        if (tasks[i].id == id) {
            for (int j = i; j < task_count - 1; j++) {
                tasks[j] = tasks[j + 1];
            }
            task_count--;
            sprintf(response, "Task %d deleted successfully.", id);
            return;
        }
    }
    sprintf(response, "Task %d not found.", id);
}

int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    int msqid = msgget(key, IPC_CREAT | 0666);

    printf("=== Task Manager Server ===\n");
    printf("Server started. Queue ID: %d\n", msqid);
    printf("Waiting for client requests...\n\n");

    while (1) {
        struct message msg;

        // Приём запроса (читаем все типы сообщений)
        if (msgrcv(msqid, &msg, sizeof(msg) - sizeof(long), 0, 0) == -1) {
```

```
            perror("msgrcv");
            continue;
        }

        printf("Received request type %ld from PID %d\n", msg.mtype, msg.client_pid);

        struct message response;
        response.mtype = msg.client_pid;  // ответ отправляем в очередь клиента
        response.client_pid = getpid();

        switch (msg.mtype) {
            case 1:  // ADD_TASK
                add_task(msg.task.priority, msg.task.description);
                sprintf(response.response, "Task added with ID=%d", next_id - 1);
                break;
            case 2:  // LIST_TASKS
                list_tasks(response.response);
                break;
            case 3:  // DELETE_TASK
                delete_task(msg.task.id, response.response);
                break;
            default:
                sprintf(response.response, "Unknown command type: %ld", msg.mtype);
        }

        msgsnd(msqid, &response, sizeof(response) - sizeof(long), 0);
        printf("Response sent to PID %d\n\n", msg.client_pid);
```

```
    }

    msgctl(msqid, IPC_RMID, NULL);
    return 0;
}
```

**Шаг 2. Реализация клиента**

Файл task_client.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

#define KEY_PATH "/tmp"
#define KEY_ID 7777
#define MSG_SIZE 512

struct message {
    long mtype;
    int client_pid;
    struct {
        int id;
        char description[200];
        int priority;
    } task;
    char response[MSG_SIZE];
};
```

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage:\n");
        fprintf(stderr, "  %s add <priority> <description>\n", argv[0]);
        fprintf(stderr, "  %s list\n", argv[0]);
        fprintf(stderr, "  %s delete <id>\n", argv[0]);
        return 1;
    }

    key_t key = ftok(KEY_PATH, KEY_ID);
    int msqid = msgget(key, 0666);
    if (msqid == -1) {
        perror("msgget (server not running?)");
        return 1;
    }

    struct message msg;
    msg.client_pid = getpid();

    if (strcmp(argv[1], "add") == 0 && argc >= 4) {
        msg.mtype = 1;  // ADD_TASK
        msg.task.priority = atoi(argv[2]);
        strcpy(msg.task.description, argv[3]);
        printf("Sending ADD request: priority=%d, desc=%s\n",
               msg.task.priority, msg.task.description);
    } else if (strcmp(argv[1], "list") == 0) {
        msg.mtype = 2;  // LIST_TASKS
        printf("Sending LIST request\n");
    } else if (strcmp(argv[1], "delete") == 0 && argc >= 3) {
        msg.mtype = 3;  // DELETE_TASK
```

```
            msg.task.id = atoi(argv[2]);
            printf("Sending DELETE request: id=%d\n", msg.task.id);
        } else {
            fprintf(stderr, "Invalid command\n");
            return 1;
        }

        // Отправка запроса
        if (msgsnd(msqid, &msg, sizeof(msg) - sizeof(long), 0) == -1) {
            perror("msgsnd");
            return 1;
        }

        // Ожидание ответа
        struct message response;
        if (msgrcv(msqid, &response, sizeof(response) - sizeof(long), getpid(), 0)
== -1) {
            perror("msgrcv");
            return 1;
        }

        printf("\n=== Server Response ===\n");
        printf("%s\n", response.response);
        printf("======================\n");

        return 0;
    }
```

**Шаг 3. Компиляция и запуск**

bash

```
gcc -Wall -g task_server.c -o task_server
```

gcc -Wall -g task_client.c -o task_client

*# Терминал 1 (сервер):*

./task_server

*# Терминал 2 (клиент):*

./task_client add 5 "Implement login feature"

./task_client add 1 "Fix minor bug"

./task_client add 10 "Deploy to production"

./task_client list

./task_client delete 2

./task_client list

**Шаг 4. Пример вывода**

**Сервер:**

=== Task Manager Server ===

Server started. Queue ID: 65536

Waiting for client requests...

Received request type 1 from PID 12345

Task added: ID=1, priority=5

Response sent to PID 12345

Received request type 1 from PID 12345

Task added: ID=2, priority=1

Response sent to PID 12345

Received request type 1 from PID 12345

Task added: ID=3, priority=10

Response sent to PID 12345

Received request type 2 from PID 12345

Response sent to PID 12345

Received request type 3 from PID 12345

Response sent to PID 12345

Received request type 2 from PID 12345

Response sent to PID 12345

**Клиент:**

Sending ADD request: priority=5, desc=Implement login feature

=== Server Response ===

Task added with ID=1

==========

Sending ADD request: priority=1, desc=Fix minor bug

=== Server Response ===

Task added with ID=2

==========

Sending LIST request

=== Server Response ===

ID=1, Priority=5, Desc=Implement login feature

ID=2, Priority=1, Desc=Fix minor bug

ID=3, Priority=10, Desc=Deploy to production

==========

Sending DELETE request: id=2

=== Server Response ===
Task 2 deleted successfully.
==========

Sending LIST request

=== Server Response ===
ID=1, Priority=5, Desc=Implement login feature
ID=3, Priority=10, Desc=Deploy to production

==========

**Вывод**

Программа демонстрирует:

- Создание сервера с фиксированным ключом очереди.
- Клиент-серверное взаимодействие с использованием типов сообщений для разных операций.
- Отправку структурированных данных (Task) между процессами.
- Использование PID клиента для адресации ответов.
- Обработку различных типов запросов (добавление, список, удаление).

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать систему «библиотека»: клиенты могут добавлять книги (название, автор), искать книги по автору, удалять книги. Использовать очередь сообщений System V. |
| 2 | Создать систему «банковские счета»: клиенты могут создавать счёт, проверять баланс, пополнять и снимать средства. Использовать очередь сообщений POSIX. |
| 3 | Реализовать систему «заказы»: клиенты отправляют заказы (товар, количество), сервер обрабатывает их и возвращает стоимость. Поддержка приоритетов (срочные заказы обрабатываются быстрее). |
| 4 | Написать программу «калькулятор»: клиент отправляет выражение, сервер вычисляет результат и возвращает. Поддержка разных типов операций (сложение, умножение и т.д.) через разные типы сообщений. |
| 5 | Создать систему «голосование»: клиенты отправляют голоса за варианты (1-5), сервер подсчитывает результаты. Клиенты могут запросить текущую статистику. |
| 6 | Реализовать систему «чат» с использованием очереди сообщений. Сервер пересылает сообщения от одного клиента другому. Каждый клиент имеет свою очередь для получения сообщений. |

**Обратить внимание:**

- Для System V необходимо использовать ftok() для генерации уникального ключа.
- После завершения работы удалять очередь сообщений (msgctl или mq_unlink).
- Учитывать ограничения на размер сообщения (обычно 8192 байта).
- При использовании POSIX очередей линковать с -lrt.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – простая отправка/приём сообщений System V;
   – отправка структурированных данных;
   – работа с очередями POSIX;
   – клиент-сервер с приоритетами;
   – (по варианту) реализация системы по индивидуальному заданию.
4. Скриншоты (обязательно):
   – работа sysv_sender и sysv_receiver;
   – работа posix_sender и posix_receiver;
   – работа priority_server и priority_client;
   – (по варианту) работа программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое очередь сообщений System V? Как она создаётся?
2. Как сгенерировать уникальный ключ для очереди сообщений?
3. Что такое тип сообщения в System V? Как он используется для приоритетов?
4. Как отправить структурированные данные через очередь сообщений?
5. Как удалить очередь сообщений System V?
6. Чем отличаются очереди сообщений POSIX от System V?
7. Как создать очередь сообщений POSIX? Какой флаг нужно указать при компиляции?
8. Как задать максимальный размер сообщения и максимальное количество сообщений в очереди POSIX?
9. Что такое приоритет сообщения в POSIX очередях?
10. Как организовать клиент-серверное взаимодействие с очередями сообщений?
11. Какие ограничения на размер сообщения существуют в System V и POSIX?
12. Что произойдёт, если очередь сообщений заполнена и процесс пытается отправить ещё одно сообщение?

# Лабораторная работа №18

# Разделяемая память

**Цель работы:** изучить механизм разделяемой памяти (Shared Memory) для организации высокоскоростного обмена большими объёмами данных между процессами, освоить использование семафоров для синхронизации доступа, реализовать кольцевой буфер в разделяемой памяти.

**Задачи:**

– Изучить разделяемую память System V и POSIX как средство IPC.

– Освоить функции работы с разделяемой памятью: shmget, shmat, shmdt, shmctl.

– Научиться создавать и присоединять сегменты разделяемой памяти.

– Изучить использование семафоров для синхронизации доступа к разделяемой памяти.

– Реализовать кольцевой буфер (ring buffer) в разделяемой памяти.

– Реализовать producer-consumer с использованием разделяемой памяти и семафоров.

– Сравнить производительность разделяемой памяти с другими IPC-механизмами.

## Теоретические сведения

1. **Разделяемая память (Shared Memory)**

   Разделяемая память – это сегмент памяти, который одновременно доступен нескольким процессам. Это самый быстрый способ IPC, так как данные не копируются между процессами. Однако требует дополнительной синхронизации (семафоры, мьютексы).

2. **Разделяемая память System V**

| Функция | Назначение |
|---|---|
| shmget(key_t key, size_t size, int shmflg) | Создание или получение идентификатора сегмента |
| shmat(int shmid, const void *shmaddr, int shmflg) | Присоединение сегмента к адресному пространству процесса |
| shmdt(const void *shmaddr) | Отсоединение сегмента |
| shmctl(int shmid, int cmd, struct shmid_ds *buf) | Управление сегментом (удаление, получение информации) |

3. **Разделяемая память POSIX**

| Функция | Назначение |
|---|---|
| shm_open(const char *name, int oflag, mode_t mode) | Создание/открытие объекта разделяемой памяти |
| ftruncate(int fd, off_t length) | Установка размера объекта |
| mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset) | Отображение объекта в адресное пространство |
| munmap(void *addr, size_t length) | Отмена отображения |
| shm_unlink(const char *name) | Удаление объекта |

4. **Структура кольцевого буфера (Ring Buffer)**

   Кольцевой буфер – это структура данных фиксированного размера, которая позволяет эффективно передавать данные между producer и consumer.

   c

```
struct ring_buffer {
    char data[BUFFER_SIZE];
    int write_pos;    // позиция записи
    int read_pos;     // позиция чтения
```

```
    int available;    // количество доступных байт
};
```

5. **Синхронизация доступа**

   Для синхронизации доступа к разделяемой памяти используются семафоры:

   - Семафор empty – количество свободных мест в буфере.
   - Семафор full – количество заполненных мест.
   - Семафор mutex – защита критических секций (индексов).

6. **Преимущества разделяемой памяти**
   - Высокая скорость (минуя копирование через ядро).
   - Подходит для больших объёмов данных.
   - Поддерживается как System V, так и POSIX.
7. **Недостатки разделяемой памяти**
   - Требуется ручная синхронизация.
   - Сложность отладки.
   - Необходимость управления размерами сегментов.

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   ```
   mkdir ~/linux_labs/lab18
   cd ~/linux_labs/lab18
   ```

2. **Разделяемая память System V: простая передача данных**

   2.1. Создайте файл sysv_shm_writer.c:

   c

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   #include <sys/ipc.h>
   ```

```
#include <sys/shm.h>
#include <unistd.h>

#define KEY_PATH "/tmp"
#define KEY_ID 1234
#define SHM_SIZE 1024

int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        return 1;
    }

    char *shmaddr = shmat(shmid, NULL, 0);
    if (shmaddr == (char *)-1) {
        perror("shmat");
        return 1;
    }

    strcpy(shmaddr, "Hello from writer!");
    printf("Writer: data written to shared memory\n");

    shmdt(shmaddr);
    return 0;
}
```

2.2. Создайте файл sysv_shm_reader.c:

c

```
#include <stdio.h>
```

```
#include <sys/ipc.h>
#include <sys/shm.h>

#define KEY_PATH "/tmp"
#define KEY_ID 1234
#define SHM_SIZE 1024

int main() {
    key_t key = ftok(KEY_PATH, KEY_ID);
    int shmid = shmget(key, SHM_SIZE, 0666);
    if (shmid == -1) {
        perror("shmget");
        return 1;
    }

    char *shmaddr = shmat(shmid, NULL, 0);
    if (shmaddr == (char *)-1) {
        perror("shmat");
        return 1;
    }

    printf("Reader: read: %s\n", shmaddr);

    shmdt(shmaddr);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

2.3. Скомпилируйте и выполните:

bash

gcc -Wall -g sysv_shm_writer.c -o sysv_writer

gcc -Wall -g sysv_shm_reader.c -o sysv_reader

./sysv_writer

./sysv_reader

3. **Разделяемая память POSIX**

3.1. Создайте файл posix_shm_writer.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_NAME "/myshm"
#define SHM_SIZE 1024

int main() {
    int fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (fd == -1) {
        perror("shm_open");
        return 1;
    }

    ftruncate(fd, SHM_SIZE);

    char *ptr = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) {
```

```
        perror("mmap");
        return 1;
    }

    strcpy(ptr, "Hello from POSIX writer!");
    printf("POSIX writer: data written\n");

    munmap(ptr, SHM_SIZE);
    close(fd);

    return 0;
}
```

3.2. Создайте файл posix_shm_reader.c:

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

#define SHM_NAME "/myshm"
#define SHM_SIZE 1024

int main() {
    int fd = shm_open(SHM_NAME, O_RDWR, 0666);
    if (fd == -1) {
        perror("shm_open");
        return 1;
    }
```

```
    char *ptr = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        return 1;
    }

    printf("POSIX reader: read: %s\n", ptr);

    munmap(ptr, SHM_SIZE);
    close(fd);
    shm_unlink(SHM_NAME);

    return 0;
}
```

3.3. Компиляция (требуется -lrt):

```
bash
gcc -Wall -g posix_shm_writer.c -o posix_writer -lrt
gcc -Wall -g posix_shm_reader.c -o posix_reader -lrt

./posix_writer
./posix_reader
```

4. **Producer-Consumer с разделяемой памятью и семафорами (System V)**

4.1. Создайте заголовочный файл ring_buffer.h:

```
c
#ifndef RING_BUFFER_H
#define RING_BUFFER_H

#define BUFFER_SIZE 256
#define SHM_KEY 0x1234
```

```
#define SEM_EMPTY_KEY 0x5678
#define SEM_FULL_KEY 0x5679
#define SEM_MUTEX_KEY 0x5680

struct ring_buffer {
    char data[BUFFER_SIZE];
    int write_pos;
    int read_pos;
    int available;
};

#endif
```

4.2. Создайте файл producer.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <unistd.h>
#include "ring_buffer.h"

void sem_wait(int semid, int sem_num) {
    struct sembuf op = {sem_num, -1, 0};
    semop(semid, &op, 1);
}

void sem_signal(int semid, int sem_num) {
    struct sembuf op = {sem_num, 1, 0};
```

```
    semop(semid, &op, 1);
}

int main() {
    // Создание разделяемой памяти
    int shmid = shmget(SHM_KEY, sizeof(struct ring_buffer), IPC_CREAT |
0666);
    struct ring_buffer *rb = shmat(shmid, NULL, 0);

    // Создание семафоров
    int sem_empty = semget(SEM_EMPTY_KEY, 1, IPC_CREAT | 0666);
    int sem_full = semget(SEM_FULL_KEY, 1, IPC_CREAT | 0666);
    int sem_mutex = semget(SEM_MUTEX_KEY, 1, IPC_CREAT | 0666);

    // Инициализация семафоров
    semctl(sem_empty, 0, SETVAL, BUFFER_SIZE);
    semctl(sem_full, 0, SETVAL, 0);
    semctl(sem_mutex, 0, SETVAL, 1);

    // Инициализация кольцевого буфера
    if (shmid != -1) {
        rb->write_pos = 0;
        rb->read_pos = 0;
        rb->available = 0;
    }

    const char *messages[] = {"Msg1", "Msg2", "Msg3", "Msg4", "Msg5",
"END"};
    int msg_count = 0;
```

```
    while (msg_count < 6) {
        sem_wait(sem_empty, 0);      // ждём свободное место
        sem_wait(sem_mutex, 0);      // захват мьютекса

        // Запись в буфер
        const char *msg = messages[msg_count];
        int len = strlen(msg) + 1;
        for (int i = 0; i < len; i++) {
            rb->data[rb->write_pos] = msg[i];
            rb->write_pos = (rb->write_pos + 1) % BUFFER_SIZE;
        }
        rb->available += len;

        printf("Producer: wrote '%s' (pos=%d, available=%d)\n",
               msg, rb->write_pos, rb->available);
        msg_count++;

        sem_signal(sem_mutex, 0);    // освобождение мьютекса
        sem_signal(sem_full, 0);     // увеличиваем счётчик заполненных

        sleep(1);
    }

    shmdt(rb);
    return 0;
}
```

4.3. Создайте файл consumer.c:

```
c
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <unistd.h>
#include "ring_buffer.h"

void sem_wait(int semid, int sem_num) {
    struct sembuf op = {sem_num, -1, 0};
    semop(semid, &op, 1);
}

void sem_signal(int semid, int sem_num) {
    struct sembuf op = {sem_num, 1, 0};
    semop(semid, &op, 1);
}

int main() {
    // Получение разделяемой памяти
    int shmid = shmget(SHM_KEY, sizeof(struct ring_buffer), 0666);
    struct ring_buffer *rb = shmat(shmid, NULL, 0);

    // Получение семафоров
    int sem_empty = semget(SEM_EMPTY_KEY, 1, 0666);
    int sem_full = semget(SEM_FULL_KEY, 1, 0666);
    int sem_mutex = semget(SEM_MUTEX_KEY, 1, 0666);

    char buffer[256];
    int received = 0;
```

```
while (received < 6) {
    sem_wait(sem_full, 0);       // ждём данные
    sem_wait(sem_mutex, 0);      // захват мьютекса

    // Чтение из буфера
    if (rb->available > 0) {
        int i = 0;
        while (rb->data[rb->read_pos] != '\0' && i < 255) {
            buffer[i++] = rb->data[rb->read_pos];
            rb->read_pos = (rb->read_pos + 1) % BUFFER_SIZE;
        }
        buffer[i] = '\0';
        rb->read_pos = (rb->read_pos + 1) % BUFFER_SIZE;
        rb->available -= (i + 1);

        printf("Consumer: read '%s' (pos=%d, available=%d)\n",
               buffer, rb->read_pos, rb->available);
        received++;
    }

    sem_signal(sem_mutex, 0);    // освобождение мьютекса
    sem_signal(sem_empty, 0);    // увеличиваем счётчик свободных

    sleep(1);
}

// Очистка
shmdt(rb);
shmctl(shmid, IPC_RMID, NULL);
semctl(sem_empty, 0, IPC_RMID);
```

```
    semctl(sem_full, 0, IPC_RMID);
    semctl(sem_mutex, 0, IPC_RMID);

    return 0;
}
```

4.4. Скомпилируйте и запустите в разных терминалах:

```
bash
gcc -Wall -g producer.c -o producer
gcc -Wall -g consumer.c -o consumer

# Терминал 1:
./consumer

# Терминал 2:
./producer
```

5. **Кольцевой буфер для передачи больших объёмов данных**

5.1. Модифицируйте код для передачи больших сообщений (например, изображений или структур данных).

## Пример выполнения задания

**Задача примера:** реализовать систему обработки видеокадров с использованием разделяемой памяти. Процесс-захватчик (producer) записывает кадры (структуры с данными) в кольцевой буфер, процесс-обработчик (consumer) читает кадры и выполняет обработку (например, преобразование в оттенки серого). Для синхронизации использовать семафоры POSIX.

**Шаг 1. Реализация заголовочного файла**

Файл frame_buffer.h:

```
c
#ifndef FRAME_BUFFER_H
```

```
#define FRAME_BUFFER_H

#include <semaphore.h>

#define FRAME_WIDTH 64
#define FRAME_HEIGHT 64
#define FRAME_SIZE (FRAME_WIDTH * FRAME_HEIGHT)
#define BUFFER_CAPACITY 4

typedef struct {
    int frame_id;
    unsigned char data[FRAME_SIZE];
} Frame;

typedef struct {
    Frame frames[BUFFER_CAPACITY];
    int write_idx;
    int read_idx;
    int count;
    sem_t empty;
    sem_t full;
    sem_t mutex;
} FrameBuffer;

#endif
```

**Шаг 2. Реализация процесса-захватчика (producer)**

Файл capture.c:

```
c
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <semaphore.h>
#include "frame_buffer.h"

#define SHM_NAME "/framebuffer"

int main() {
    // Создание разделяемой памяти
    int fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    ftruncate(fd, sizeof(FrameBuffer));
    FrameBuffer *fb = mmap(NULL, sizeof(FrameBuffer),
                           PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    // Инициализация семафоров
    sem_init(&fb->empty, 1, BUFFER_CAPACITY);
    sem_init(&fb->full, 1, 0);
    sem_init(&fb->mutex, 1, 1);
    fb->write_idx = 0;
    fb->read_idx = 0;
    fb->count = 0;

    printf("Capture process started. Simulating frame capture...\n");

    for (int frame_id = 1; frame_id <= 10; frame_id++) {
        sem_wait(&fb->empty);
        sem_wait(&fb->mutex);
```

```
        // Запись кадра
        Frame *frame = &fb->frames[fb->write_idx];
        frame->frame_id = frame_id;

        // Имитация заполнения данными (случайные пиксели)
        for (int i = 0; i < FRAME_SIZE; i++) {
            frame->data[i] = rand() % 256;
        }

        printf("Capture: wrote frame %d at slot %d (count=%d)\n",
               frame_id, fb->write_idx, fb->count + 1);

        fb->write_idx = (fb->write_idx + 1) % BUFFER_CAPACITY;
        fb->count++;

        sem_post(&fb->mutex);
        sem_post(&fb->full);

        usleep(500000);  // 0.5 секунды на кадр
    }

    printf("Capture finished.\n");
    sleep(2);

    munmap(fb, sizeof(FrameBuffer));
    close(fd);

    return 0;
}
```

**Шаг 3. Реализация процесса-обработчика (consumer)**

Файл process.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <semaphore.h>
#include "frame_buffer.h"

#define SHM_NAME "/framebuffer"

int main() {
    // Открытие разделяемой памяти
    int fd = shm_open(SHM_NAME, O_RDWR, 0666);
    FrameBuffer *fb = mmap(NULL, sizeof(FrameBuffer),
                           PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    printf("Processing process started.\n");

    int processed = 0;
    while (processed < 10) {
        sem_wait(&fb->full);
        sem_wait(&fb->mutex);

        if (fb->count > 0) {
            Frame *frame = &fb->frames[fb->read_idx];

            // Имитация обработки (преобразование в оттенки серого)
            // Для простоты – просто вычисляем среднее значение
```

```
            int sum = 0;
            for (int i = 0; i < FRAME_SIZE; i++) {
                sum += frame->data[i];
            }
            int avg = sum / FRAME_SIZE;

            printf("Process: frame %d processed, avg_pixel=%d (slot %d, count=%d)\n",
                   frame->frame_id, avg, fb->read_idx, fb->count - 1);

            fb->read_idx = (fb->read_idx + 1) % BUFFER_CAPACITY;
            fb->count--;
            processed++;
        }

        sem_post(&fb->mutex);
        sem_post(&fb->empty);

        usleep(800000);  // 0.8 секунды на обработку
    }

    printf("Processing finished.\n");

    // Очистка
    munmap(fb, sizeof(FrameBuffer));
    close(fd);
    shm_unlink(SHM_NAME);
    sem_destroy(&fb->empty);
    sem_destroy(&fb->full);
    sem_destroy(&fb->mutex);
```

```
    return 0;
}
```

**Шаг 4. Компиляция и запуск**

bash

```
gcc -Wall -g capture.c -o capture -lrt -pthread
gcc -Wall -g process.c -o process -lrt -pthread

# Терминал 1 (обработчик):
./process

# Терминал 2 (захватчик):
./capture
```

**Шаг 5. Пример вывода**

**Терминал 1 (process):**

```
Processing process started.
Process: frame 1 processed, avg_pixel=127 (slot 0, count=0)
Process: frame 2 processed, avg_pixel=128 (slot 1, count=0)
Process: frame 3 processed, avg_pixel=126 (slot 2, count=0)
Process: frame 4 processed, avg_pixel=129 (slot 3, count=0)
Process: frame 5 processed, avg_pixel=127 (slot 0, count=0)
Process: frame 6 processed, avg_pixel=128 (slot 1, count=0)
Process: frame 7 processed, avg_pixel=126 (slot 2, count=0)
Process: frame 8 processed, avg_pixel=129 (slot 3, count=0)
Process: frame 9 processed, avg_pixel=127 (slot 0, count=0)
Process: frame 10 processed, avg_pixel=128 (slot 1, count=0)
Processing finished.
```

**Терминал 2 (capture):**

Capture process started. Simulating frame capture...

Capture: wrote frame 1 at slot 0 (count=1)

Capture: wrote frame 2 at slot 1 (count=2)

Capture: wrote frame 3 at slot 2 (count=3)

Capture: wrote frame 4 at slot 3 (count=4)

Capture: wrote frame 5 at slot 0 (count=4)

Capture: wrote frame 6 at slot 1 (count=4)

Capture: wrote frame 7 at slot 2 (count=4)

Capture: wrote frame 8 at slot 3 (count=4)

Capture: wrote frame 9 at slot 0 (count=4)

Capture: wrote frame 10 at slot 1 (count=4)

Capture finished.

**Вывод**

Программа демонстрирует:

- Использование разделяемой памяти POSIX для обмена большими объёмами данных (кадры 64×64 пикселей).
- Реализацию кольцевого буфера для передачи данных между producer и consumer.
- Использование семафоров POSIX для синхронизации доступа.
- Разделение процессов захвата и обработки данных.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать программу для передачи сообщений между процессами через разделяемую память. Использовать кольцевой буфер и семафоры. Добавить поддержку нескольких читателей. |
| 2 | Создать систему «сенсоры»: несколько процессов-датчиков записывают данные в разделяемую память (температура, давление), один процесс отображает их. |
| 3 | Реализовать программу для параллельной обработки массива данных: родительский процесс загружает массив в разделяемую память, дочерние процессы обрабатывают свои части, родитель собирает результаты. |
| 4 | Написать программу «логирование»: несколько процессов записывают сообщения в кольцевой буфер, один процесс сохраняет их в файл. Использовать разделяемую память и семафоры. |
| 5 | Создать программу для потоковой передачи звука: producer записывает PCM-данные, consumer воспроизводит. Использовать разделяемую память для буферизации. |
| 6 | Реализовать систему «скоростной обмен структурами»: два процесса обмениваются сложными структурами данных (например, координаты объектов) через разделяемую память. |

**Обратить внимание:**

- Всегда проверять возвращаемые значения функций (shmget, shmat, semget и т.д.).
- Удалять сегменты разделяемой памяти после завершения работы (shmctl или shm_unlink).
- Для синхронизации использовать семафоры (System V или POSIX).
- Учитывать выравнивание структур при передаче данных.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – работа с разделяемой памятью System V;
   – работа с разделяемой памятью POSIX;
   – реализация кольцевого буфера;
   – синхронизация с помощью семафоров;
   – (по варианту) реализация системы по индивидуальному заданию.
4. Скриншоты (обязательно):
   – работа sysv_writer и sysv_reader;
   – работа posix_writer и posix_reader;
   – работа producer и consumer с кольцевым буфером;
   – (по варианту) работа программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое разделяемая память? В чём её преимущество перед другими IPC-механизмами?

2. Как создать сегмент разделяемой памяти System V? Какие функции используются?

3. Как присоединить и отсоединить сегмент разделяемой памяти?

4. Как удалить сегмент разделяемой памяти?

5. В чём отличие разделяемой памяти POSIX от System V?

6. Как создать объект разделяемой памяти POSIX? Какие функции используются?

7. Как отобразить объект разделяемой памяти в адресное пространство процесса?

8. Почему разделяемая память требует дополнительной синхронизации?

9. Как реализовать кольцевой буфер в разделяемой памяти?

10. Какие семафоры (POSIX или System V) удобнее использовать с разделяемой памятью?

11. Какие ограничения на размер сегмента разделяемой памяти существуют?

12. Как узнать, какие сегменты разделяемой памяти существуют в системе?

# Лабораторная работа №19

# TCP-клиент

**Цель работы:** освоить создание клиентских приложений для взаимодействия по протоколу TCP, научиться устанавливать соединение с удалённым хостом, отправлять HTTP-запросы и получать ответы, обрабатывать ошибки соединения и сетевые сбои.

**Задачи:**

– Изучить основные системные вызовы для TCP-сокетов: socket, connect, send, recv, close.

– Освоить преобразование адресов с помощью inet_addr, inet_pton, gethostbyname.

– Научиться работать с struct sockaddr_in и struct hostent.

– Реализовать TCP-клиент, подключающийся к заданному хосту и порту.

– Разработать простой HTTP-клиент для отправки GET-запросов и получения ответов.

– Научиться обрабатывать ошибки соединения (недоступный хост, отказ в соединении, таймаут).

## Теоретические сведения

1. **Протокол TCP (Transmission Control Protocol)**

   TCP – это надёжный, установленный, потоковый протокол транспортного уровня. Гарантирует доставку данных в правильном порядке без потерь и дублирования.
2. **Основные системные вызовы для TCP-клиента**

| Функция | Назначение |
|---|---|
| socket(int domain, int type, int protocol) | Создание сокета |
| connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen) | Установка соединения с сервером |
| send(int sockfd, const void *buf, size_t len, int flags) | Отправка данных |
| recv(int sockfd, void *buf, size_t len, int flags) | Приём данных |
| close(int fd) | Закрытие сокета |

3. **Структуры адресов**

   c

   struct sockaddr_in {

   sa_family_t sin_family; *// AF_INET*

   in_port_t sin_port; *// порт (в сетевом порядке)*

   struct in_addr sin_addr; *// IP-адрес*

   };

4. **Преобразование адресов**
   - inet_addr(const char *cp) – преобразует строку IPv4 в двоичный формат (устаревшая).
   - inet_pton(int af, const char *src, void *dst) – современная функция.
   - gethostbyname(const char *name) – получение IP-адреса по доменному имени.
5. **HTTP-протокол (основы)**

   HTTP GET-запрос имеет следующую структуру:

   GET /path HTTP/1.1\r\n

   Host: example.com\r\n

   User-Agent: MyClient/1.0\r\n

   Connection: close\r\n

\r\n

6. **Обработка ошибок**

   Основные ошибки при соединении:

   - ECONNREFUSED – сервер не слушает указанный порт.
   - ETIMEDOUT – таймаут соединения.
   - EHOSTUNREACH – хост недоступен.
   - ENETUNREACH – сеть недоступна.

7. **Таймауты**

   Для установки таймаута соединения можно использовать alarm() с сигналом SIGALRM или неблокирующие сокеты с select/poll.

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab19

   cd ~/linux_labs/lab19

2. **Простейший TCP-клиент (echo-сервер)**

   2.1. Создайте файл tcp_client.c (подключение к echo-серверу на порту 7):

   c

   #include <stdio.h>

   #include <stdlib.h>

   #include <string.h>

   #include <unistd.h>

   #include <sys/socket.h>

   #include <netinet/in.h>

   #include <arpa/inet.h>

   #define PORT 7

   #define BUFFER_SIZE 1024

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <server_ip>\n", argv[0]);
        return 1;
    }

    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("socket");
        return 1;
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr(argv[1]);

    if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) ==
-1) {
        perror("connect");
        close(sock);
        return 1;
    }

    printf("Connected to %s:%d\n", argv[1], PORT);

    char message[] = "Hello, TCP server!\n";
    send(sock, message, strlen(message), 0);
    printf("Sent: %s", message);
```

```
    char buffer[BUFFER_SIZE];
    int bytes = recv(sock, buffer, BUFFER_SIZE - 1, 0);
    if (bytes > 0) {
        buffer[bytes] = '\0';
        printf("Received: %s", buffer);
    }

    close(sock);
    return 0;
}
```

2.2. Скомпилируйте и протестируйте (нужен запущенный echo-сервер):

```
bash
gcc -Wall -g tcp_client.c -o tcp_client
# Для тестирования можно использовать локальный echo-сервер или nc -l -p 7
```

3. **HTTP-клиент: загрузка веб-страницы**

3.1. Создайте файл http_client.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE 4096
```

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

    const char *host = argv[1];
    int port = atoi(argv[2]);

    // Получение IP-адреса по доменному имени
    struct hostent *server = gethostbyname(host);
    if (server == NULL) {
        fprintf(stderr, "No such host: %s\n", host);
        return 1;
    }

    // Создание сокета
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("socket");
        return 1;
    }

    // Настройка адреса сервера
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    memcpy(&server_addr.sin_addr.s_addr, server->h_addr, server->h_length);
```

```
    // Подключение
    if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) ==
-1) {
        perror("connect");
        close(sock);
        return 1;
    }

    printf("Connected to %s:%d\n", host, port);

    // Формирование HTTP GET-запроса
    char request[1024];
    snprintf(request, sizeof(request),
             "GET / HTTP/1.1\r\n"
             "Host: %s\r\n"
             "User-Agent: SimpleHTTPClient/1.0\r\n"
             "Connection: close\r\n"
             "\r\n", host);

    // Отправка запроса
    send(sock, request, strlen(request), 0);
    printf("Request sent:\n%s\n", request);

    // Получение ответа
    char buffer[BUFFER_SIZE];
    int bytes;
    printf("\n=== Server Response ===\n");
    while ((bytes = recv(sock, buffer, BUFFER_SIZE - 1, 0)) > 0) {
        buffer[bytes] = '\0';
        printf("%s", buffer);
```

```
    }

    close(sock);
    return 0;
}
```

3.2. Скомпилируйте и выполните:

```
bash
gcc -Wall -g http_client.c -o http_client
./http_client example.com 80
./http_client google.com 80
```

4. **Обработка ошибок подключения**

4.1. Создайте файл robust_client.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <signal.h>
#include <setjmp.h>

#define TIMEOUT_SEC 5

static jmp_buf env;
static int sock = -1;
```

```
void timeout_handler(int sig) {
    longjmp(env, 1);
}

void set_timeout(int seconds) {
    signal(SIGALRM, timeout_handler);
    alarm(seconds);
}

int connect_with_timeout(const char *host, int port) {
    struct hostent *server = gethostbyname(host);
    if (server == NULL) {
        fprintf(stderr, "DNS error: %s\n", host);
        return -1;
    }

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("socket");
        return -1;
    }

    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    memcpy(&server_addr.sin_addr.s_addr, server->h_addr, server->h_length);

    set_timeout(TIMEOUT_SEC);
```

```
    if (setjmp(env) == 0) {
        if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr))
== -1) {
            alarm(0);
            perror("connect");
            close(sock);
            return -1;
        }
    } else {
        printf("Connection timeout after %d seconds\n", TIMEOUT_SEC);
        close(sock);
        return -1;
    }

    alarm(0);
    return sock;
}

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

    int port = atoi(argv[2]);
    int sockfd = connect_with_timeout(argv[1], port);

    if (sockfd == -1) {
        printf("Failed to connect to %s:%d\n", argv[1], port);
        return 1;
```

```
    }

    printf("Successfully connected to %s:%d\n", argv[1], port);

    char request[] = "GET / HTTP/1.0\r\nHost: example\r\n\r\n";
    send(sockfd, request, strlen(request), 0);

    char buffer[4096];
    int bytes = recv(sockfd, buffer, sizeof(buffer) - 1, 0);
    if (bytes > 0) {
        buffer[bytes] = '\0';
        printf("Received %d bytes\n", bytes);
        printf("First 500 bytes:\n%.500s\n", buffer);
    }

    close(sockfd);
    return 0;
}
```

4.2. Скомпилируйте и протестируйте:

```
bash
gcc -Wall -g robust_client.c -o robust_client
./robust_client google.com 80
./robust_client localhost 9999  # недоступный порт
```

5. **DNS-резолвинг с getaddrinfo (современный способ)**

5.1. Создайте файл modern_client.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#include <sys/socket.h>
#include <netdb.h>

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

    struct addrinfo hints, *res, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;    // IPv4 или IPv6
    hints.ai_socktype = SOCK_STREAM; // TCP

    int status = getaddrinfo(argv[1], argv[2], &hints, &res);
    if (status != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
        return 1;
    }

    int sock = -1;
    for (p = res; p != NULL; p = p->ai_next) {
        sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (sock == -1) continue;

        if (connect(sock, p->ai_addr, p->ai_addrlen) == 0) break;

        close(sock);
        sock = -1;
    }
```

```
    if (sock == -1) {
        fprintf(stderr, "Failed to connect to %s:%s\n", argv[1], argv[2]);
        freeaddrinfo(res);
        return 1;
    }

    printf("Connected to %s:%s\n", argv[1], argv[2]);

    // Отображение IP-адреса
    char ip[INET6_ADDRSTRLEN];
    struct sockaddr_in *addr = (struct sockaddr_in*)p->ai_addr;
    inet_ntop(p->ai_family, &addr->sin_addr, ip, sizeof(ip));
    printf("Using IP: %s\n", ip);

    // Простой запрос
    char request[] = "GET / HTTP/1.0\r\nHost: example\r\n\r\n";
    send(sock, request, strlen(request), 0);

    char buffer[4096];
    int bytes = recv(sock, buffer, sizeof(buffer) - 1, 0);
    if (bytes > 0) {
        buffer[bytes] = '\0';
        printf("\n=== Response (first 1000 bytes) ===\n%.1000s\n", buffer);
    }

    close(sock);
    freeaddrinfo(res);
    return 0;
}
```

5.2. Компиляция и запуск:

bash

```
gcc -Wall -g modern_client.c -o modern_client
./modern_client example.com 80
```

## Пример выполнения задания

**Задача примера:** разработать HTTP-клиент с поддержкой HTTPS (через TLS) и возможностью сохранения загруженного содержимого в файл. Клиент должен поддерживать GET-запросы, обрабатывать перенаправления (код 301/302) и корректно завершаться при ошибках.

**Шаг 1. Реализация HTTP-клиента с обработкой перенаправлений**

Файл http_downloader.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUFFER_SIZE 16384
#define MAX_REDIRECTS 5

typedef struct {
    char *data;
    size_t size;
} HttpResponse;
```

```
HttpResponse* create_response() {
    HttpResponse *resp = malloc(sizeof(HttpResponse));
    resp->data = malloc(1);
    resp->data[0] = '\0';
    resp->size = 0;
    return resp;
}

void append_response(HttpResponse *resp, const char *buffer, size_t len) {
    resp->data = realloc(resp->data, resp->size + len + 1);
    memcpy(resp->data + resp->size, buffer, len);
    resp->size += len;
    resp->data[resp->size] = '\0';
}

void free_response(HttpResponse *resp) {
    free(resp->data);
    free(resp);
}

int connect_to_host(const char *host, int port) {
    struct hostent *server = gethostbyname(host);
    if (server == NULL) return -1;

    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) return -1;

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
```

```
    memcpy(&addr.sin_addr.s_addr, server->h_addr, server->h_length);

    if (connect(sock, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
        close(sock);
        return -1;
    }

    return sock;
}

char* extract_header(const char *response, const char *header_name) {
    char search[256];
    snprintf(search, sizeof(search), "\r\n%s: ", header_name);

    char *start = strstr(response, search);
    if (!start) return NULL;

    start += strlen(search);
    char *end = strstr(start, "\r\n");
    if (!end) return NULL;

    size_t len = end - start;
    char *value = malloc(len + 1);
    strncpy(value, start, len);
    value[len] = '\0';

    return value;
}

int get_status_code(const char *response) {
```

```
    if (strncmp(response, "HTTP/", 5) != 0) return -1;

    const char *space = strchr(response, ' ');
    if (!space) return -1;

    return atoi(space + 1);
}

HttpResponse* send_request(const char *host, const char *path, int port) {
    int sock = connect_to_host(host, port);
    if (sock == -1) return NULL;

    char request[2048];
    snprintf(request, sizeof(request),
             "GET %s HTTP/1.1\r\n"
             "Host: %s\r\n"
             "User-Agent: HTTPDownloader/1.0\r\n"
             "Connection: close\r\n"
             "\r\n", path, host);

    send(sock, request, strlen(request), 0);

    HttpResponse *response = create_response();
    char buffer[BUFFER_SIZE];
    int bytes;

    while ((bytes = recv(sock, buffer, BUFFER_SIZE - 1, 0)) > 0) {
        append_response(response, buffer, bytes);
    }
```

```
    close(sock);
    return response;
}

void save_to_file(const char *filename, const char *content, size_t size) {
    FILE *f = fopen(filename, "wb");
    if (f) {
        fwrite(content, 1, size, f);
        fclose(f);
        printf("Saved to %s (%zu bytes)\n", filename, size);
    } else {
        perror("fopen");
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <url> [output_file]\n", argv[0]);
        fprintf(stderr, "Example: %s http://example.com/index.html\n",
argv[0]);
        return 1;
    }

    char *url = argv[1];
    char *output_file = (argc > 2) ? argv[2] : "downloaded.html";

    // Простой парсинг URL
    char *host_start = NULL;
    char *path_start = NULL;
    int port = 80;
```

```
if (strncmp(url, "http://", 7) == 0) {
    host_start = url + 7;
} else {
    host_start = url;
}

path_start = strchr(host_start, '/');
char host[256];
char path[1024] = "/";

if (path_start) {
    size_t host_len = path_start - host_start;
    strncpy(host, host_start, host_len);
    host[host_len] = '\0';
    strcpy(path, path_start);
} else {
    strcpy(host, host_start);
}

// Проверка порта в хосте
char *colon = strchr(host, ':');
if (colon) {
    *colon = '\0';
    port = atoi(colon + 1);
}

printf("Host: %s, Port: %d, Path: %s\n", host, port, path);

int redirect_count = 0;
```

```
char current_host[256];
char current_path[1024];
strcpy(current_host, host);
strcpy(current_path, path);

while (redirect_count < MAX_REDIRECTS) {
    printf("Requesting: http://%s%s\n", current_host, current_path);

    HttpResponse *resp = send_request(current_host, current_path, port);
    if (!resp) {
        fprintf(stderr, "Failed to connect\n");
        return 1;
    }

    int status = get_status_code(resp->data);
    printf("HTTP Status: %d\n", status);

    if (status == 200) {
        // Успешный ответ
        char *body = strstr(resp->data, "\r\n\r\n");
        if (body) {
            body += 4;
            size_t body_size = resp->size - (body - resp->data);
            save_to_file(output_file, body, body_size);
        }
        free_response(resp);
        break;
    } else if (status == 301 || status == 302) {
        // Перенаправление
        char *location = extract_header(resp->data, "Location");
```

```
    if (location) {
        printf("Redirecting to: %s\n", location);

        // Парсинг нового URL
        if (strncmp(location, "http://", 7) == 0) {
            // Полный URL
            char *new_host_start = location + 7;
            char *new_path_start = strchr(new_host_start, '/');
            if (new_path_start) {
                size_t host_len = new_path_start - new_host_start;
                strncpy(current_host, new_host_start, host_len);
                current_host[host_len] = '\0';
                strcpy(current_path, new_path_start);
            } else {
                strcpy(current_host, new_host_start);
                strcpy(current_path, "/");
            }
        } else {
            // Относительный URL
            strcpy(current_path, location);
        }

        free(location);
        redirect_count++;
    } else {
        fprintf(stderr, "Redirect without Location header\n");
        free_response(resp);
        break;
    }
} else {
```

```
            printf("Unhandled status code: %d\n", status);
            free_response(resp);
            break;
        }

        free_response(resp);
    }

    if (redirect_count >= MAX_REDIRECTS) {
        fprintf(stderr, "Too many redirects\n");
    }

    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

```
gcc -Wall -g http_downloader.c -o http_downloader
./http_downloader http://example.com/index.html example.html
./http_downloader http://httpbin.org/redirect/2
./http_downloader http://google.com
```

**Шаг 3. Пример вывода**

```
Host: example.com, Port: 80, Path: /index.html
Requesting: http://example.com/index.html
HTTP Status: 200
Saved to example.html (1256 bytes)

Host: httpbin.org, Port: 80, Path: /redirect/2
Requesting: http://httpbin.org/redirect/2
HTTP Status: 302
```

Redirecting to: /relative-redirect/1

Requesting: http://httpbin.org/relative-redirect/1

HTTP Status: 302

Redirecting to: /get

Requesting: http://httpbin.org/get

HTTP Status: 200

Saved to downloaded.html (378 bytes)

**Вывод**

Программа демонстрирует:

- Установку TCP-соединения с HTTP-сервером.
- Формирование и отправку HTTP GET-запроса.
- Получение и разбор HTTP-ответа (заголовки, тело).
- Обработку перенаправлений (301/302).
- Сохранение загруженного содержимого в файл.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать TCP-клиент для протокола HTTP с поддержкой метода POST. Клиент должен отправлять данные (например, форму) на сервер и выводить ответ. |
| 2 | Создать программу для проверки доступности порта (port scanner). Клиент пытается подключиться к заданному диапазону портов и выводит открытые. |
| 3 | Реализовать клиент для протокола WHOIS (порт 43). Клиент отправляет доменное имя и выводит информацию о регистрации. |
| 4 | Написать программу для загрузки файла по HTTP с поддержкой докачки (HTTP Range). Клиент должен сохранять файл и при прерывании возобновлять загрузку. |
| 5 | Создать клиент для простого протокола «эхо» (echo client). Клиент отправляет сообщения серверу и получает их обратно. Добавить поддержку таймаута и повторных попыток. |
| 6 | Реализовать клиент для протокола SMTP (порт 25). Клиент должен подключаться к почтовому серверу и отправлять простое текстовое письмо. |

**Обратить внимание:**

- Всегда проверять возвращаемые значения системных вызовов.
- Использовать htons() для преобразования порта.
- Для доменных имён использовать gethostbyname() или getaddrinfo().
- Обрабатывать частичную отправку/получение данных (циклы send/recv).

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание простого TCP-клиента;
   – реализация HTTP-клиента;
   – обработка ошибок и таймаутов;
   – DNS-резолвинг через getaddrinfo;
   – (по варианту) реализация по индивидуальному заданию.
4. Скриншоты (обязательно):
   – вывод http_client с получением веб-страницы;
   – работа robust_client с успешным и ошибочным подключением;
   – вывод modern_client;
   – (по варианту) работа программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Какие системные вызовы используются для создания TCP-клиента?
2. Что такое сокет? Какие параметры передаются в socket() для TCP?
3. Как преобразовать строку с IP-адресом в двоичный формат?
4. Как получить IP-адрес по доменному имени?
5. Что делает функция connect()? Какие ошибки могут возникнуть?
6. В чём разница между send()/recv() и write()/read()?
7. Как установить таймаут на подключение?
8. Какую структуру имеет HTTP GET-запрос?
9. Что означают коды ответа HTTP: 200, 301, 302, 404, 500?
10. Почему при получении данных из сокета нужно использовать цикл?
11. Что такое getaddrinfo() и в чём её преимущество перед gethostbyname()?
12. Как корректно завершить TCP-соединение?

# Лабораторная работа №20

# TCP-сервер (итеративный и многопоточный)

**Цель работы:** освоить создание серверных приложений для обработки клиентских запросов по протоколу TCP, разработать итеративную и многопоточную версии эхо-сервера, научиться обрабатывать несколько одновременных подключений.

**Задачи:**

– Изучить основные системные вызовы для TCP-сервера: socket, bind, listen, accept.

– Разработать итеративную версию эхо-сервера (обработка клиентов последовательно).

– Реализовать многопоточную версию сервера с созданием отдельного потока для каждого клиента.

– Изучить проблемы синхронизации при доступе к общим ресурсам в многопоточной версии.

– Научиться корректно завершать сервер и освобождать ресурсы.

– Сравнить производительность итеративной и многопоточной версий.

## Теоретические сведения

1. **Системные вызовы для TCP-сервера**

| Функция | Назначение |
|---|---|
| socket(int domain, int type, int protocol) | Создание сокета |
| bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen) | Привязка сокета к адресу и порту |
| listen(int sockfd, int backlog) | Перевод сокета в режим ожидания подключений |
| accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen) | Принятие входящего подключения |

| Функция | Назначение |
|---|---|
| send/recv или write/read | Обмен данными с клиентом |
| close(int fd) | Закрытие сокета |

2. **Итеративный сервер**

   Итеративный сервер обрабатывает клиентов последовательно: после принятия подключения он обслуживает клиента полностью (читает данные, отправляет ответ) и только после закрытия соединения принимает следующего клиента.

   **Недостаток:** если один клиент работает долго, остальные ожидают.
3. **Многопоточный сервер**

   При подключении клиента сервер создаёт отдельный поток (или процесс), который обрабатывает этого клиента. Основной поток продолжает принимать новых клиентов.

   **Преимущество:** параллельная обработка нескольких клиентов.
4. **Структура многопоточного сервера**

   c

   ```
   while (1) {
      client_fd = accept(server_fd, ...);
      pthread_create(&thread, NULL, client_handler, &client_fd);
      pthread_detach(thread);  // автоматическое освобождение ресурсов
   }
   ```
5. **Проблемы многопоточного сервера**
   - Гонки данных при доступе к общим ресурсам (требуются мьютексы).
   - Ограничения на количество потоков (можно использовать пул потоков).
   - Необходимость корректной передачи файлового дескриптора в поток.

6. **Обработка сигналов в сервере**

   Для корректного завершения сервера (например, по SIGINT или SIGTERM) необходимо:

   - Установить флаг завершения.
   - Закрыть слушающий сокет.
   - Дождаться завершения всех рабочих потоков.

7. **Сравнение подходов**

| **Характеристика** | **Итеративный** | **Многопоточный** |
|---|---|---|
| Сложность | Низкая | Средняя |
| Параллелизм | Нет | Да |
| Использование ресурсов | Минимальное | Высокое (на каждый клиент) |
| Подходит для | Небольшое число клиентов, быстрые операции | Много клиентов, длительные операции |

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab20

   cd ~/linux_labs/lab20

2. **Итеративный эхо-сервер**

   2.1. Создайте файл iterative_echo_server.c:

   c

   #include <stdio.h>

   #include <stdlib.h>

   #include <string.h>

   #include <unistd.h>

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);
    char buffer[BUFFER_SIZE];

    // Создание сокета
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket");
        return 1;
    }

    // Настройка адреса сервера
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Привязка
```

```
        if (bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr))
== -1) {
            perror("bind");
            return 1;
        }

        // Прослушивание
        if (listen(server_fd, 5) == -1) {
            perror("listen");
            return 1;
        }

        printf("Iterative echo server listening on port %d\n", PORT);

        while (1) {
            // Принятие подключения
            client_fd = accept(server_fd, (struct sockaddr*)&client_addr,
&client_len);
            if (client_fd == -1) {
                perror("accept");
                continue;
            }

            printf("Client connected: %s:%d\n",
                   inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

            // Обработка клиента
            ssize_t bytes;
            while ((bytes = recv(client_fd, buffer, BUFFER_SIZE - 1, 0)) > 0) {
                buffer[bytes] = '\0';
```

```
            printf("Received: %s", buffer);
            send(client_fd, buffer, bytes, 0);
        }

        close(client_fd);
        printf("Client disconnected\n");
    }

    close(server_fd);
    return 0;
}
```

2.2. Скомпилируйте и запустите:

```
bash
gcc -Wall -g iterative_echo_server.c -o iterative_server
./iterative_server
```

2.3. В другом терминале подключитесь с помощью telnet или nc:

```
bash
telnet localhost 8080
# или
nc localhost 8080
```

3. **Многопоточный эхо-сервер**

3.1. Создайте файл threaded_echo_server.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>

#define PORT 8081
#define BUFFER_SIZE 1024

pthread_mutex_t log_mutex = PTHREAD_MUTEX_INITIALIZER;

void log_message(const char *msg) {
    pthread_mutex_lock(&log_mutex);
    printf("%s\n", msg);
    fflush(stdout);
    pthread_mutex_unlock(&log_mutex);
}

void *handle_client(void *arg) {
    int client_fd = *(int*)arg;
    free(arg);

    struct sockaddr_in addr;
    socklen_t addr_len = sizeof(addr);
    getpeername(client_fd, (struct sockaddr*)&addr, &addr_len);

    char log_buf[256];
    snprintf(log_buf, sizeof(log_buf), "Thread %lu: client connected from %s:%d",
             pthread_self(), inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
    log_message(log_buf);

    char buffer[BUFFER_SIZE];
    ssize_t bytes;
```

```
    while ((bytes = recv(client_fd, buffer, BUFFER_SIZE - 1, 0)) > 0) {
        buffer[bytes] = '\0';

        snprintf(log_buf, sizeof(log_buf), "Thread %lu: received %ld bytes",
                 pthread_self(), bytes);
        log_message(log_buf);

        send(client_fd, buffer, bytes, 0);
    }

    close(client_fd);
    snprintf(log_buf, sizeof(log_buf), "Thread %lu: client disconnected",
pthread_self());
    log_message(log_buf);

    return NULL;
}

int main() {
    int server_fd;
    struct sockaddr_in server_addr;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("socket");
        return 1;
    }

    int opt = 1;
```

```
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
        perror("bind");
        return 1;
    }

    if (listen(server_fd, 10) == -1) {
        perror("listen");
        return 1;
    }

    printf("Threaded echo server listening on port %d\n", PORT);
    printf("Using threads for each client\n");

    while (1) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);

        int *client_fd = malloc(sizeof(int));
        *client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &client_len);

        if (*client_fd == -1) {
```

```
            perror("accept");
            free(client_fd);
            continue;
        }

        pthread_t thread;
        pthread_create(&thread, NULL, handle_client, client_fd);
        pthread_detach(thread);
    }

    close(server_fd);
    return 0;
}
```

3.2. Скомпилируйте и запустите:

```
bash
gcc -Wall -g threaded_echo_server.c -o threaded_server -pthread
./threaded_server
```

3.3. Подключитесь несколькими клиентами одновременно:

```
bash
# Терминал 2:
nc localhost 8081

# Терминал 3:
nc localhost 8081

# Терминал 4:
nc localhost 8081
```

4. **Эхо-сервер с пулом потоков**

4.1. Создайте файл thread_pool_server.c (упрощённая версия с фиксированным числом потоков):

c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 8082
#define BUFFER_SIZE 1024
#define THREAD_POOL_SIZE 4

typedef struct {
    int client_fd;
    struct sockaddr_in addr;
} Task;

Task task_queue[100];
int task_count = 0;
int head = 0, tail = 0;
pthread_mutex_t queue_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t queue_cond = PTHREAD_COND_INITIALIZER;
int server_running = 1;

void *worker_thread(void *arg) {
    int id = *(int*)arg;
    free(arg);
```

```
        while (server_running) {
            pthread_mutex_lock(&queue_mutex);
            while (task_count == 0 && server_running) {
                pthread_cond_wait(&queue_cond, &queue_mutex);
            }

            if (!server_running && task_count == 0) {
                pthread_mutex_unlock(&queue_mutex);
                break;
            }

            Task task = task_queue[head];
            head = (head + 1) % 100;
            task_count--;

            pthread_mutex_unlock(&queue_mutex);

            // Обработка клиента
            char buffer[BUFFER_SIZE];
            ssize_t bytes;
            printf("Worker %d: handling client from %s:%d\n",
                   id, inet_ntoa(task.addr.sin_addr), ntohs(task.addr.sin_port));

            while ((bytes = recv(task.client_fd, buffer, BUFFER_SIZE - 1, 0)) > 0)
{
                send(task.client_fd, buffer, bytes, 0);
            }

            close(task.client_fd);
            printf("Worker %d: client disconnected\n", id);
```

```
        }

        return NULL;
    }

    int main() {
        int server_fd;
        struct sockaddr_in server_addr;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        int opt = 1;
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt));

        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = INADDR_ANY;
        server_addr.sin_port = htons(PORT);

        bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));
        listen(server_fd, 10);

        printf("Thread pool server listening on port %d\n", PORT);
        printf("Pool size: %d\n", THREAD_POOL_SIZE);

        // Создание пула потоков
        pthread_t threads[THREAD_POOL_SIZE];
        for (int i = 0; i < THREAD_POOL_SIZE; i++) {
            int *id = malloc(sizeof(int));
            *id = i;
            pthread_create(&threads[i], NULL, worker_thread, id);
```

```
        }

        // Приём клиентов
        while (1) {
            struct sockaddr_in client_addr;
            socklen_t client_len = sizeof(client_addr);
            int client_fd = accept(server_fd, (struct sockaddr*)&client_addr,
&client_len);

            if (client_fd == -1) continue;

            pthread_mutex_lock(&queue_mutex);
            while (task_count >= 100) {
                pthread_mutex_unlock(&queue_mutex);
                usleep(10000);
                pthread_mutex_lock(&queue_mutex);
            }

            task_queue[tail].client_fd = client_fd;
            task_queue[tail].addr = client_addr;
            tail = (tail + 1) % 100;
            task_count++;

            pthread_cond_signal(&queue_cond);
            pthread_mutex_unlock(&queue_mutex);
        }

        close(server_fd);
        return 0;
    }
```

## Пример выполнения задания

**Задача примера:** разработать TCP-сервер для обработки клиентских запросов на вычисление математических выражений. Клиент отправляет выражение (например, "2 + 3"), сервер вычисляет результат и возвращает его. Сервер должен поддерживать несколько одновременных клиентов с помощью многопоточности.

**Шаг 1. Реализация сервера**

Файл calc_server.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 8083
#define BUFFER_SIZE 256

pthread_mutex_t log_mutex = PTHREAD_MUTEX_INITIALIZER;

void log_message(const char *msg) {
    pthread_mutex_lock(&log_mutex);
    printf("[%ld] %s\n", pthread_self(), msg);
    fflush(stdout);
    pthread_mutex_unlock(&log_mutex);
}
```

```
int calculate(const char *expr, char *result) {
    int a, b;
    char op;
    if (sscanf(expr, "%d %c %d", &a, &op, &b) != 3) {
        sprintf(result, "ERROR: Invalid format. Use: number operator number");
        return -1;
    }

    switch (op) {
        case '+': sprintf(result, "%d", a + b); break;
        case '-': sprintf(result, "%d", a - b); break;
        case '*': sprintf(result, "%d", a * b); break;
        case '/':
            if (b == 0) {
                sprintf(result, "ERROR: Division by zero");
                return -1;
            }
            sprintf(result, "%.2f", (float)a / b);
            break;
        default:
            sprintf(result, "ERROR: Unknown operator '%c'", op);
            return -1;
    }
    return 0;
}

void *handle_client(void *arg) {
    int client_fd = *(int*)arg;
    free(arg);
```

```
        struct sockaddr_in addr;
        socklen_t addr_len = sizeof(addr);
        getpeername(client_fd, (struct sockaddr*)&addr, &addr_len);

        char log_buf[256];
        snprintf(log_buf, sizeof(log_buf), "Client connected from %s:%d",
                 inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
        log_message(log_buf);

        char buffer[BUFFER_SIZE];
        char result[BUFFER_SIZE];
        ssize_t bytes;

        // Отправка приветствия
        char *welcome = "Math Calculator Server\nCommands: a + b, a - b, a * b,
a / b\nType 'quit' to exit\n\n";
        send(client_fd, welcome, strlen(welcome), 0);

        while ((bytes = recv(client_fd, buffer, BUFFER_SIZE - 1, 0)) > 0) {
            buffer[bytes] = '\0';
            // Удаление символа новой строки
            buffer[strcspn(buffer, "\n")] = '\0';

            if (strcmp(buffer, "quit") == 0) {
                send(client_fd, "Goodbye!\n", 9, 0);
                break;
            }

            snprintf(log_buf, sizeof(log_buf), "Received: %s", buffer);
            log_message(log_buf);
```

```
            if (calculate(buffer, result) == 0) {
                snprintf(buffer, sizeof(buffer), "%s\n", result);
            } else {
                snprintf(buffer, sizeof(buffer), "%s\n", result);
            }

            send(client_fd, buffer, strlen(buffer), 0);
        }

        close(client_fd);
        log_message("Client disconnected");
        return NULL;
    }

    int main() {
        int server_fd;
        struct sockaddr_in server_addr;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd == -1) {
            perror("socket");
            return 1;
        }

        int opt = 1;
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt));

        server_addr.sin_family = AF_INET;
```

```
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr))
== -1) {
        perror("bind");
        return 1;
    }

    if (listen(server_fd, 10) == -1) {
        perror("listen");
        return 1;
    }

    printf("=== Math Calculator Server ===\n");
    printf("Listening on port %d\n", PORT);
    printf("Multi-threaded mode\n\n");

    while (1) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);

        int *client_fd = malloc(sizeof(int));
        *client_fd = accept(server_fd, (struct sockaddr*)&client_addr,
&client_len);

        if (*client_fd == -1) {
            perror("accept");
            free(client_fd);
            continue;
```

```
        }

        pthread_t thread;
        pthread_create(&thread, NULL, handle_client, client_fd);
        pthread_detach(thread);
    }

    close(server_fd);
    return 0;
}
```

**Шаг 2. Компиляция и запуск**

bash

```
gcc -Wall -g calc_server.c -o calc_server -pthread
./calc_server
```

**Шаг 3. Подключение клиентов**

bash

```
# Терминал 2:
nc localhost 8083
# Вводим:
10 + 5
20 * 3
100 / 4
quit

# Терминал 3 (одновременно):
nc localhost 8083
# Вводим:
7 - 2
9 / 0
quit
```

**Шаг 4. Пример вывода**

**Сервер:**

=== Math Calculator Server ===
Listening on port 8083
Multi-threaded mode

[140123456789] Client connected from 127.0.0.1:54321
[140123456789] Received: 10 + 5
[140123456789] Received: 20 * 3
[140123456789] Received: 100 / 4
[140123456789] Received: quit
[140123456789] Client disconnected
[140123456790] Client connected from 127.0.0.1:54322
[140123456790] Received: 7 - 2
[140123456790] Received: 9 / 0
[140123456790] Received: quit
[140123456790] Client disconnected

**Клиент 1:**

Math Calculator Server
Commands: a + b, a - b, a * b, a / b
Type 'quit' to exit

10 + 5
15
20 * 3
60
100 / 4
25.00

quit

Goodbye!

**Вывод**

Программа демонстрирует:

- Создание многопоточного TCP-сервера с отдельным потоком для каждого клиента.
- Обработку математических выражений с поддержкой четырёх операций.
- Параллельную обработку нескольких клиентов (видно по разным ID потоков).
- Корректное завершение соединения по команде quit.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать итеративный сервер для перевода текста в верхний регистр. Клиент отправляет строку, сервер возвращает её в верхнем регистре. |
| 2 | Создать многопоточный сервер для хранения пар «ключ-значение» (простая база данных в памяти). Клиенты могут отправлять команды SET, GET, DELETE. |
| 3 | Реализовать итеративный и многопоточный варианты сервера для вычисления чисел Фибоначчи. Сравнить производительность при нескольких одновременных запросах. |
| 4 | Создать сервер для игры «Угадай число». Сервер загадывает случайное число, клиент пытается угадать. Сервер отвечает «больше», «меньше», «угадал». |
| 5 | Реализовать многопоточный сервер для проверки простоты числа. Клиент отправляет число, сервер возвращает, простое оно или нет. |
| 6 | Создать сервер для конвертации валют (фиксированный курс). Клиент отправляет сумму и код валюты (USD, EUR), сервер возвращает сумму в рублях. |

**Обратить внимание:**

– В многопоточной версии передавать клиентский сокет в поток через динамическую память.

– Использовать pthread_detach для автоматической очистки ресурсов.

– Для синхронизации доступа к общим ресурсам (например, лог-файл) использовать мьютексы.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание итеративного эхо-сервера;
   – создание многопоточного эхо-сервера;
   – тестирование с несколькими клиентами;
   – (по варианту) реализация по индивидуальному заданию.
4. Скриншоты (обязательно):
   – работа итеративного сервера с клиентом;
   – работа многопоточного сервера с несколькими одновременными клиентами;
   – вывод сервера с разными ID потоков;
   – (по варианту) работа программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Какие системные вызовы используются для создания TCP-сервера?
2. Для чего нужен вызов bind()? Что произойдёт, если не вызвать bind()?
3. Что делает вызов listen()? Что такое параметр backlog?
4. Как работает вызов accept()? Какие данные он возвращает?
5. В чём отличие итеративного сервера от многопоточного?
6. Какие преимущества и недостатки у многопоточного сервера?
7. Как правильно передать файловый дескриптор в поток?
8. Для чего используется pthread_detach()?
9. Как организовать корректное завершение многопоточного сервера?
10. Почему в многопоточном сервере нужны мьютексы для доступа к общим ресурсам?
11. Что такое SO_REUSEADDR? Когда его используют?
12. Как проверить работу сервера с несколькими одновременными клиентами?

# Лабораторная работа №21

# UDP-сокеты

**Цель работы:** освоить работу с UDP-сокетами для обмена дейтаграммами, разработать UDP-клиент и сервер, реализовать простой сетевой чат без установления соединения, сравнить поведение TCP и UDP при потере пакетов.

**Задачи:**

– Изучить основные системные вызовы для UDP-сокетов: socket, sendto, recvfrom, bind.

– Разработать UDP-сервер, принимающий сообщения от клиентов и отправляющий ответы.

– Разработать UDP-клиент, отправляющий сообщения серверу.

– Реализовать простой сетевой чат с возможностью обмена сообщениями.

– Изучить отличия UDP от TCP (ненадёжность, отсутствие соединения, дейтаграммы).

– Экспериментально сравнить поведение протоколов при потере пакетов.

## Теоретические сведения

1. **Протокол UDP (User Datagram Protocol)**

   UDP – это простой, ненадёжный, протокол транспортного уровня без установления соединения. Данные передаются в виде дейтаграмм (сообщений фиксированного размера).

   **Характеристики:**

   - Нет гарантии доставки.
   - Нет контроля порядка следования пакетов.
   - Нет контроля перегрузки.

- Меньшие накладные расходы (нет установления соединения).
- Быстрее TCP для приложений, где допустима потеря данных (видео, аудио, голос).

2. **Основные системные вызовы для UDP**

| **Функция** | **Назначение** |
|---|---|
| socket(AF_INET, SOCK_DGRAM, 0) | Создание UDP-сокета |
| bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen) | Привязка сокета к порту (для сервера) |
| sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen) | Отправка дейтаграммы |
| recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen) | Приём дейтаграммы |
| close(int fd) | Закрытие сокета |

3. **Отличия UDP от TCP**

| **Характеристика** | **TCP** | **UDP** |
|---|---|---|
| Соединение | Ориентирован на соединение | Без соединения |
| Надёжность | Гарантированная доставка | Нет гарантии |
| Порядок данных | Сохраняется | Может нарушаться |
| Границы сообщений | Потоковый (нет границ) | Дейтаграммный (границы сохраняются) |
| Контроль перегрузки | Есть | Нет |
| Скорость | Ниже | Выше |

4. **Структура UDP-сервера**

   c

   ```
   int sock = socket(AF_INET, SOCK_DGRAM, 0);
   bind(sock, ...);
   while (1) {
      recvfrom(sock, buffer, sizeof(buffer), 0, &client_addr, &len);
      // обработка запроса
      sendto(sock, response, strlen(response), 0, &client_addr, len);
   }
   ```

5. **Структура UDP-клиента**

   c

   ```
   int sock = socket(AF_INET, SOCK_DGRAM, 0);
   // bind() не обязателен (ядро назначит порт автоматически)
   sendto(sock, message, strlen(message), 0, &server_addr, sizeof(server_addr));
   recvfrom(sock, buffer, sizeof(buffer), 0, NULL, NULL);
   ```

6. **Особенности UDP**
   - Размер дейтаграммы ограничен (обычно до 64 КБ).
   - recvfrom читает всю дейтаграмму целиком (нельзя прочитать часть).
   - Если буфер меньше дейтаграммы, лишние данные отбрасываются.
   - Несколько вызовов sendto могут быть приняты одним recvfrom? Нет, каждая дейтаграмма принимается отдельно.
7. **Потеря пакетов в UDP**

   UDP не требует подтверждения получения. Пакет может быть потерян в сети без уведомления отправителя. Для надёжности приложения должны реализовывать свои механизмы (таймауты, подтверждения, повторные отправки).

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab21

   cd ~/linux_labs/lab21

2. **Простейший UDP-сервер и клиент (эхо)**

   2.1. Создайте файл udp_echo_server.c:

   c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9090
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);
    char buffer[BUFFER_SIZE];

    // Создание UDP-сокета
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) {
        perror("socket");
```

```
        return 1;
    }

    // Настройка адреса сервера
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Привязка
    if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -
1) {
        perror("bind");
        return 1;
    }

    printf("UDP Echo Server listening on port %d\n", PORT);

    while (1) {
        memset(buffer, 0, BUFFER_SIZE);
        ssize_t bytes = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0,
                                 (struct sockaddr*)&client_addr, &client_len);
        if (bytes == -1) {
            perror("recvfrom");
            continue;
        }

        printf("Received from %s:%d: %s\n",
               inet_ntoa(client_addr.sin_addr),      ntohs(client_addr.sin_port),
buffer);
```

```
        // Отправка обратно (эхо)
        sendto(sockfd, buffer, bytes, 0,
               (struct sockaddr*)&client_addr, client_len);
    }

    close(sockfd);
    return 0;
}
```

2.2. Создайте файл udp_echo_client.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9090
#define BUFFER_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <server_ip>\n", argv[0]);
        return 1;
    }

    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];
```

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd == -1) {
    perror("socket");
    return 1;
}

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
server_addr.sin_addr.s_addr = inet_addr(argv[1]);

printf("UDP Echo Client. Type messages, 'quit' to exit\n");

while (1) {
    printf("Enter message: ");
    fflush(stdout);
    fgets(buffer, BUFFER_SIZE, stdin);
    buffer[strcspn(buffer, "\n")] = '\0';

    if (strcmp(buffer, "quit") == 0) break;

    sendto(sockfd, buffer, strlen(buffer), 0,
           (struct sockaddr*)&server_addr, sizeof(server_addr));

    memset(buffer, 0, BUFFER_SIZE);
    socklen_t addr_len = sizeof(server_addr);
    ssize_t bytes = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0,
                             (struct sockaddr*)&server_addr, &addr_len);
    if (bytes > 0) {
        printf("Server echo: %s\n", buffer);
```

```
        }
    }

    close(sockfd);
    return 0;
}
```

2.3. Скомпилируйте и запустите:

```
bash
gcc -Wall -g udp_echo_server.c -o udp_server
gcc -Wall -g udp_echo_client.c -o udp_client

# Терминал 1:
./udp_server

# Терминал 2:
./udp_client 127.0.0.1
```

3. **UDP-чат без установления соединения**

3.1. Создайте файл udp_chat_server.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9091
#define BUFFER_SIZE 1024
#define MAX_CLIENTS 10
```

```
typedef struct {
    struct sockaddr_in addr;
    char name[32];
} Client;

Client clients[MAX_CLIENTS];
int client_count = 0;

void broadcast_message(const char *msg, const struct sockaddr_in *sender_addr, int sockfd) {
    for (int i = 0; i < client_count; i++) {
        if (memcmp(&clients[i].addr, sender_addr, sizeof(struct sockaddr_in)) != 0) {
            sendto(sockfd, msg, strlen(msg), 0,
                (struct sockaddr*)&clients[i].addr, sizeof(clients[i].addr));
        }
    }
}

int find_client(const struct sockaddr_in *addr) {
    for (int i = 0; i < client_count; i++) {
        if (clients[i].addr.sin_addr.s_addr == addr->sin_addr.s_addr &&
            clients[i].addr.sin_port == addr->sin_port) {
            return i;
        }
    }
    return -1;
}
```

```
int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);
    char buffer[BUFFER_SIZE];

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) { perror("socket"); return 1; }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -
1) {
        perror("bind"); return 1;
    }

    printf("UDP Chat Server running on port %d\n", PORT);

    while (1) {
        memset(buffer, 0, BUFFER_SIZE);
        ssize_t bytes = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0,
                                 (struct sockaddr*)&client_addr, &client_len);
        if (bytes == -1) { perror("recvfrom"); continue; }

        buffer[bytes] = '\0';

        // Формат сообщения: /register <name> или <name>: <message>
        int client_idx = find_client(&client_addr);
```

```
            if (strncmp(buffer, "/register ", 10) == 0) {
                // Регистрация нового клиента
                if (client_idx == -1 && client_count < MAX_CLIENTS) {
                    clients[client_count].addr = client_addr;
                    strcpy(clients[client_count].name, buffer + 10);
                    char welcome[256];
                    snprintf(welcome, sizeof(welcome), "Server: %s joined the chat\n",
buffer + 10);
                    broadcast_message(welcome, &client_addr, sockfd);
                    client_count++;
                    printf("%s (%s:%d) registered\n", buffer + 10,
                           inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
                }
            } else if (client_idx != -1) {
                // Отправка сообщения всем
                char msg[BUFFER_SIZE + 50];
                snprintf(msg, sizeof(msg), "%s: %s", clients[client_idx].name,
buffer);
                broadcast_message(msg, &client_addr, sockfd);
                printf("%s\n", msg);
            } else {
                // Неизвестный клиент
                char *error = "Server: Please register first: /register <name>\n";
                sendto(sockfd, error, strlen(error), 0,
                       (struct sockaddr*)&client_addr, client_len);
            }
        }

        close(sockfd);
```

```
    return 0;
}
```

3.2. Создайте файл udp_chat_client.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9091
#define BUFFER_SIZE 1024

int sockfd;
struct sockaddr_in server_addr;

void *receive_messages(void *arg) {
    char buffer[BUFFER_SIZE];
    socklen_t addr_len = sizeof(server_addr);

    while (1) {
        memset(buffer, 0, BUFFER_SIZE);
        ssize_t bytes = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0,
                                 (struct sockaddr*)&server_addr, &addr_len);
        if (bytes > 0) {
            printf("\r%s\n> ", buffer);
            fflush(stdout);
```

```
        }
    }
    return NULL;
}

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

    const char *server_ip = argv[1];
    const char *username = argv[2];

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) { perror("socket"); return 1; }

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr(server_ip);

    // Регистрация
    char register_msg[BUFFER_SIZE];
    snprintf(register_msg, sizeof(register_msg), "/register %s", username);
    sendto(sockfd, register_msg, strlen(register_msg), 0,
           (struct sockaddr*)&server_addr, sizeof(server_addr));

    printf("Connected to chat server as %s\n", username);
    printf("Type messages, 'quit' to exit\n\n");
```

```
    // Поток для приёма сообщений
    pthread_t recv_thread;
    pthread_create(&recv_thread, NULL, receive_messages, NULL);
    pthread_detach(recv_thread);

    char message[BUFFER_SIZE];
    while (1) {
        printf("> ");
        fflush(stdout);
        fgets(message, BUFFER_SIZE, stdin);
        message[strcspn(message, "\n")] = '\0';

        if (strcmp(message, "quit") == 0) break;

        sendto(sockfd, message, strlen(message), 0,
               (struct sockaddr*)&server_addr, sizeof(server_addr));
    }

    close(sockfd);
    return 0;
}
```

3.3. Скомпилируйте и запустите:

```
bash
gcc -Wall -g udp_chat_server.c -o chat_server
gcc -Wall -g udp_chat_client.c -o chat_client -pthread

# Терминал 1:
./chat_server

# Терминал 2:
```

./chat_client 127.0.0.1 Alice

*# Терминал 3:*

./chat_client 127.0.0.1 Bob

4. **Сравнение TCP и UDP при потере пакетов**

4.1. Создайте файл udp_loss_test.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9092
#define NUM_PACKETS 100
#define TIMEOUT_SEC 2

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <server|client>\n", argv[0]);
        return 1;
    }

    if (strcmp(argv[1], "server") == 0) {
        int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
        struct sockaddr_in addr = {AF_INET, htons(PORT), INADDR_ANY};
        bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
```

```
            printf("UDP Server: listening on port %d\n", PORT);

            int received = 0;
            struct sockaddr_in client_addr;
            socklen_t len = sizeof(client_addr);
            char buffer[256];

            for (int i = 0; i < NUM_PACKETS; i++) {
               recvfrom(sockfd, buffer, sizeof(buffer), 0,
                        (struct sockaddr*)&client_addr, &len);
               received++;
               printf("Received packet %d\n", i + 1);
            }

            printf("\n=== UDP Results ===\n");
            printf("Packets sent: %d\n", NUM_PACKETS);
            printf("Packets received: %d\n", received);
            printf("Loss rate: %.1f%%\n", (1.0 - (float)received / NUM_PACKETS)
* 100);

            close(sockfd);
         } else {
            int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
            struct sockaddr_in addr = {AF_INET, htons(PORT)};
            inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);

            printf("UDP Client: sending %d packets\n", NUM_PACKETS);

            for (int i = 0; i < NUM_PACKETS; i++) {
               char msg[32];
```

```
            snprintf(msg, sizeof(msg), "Packet %d", i + 1);
            sendto(sockfd, msg, strlen(msg), 0,
                  (struct sockaddr*)&addr, sizeof(addr));
            usleep(10000);
        }

        printf("All packets sent.\n");
        close(sockfd);
    }

    return 0;
}
```

4.2. Запустите тест потери пакетов:

```
bash
gcc -Wall -g udp_loss_test.c -o udp_loss

# Терминал 1:
./udp_loss server

# Терминал 2:
./udp_loss client
```

## Пример выполнения задания

**Задача примера:** разработать систему для передачи файлов по протоколу UDP с простым механизмом подтверждения. Клиент отправляет файл серверу блоками фиксированного размера, сервер подтверждает получение каждого блока, при потере клиент повторяет отправку.

**Шаг 1. Реализация сервера**

Файл udp_file_server.c:

```
c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9093
#define BLOCK_SIZE 1024
#define TIMEOUT_SEC 3

typedef struct {
    int block_num;
    int total_blocks;
    char data[BLOCK_SIZE];
} Packet;

int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) { perror("socket"); return 1; }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);
```

```
    if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -
1) {
        perror("bind"); return 1;
    }

    printf("UDP File Server listening on port %d\n", PORT);
    printf("Waiting for file transfer...\n");

    // Установка таймаута
    struct timeval tv = {TIMEOUT_SEC, 0};
    setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

    FILE *file = fopen("received_file", "wb");
    if (!file) { perror("fopen"); return 1; }

    Packet packet;
    int expected_block = 0;
    int total_blocks = -1;
    int received_blocks = 0;

    while (1) {
        memset(&packet, 0, sizeof(packet));
        ssize_t bytes = recvfrom(sockfd, &packet, sizeof(packet), 0,
                                 (struct sockaddr*)&client_addr, &client_len);

        if (bytes == -1) {
            printf("Timeout waiting for block %d\n", expected_block);
            break;
        }
```

```
    if (total_blocks == -1) {
        total_blocks = packet.total_blocks;
        printf("Receiving file: %d blocks\n", total_blocks);
    }

    if (packet.block_num == expected_block) {
        fwrite(packet.data, 1, BLOCK_SIZE, file);
        received_blocks++;
        printf("Received block %d/%d\n", expected_block + 1, total_blocks);
        expected_block++;

        // Отправка подтверждения
        char ack[32];
        snprintf(ack, sizeof(ack), "ACK%d", packet.block_num);
        sendto(sockfd, ack, strlen(ack), 0,
               (struct sockaddr*)&client_addr, client_len);

        if (expected_block >= total_blocks) break;
    } else {
        printf("Unexpected block: got %d, expected %d\n",
               packet.block_num, expected_block);
        // Повторная отправка подтверждения
        char ack[32];
        snprintf(ack, sizeof(ack), "ACK%d", expected_block - 1);
        sendto(sockfd, ack, strlen(ack), 0,
               (struct sockaddr*)&client_addr, client_len);
    }
}

fclose(file);
```

```
    printf("\nFile received. Total blocks: %d\n", received_blocks);
    close(sockfd);
    return 0;
}
```

**Шаг 2. Реализация клиента**

Файл udp_file_client.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9093
#define BLOCK_SIZE 1024
#define TIMEOUT_SEC 2
#define MAX_RETRIES 5

typedef struct {
    int block_num;
    int total_blocks;
    char data[BLOCK_SIZE];
} Packet;

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

```
}

const char *server_ip = argv[1];
const char *filename = argv[2];

FILE *file = fopen(filename, "rb");
if (!file) { perror("fopen"); return 1; }

// Определение размера файла
fseek(file, 0, SEEK_END);
long file_size = ftell(file);
fseek(file, 0, SEEK_SET);

int total_blocks = (file_size + BLOCK_SIZE - 1) / BLOCK_SIZE;
printf("File size: %ld bytes, blocks: %d\n", file_size, total_blocks);

int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd == -1) { perror("socket"); return 1; }

struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
inet_pton(AF_INET, server_ip, &server_addr.sin_addr);

// Установка таймаута
struct timeval tv = {TIMEOUT_SEC, 0};
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));

Packet packet;
char ack[32];
```

```
            int current_block = 0;
            int retries = 0;

            while (current_block < total_blocks) {
                // Чтение блока из файла
                packet.block_num = current_block;
                packet.total_blocks = total_blocks;
                size_t bytes_read = fread(packet.data, 1, BLOCK_SIZE, file);
                if (bytes_read < BLOCK_SIZE) {
                    memset(packet.data + bytes_read, 0, BLOCK_SIZE - bytes_read);
                }

                // Отправка блока
                sendto(sockfd, &packet, sizeof(packet), 0,
                       (struct sockaddr*)&server_addr, sizeof(server_addr));
                printf("Sent block %d/%d (retry %d)\n", current_block + 1, total_blocks,
retries);

                // Ожидание подтверждения
                socklen_t addr_len = sizeof(server_addr);
                ssize_t bytes = recvfrom(sockfd, ack, sizeof(ack) - 1, 0,
                                         (struct sockaddr*)&server_addr, &addr_len);

                if (bytes > 0 && strncmp(ack, "ACK", 3) == 0) {
                    int acked_block = atoi(ack + 3);
                    if (acked_block == current_block) {
                        current_block++;
                        retries = 0;
                        printf("Received ACK for block %d\n", current_block);
                    }
```

```
        } else {
            retries++;
            printf("Timeout, retransmitting block %d (retry %d/%d)\n",
                   current_block + 1, retries, MAX_RETRIES);

            if (retries >= MAX_RETRIES) {
                printf("Max retries reached. Transfer aborted.\n");
                break;
            }

            // Возврат к началу блока
            fseek(file, current_block * BLOCK_SIZE, SEEK_SET);
        }
    }

    fclose(file);
    close(sockfd);
    printf("\nFile transfer completed!\n");
    return 0;
}
```

**Шаг 3. Компиляция и запуск**

```
bash
gcc -Wall -g udp_file_server.c -o file_server
gcc -Wall -g udp_file_client.c -o file_client

# Терминал 1:
./file_server

# Терминал 2 (создайте тестовый файл):
echo "Test file content" > test.txt
```

./file_client 127.0.0.1 test.txt

**Шаг 4. Пример вывода**

**Сервер:**

UDP File Server listening on port 9093
Waiting for file transfer...
Receiving file: 1 blocks
Received block 1/1

File received. Total blocks: 1

**Клиент:**

File size: 19 bytes, blocks: 1
Sent block 1/1 (retry 0)
Received ACK for block 1

File transfer completed!

**Вывод**

Программа демонстрирует:

- Передачу файла по UDP с механизмом подтверждения.
- Обработку потери пакетов (таймаут и повторная отправка).
- Разбиение файла на блоки фиксированного размера.
- Отличие UDP от TCP: при отсутствии подтверждения пакет может быть потерян, требуется ручная реализация надёжности.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Реализовать UDP-сервер для перевода текста в верхний регистр. Клиент отправляет строку, сервер возвращает её в верхнем регистре. |
| 2 | Создать простой UDP-чат с поддержкой приватных сообщений (формат: @username message). |
| 3 | Реализовать UDP-сервер для вычисления чисел Фибоначчи. Клиент отправляет число N, сервер возвращает N-е число Фибоначчи. |
| 4 | Создать программу для измерения времени передачи пакета по UDP (ping). Клиент отправляет пакет с меткой времени, сервер возвращает его обратно. |
| 5 | Реализовать UDP-сервер для получения текущего времени. Клиент отправляет запрос, сервер возвращает текущее время и дату. |
| 6 | Создать систему для отправки сообщений с подтверждением (ACK). Клиент отправляет сообщение, сервер подтверждает получение. При отсутствии подтверждения клиент повторяет отправку (максимум 3 раза). |

**Обратить внимание:**

– UDP не гарантирует доставку – в программах с требованиями к надёжности нужно реализовывать подтверждения.

– sendto и recvfrom требуют указания адреса при каждой операции.

– Размер дейтаграммы ограничен – для больших данных используйте разбиение на блоки.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание UDP-эхо сервера и клиента;
   – реализация UDP-чата;
   – тестирование потери пакетов;
   – (по варианту) реализация по индивидуальному заданию.
4. Скриншоты (обязательно):
   – работа UDP-эхо клиента и сервера;
   – работа UDP-чата с несколькими клиентами;
   – результаты теста потери пакетов;
   – (по варианту) работа программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Чем отличается UDP от TCP? В каких случаях используют UDP?
2. Какие системные вызовы используются для UDP-сокетов?
3. Зачем нужен bind() в UDP-сервере? Нужен ли bind() в UDP-клиенте?
4. В чём разница между sendto/recvfrom и send/recv?
5. Почему UDP называют протоколом без установления соединения?
6. Гарантирует ли UDP порядок доставки пакетов?
7. Что произойдёт, если размер буфера в recvfrom меньше размера дейтаграммы?
8. Как организовать надёжную передачу данных поверх UDP?
9. Как измерить процент потери пакетов в UDP?
10. Какие приложения используют UDP и почему?
11. Можно ли использовать один UDP-сокет для обмена данными с несколькими клиентами?
12. Как установить таймаут на операцию recvfrom?

# Лабораторная работа №22

# Создание статических и динамических библиотек

**Цель работы:** научиться создавать и использовать статические и динамические библиотеки в Linux, освоить компоновку программ с библиотеками, изучить механизм динамической загрузки библиотек с использованием dlopen, dlsym, dlclose.

**Задачи:**

– Изучить структуру и принципы создания статических библиотек (.a).

– Научиться создавать статическую библиотеку с помощью ar и ranlib.

– Освоить компоновку программы со статической библиотекой.

– Изучить создание динамических разделяемых библиотек (.so).

– Научиться компилировать программы с динамическими библиотеками (флаги -shared, -fPIC).

– Освоить динамическую загрузку библиотек во время выполнения (dlopen, dlsym, dlclose).

– Сравнить преимущества и недостатки статической и динамической компоновки.

## Теоретические сведения

1. **Статические библиотеки (Static libraries)**

   Статическая библиотека (.a – archive) – это архив объектных файлов. При компоновке код библиотеки копируется в исполняемый файл. **Преимущества:** независимость от наличия библиотеки в системе, скорость выполнения (нет накладных расходов на динамическую загрузку).

   **Недостатки:** увеличение размера исполняемого файла, необходимость перекомпоновки при обновлении библиотеки.

2. **Динамические библиотеки (Shared libraries)**

Динамическая библиотека (.so – shared object) загружается в память при запуске программы или во время выполнения. Несколько программ могут использовать одну копию библиотеки в памяти. **Преимущества:** экономия памяти, возможность обновления без перекомпоновки, уменьшение размера исполняемых файлов. **Недостатки:** накладные расходы на загрузку, зависимость от наличия библиотеки в системе.

3. **Создание статической библиотеки**

   bash

   gcc -c file1.c file2.c  # *создание объектных файлов*

   ar rcs libmylib.a file1.o file2.o  # *создание архива*

   ranlib libmylib.a  # *индексация (опционально)*

   Компоновка: gcc main.c -L. -lmylib -o program

4. **Создание динамической библиотеки**

   bash

   gcc -fPIC -c file1.c file2.c  # *Position Independent Code*

   gcc -shared -o libmylib.so file1.o file2.o

   Компоновка: gcc main.c -L. -lmylib -o program

5. **Флаги компиляции для динамических библиотек**
   - -fPIC (Position Independent Code) – код, который может быть загружен по любому адресу.
   - -shared – создание разделяемой библиотеки.
   - -Wl,-rpath,<путь> – добавление пути поиска библиотек в исполняемый файл.
6. **Динамическая загрузка библиотек (dlopen/dlsym)**

   Позволяет загружать библиотеку и получать адреса функций во время выполнения программы.

| Функция | Назначение |
|---|---|
| dlopen(const char *filename, int flag) | Загрузка библиотеки |
| dlsym(void *handle, const char *symbol) | Получение адреса символа |
| dlclose(void *handle) | Выгрузка библиотеки |
| dlerror(void) | Получение сообщения об ошибке |

7. **Пути поиска динамических библиотек**
   - Каталоги, указанные в LD_LIBRARY_PATH
   - Каталоги, указанные в /etc/ld.so.conf
   - Стандартные каталоги (/lib, /usr/lib, /usr/local/lib)
8. **Сравнение подходов**

| Характеристика | Статическая | Динамическая (компоновка) | Динамическая (загрузка) |
|---|---|---|---|
| Время связывания | Компоновка | Компоновка + загрузка | Выполнение |
| Размер программы | Большой | Малый | Малый |
| Зависимости | Нет | Есть (версия .so) | Есть (.so во время выполнения) |
| Гибкость | Низкая | Средняя | Высокая (плагины) |

## Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте каталог для лабораторной работы:

   bash

   mkdir ~/linux_labs/lab22

   cd ~/linux_labs/lab22

2. **Создание статической библиотеки**

2.1. Создайте файлы библиотеки math_utils.h и math_utils.c:

math_utils.h:

c

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
double divide(int a, int b);
int factorial(int n);

#endif
```

math_utils.c:

c

```
#include "math_utils.h"

int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }

double divide(int a, int b) {
    if (b == 0) return 0;
    return (double)a / b;
}

int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
```

```
}
```

2.2. Создайте тестовую программу test_static.c:

```
c
#include <stdio.h>
#include "math_utils.h"

int main() {
    printf("10 + 5 = %d\n", add(10, 5));
    printf("10 - 5 = %d\n", subtract(10, 5));
    printf("10 * 5 = %d\n", multiply(10, 5));
    printf("10 / 3 = %.2f\n", divide(10, 3));
    printf("5! = %d\n", factorial(5));
    return 0;
}
```

2.3. Создайте статическую библиотеку и скомпонуйте:

```
bash
gcc -c math_utils.c -o math_utils.o
ar rcs libmath.a math_utils.o
ranlib libmath.a

gcc test_static.c -L. -lmath -o test_static
./test_static
```

3. **Создание динамической библиотеки**

3.1. Создайте те же файлы для динамической библиотеки или используйте существующие.

3.2. Создайте динамическую библиотеку:

```
bash
gcc -fPIC -c math_utils.c -o math_utils_pic.o
gcc -shared -o libmath.so math_utils_pic.o
```

3.3. Создайте тестовую программу test_dynamic.c (аналогична test_static.c).

3.4. Скомпонуйте с динамической библиотекой:

bash

```
gcc test_dynamic.c -L. -lmath -o test_dynamic
```

3.5. Запустите (может потребоваться указать путь к библиотеке):

bash

```
LD_LIBRARY_PATH=. ./test_dynamic
```

4. **Сравнение размеров исполняемых файлов**

4.1. Сравните размеры test_static и test_dynamic:

bash

```
ls -la test_static test_dynamic
```

5. **Динамическая загрузка библиотеки (dlopen/dlsym)**

5.1. Создайте файл plugin_test.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

typedef int (*math_func)(int, int);
typedef double (*math_func_d)(int, int);
typedef int (*math_func_fact)(int);

int main() {
    void *handle = dlopen("./libmath.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "dlopen error: %s\n", dlerror());
        return 1;
    }
```

```
    // Очистка ошибок
    dlerror();

    // Получение адресов функций
    math_func add = (math_func)dlsym(handle, "add");
    math_func subtract = (math_func)dlsym(handle, "subtract");
    math_func multiply = (math_func)dlsym(handle, "multiply");
    math_func_d divide = (math_func_d)dlsym(handle, "divide");
    math_func_fact factorial = (math_func_fact)dlsym(handle, "factorial");

    const char *error = dlerror();
    if (error) {
        fprintf(stderr, "dlsym error: %s\n", error);
        dlclose(handle);
        return 1;
    }

    printf("=== Dynamic loading with dlopen/dlsym ===\n");
    printf("10 + 5 = %d\n", add(10, 5));
    printf("10 - 5 = %d\n", subtract(10, 5));
    printf("10 * 5 = %d\n", multiply(10, 5));
    printf("10 / 3 = %.2f\n", divide(10, 3));
    printf("5! = %d\n", factorial(5));

    dlclose(handle);
    return 0;
}
```

5.2. Скомпилируйте и запустите (требуется линковка с -ldl):

bash

```
gcc -Wall -g plugin_test.c -o plugin_test -ldl
```

LD_LIBRARY_PATH=. ./plugin_test

6. **Создание библиотеки с функциями для строк**

6.1. Создайте файлы string_utils.h и string_utils.c:

string_utils.h:

c

```
#ifndef STRING_UTILS_H
#define STRING_UTILS_H

void to_upper(char *str);
void to_lower(char *str);
int string_length(const char *str);
char* string_reverse(const char *str);

#endif
```

string_utils.c:

c

```
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "string_utils.h"

void to_upper(char *str) {
    for (int i = 0; str[i]; i++)
        str[i] = toupper(str[i]);
}

void to_lower(char *str) {
    for (int i = 0; str[i]; i++)
        str[i] = tolower(str[i]);
}
```

```
int string_length(const char *str) {
    return strlen(str);
}

char* string_reverse(const char *str) {
    int len = strlen(str);
    char *reversed = malloc(len + 1);
    for (int i = 0; i < len; i++)
        reversed[i] = str[len - 1 - i];
    reversed[len] = '\0';
    return reversed;
}
```

6.2. Создайте статическую и динамическую версии этой библиотеки.

## Пример выполнения задания

**Задача примера:** разработать систему плагинов для программы, где каждый плагин реализует определённый алгоритм обработки данных. Основная программа загружает плагины динамически через dlopen. Плагины реализуют функции с единым интерфейсом.

**Шаг 1. Создание заголовочного файла с интерфейсом плагина**

Файл plugin_interface.h:

c

```
#ifndef PLUGIN_INTERFACE_H
#define PLUGIN_INTERFACE_H

typedef struct {
    const char *name;
    const char *version;
    const char *description;
```

```
    int (*process)(int input, int *output);
} PluginInterface;

#endif
```

**Шаг 2. Реализация плагина 1 (умножение на 2)**

Файл plugin_double.c:

```
c
#include "plugin_interface.h"

int process_double(int input, int *output) {
    *output = input * 2;
    return 0;  // success
}

PluginInterface plugin = {
    .name = "Double Plugin",
    .version = "1.0",
    .description = "Multiplies input by 2",
    .process = process_double
};
```

**Шаг 3. Реализация плагина 2 (возведение в квадрат)**

Файл plugin_square.c:

```
c
#include "plugin_interface.h"

int process_square(int input, int *output) {
    *output = input * input;
    return 0;
}
```

```
PluginInterface plugin = {
    .name = "Square Plugin",
    .version = "1.0",
    .description = "Squares the input",
    .process = process_square
};
```

**Шаг 4. Реализация плагина 3 (сложение с самим собой)**

Файл plugin_addself.c:

```
c
#include "plugin_interface.h"

int process_addself(int input, int *output) {
    *output = input + input;
    return 0;
}

PluginInterface plugin = {
    .name = "AddSelf Plugin",
    .version = "1.0",
    .description = "Adds input to itself",
    .process = process_addself
};
```

**Шаг 5. Создание динамических библиотек плагинов**

```
bash
gcc -fPIC -c plugin_double.c -o plugin_double.o
gcc -shared -o libdouble.so plugin_double.o

gcc -fPIC -c plugin_square.c -o plugin_square.o
gcc -shared -o libsquare.so plugin_square.o
```

```
gcc -fPIC -c plugin_addself.c -o plugin_addself.o
gcc -shared -o libaddself.so plugin_addself.o
```

**Шаг 6. Реализация основной программы**

Файл plugin_manager.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include "plugin_interface.h"

#define MAX_PLUGINS 10
#define MAX_PATH 256

typedef struct {
    void *handle;
    PluginInterface *interface;
} Plugin;

Plugin plugins[MAX_PLUGINS];
int plugin_count = 0;

int load_plugin(const char *path) {
    void *handle = dlopen(path, RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "Failed to load %s: %s\n", path, dlerror());
        return -1;
    }

    dlerror();
```

```
    PluginInterface *interface = (PluginInterface*)dlsym(handle, "plugin");
    const char *error = dlerror();
    if (error) {
        fprintf(stderr, "Failed to get plugin symbol: %s\n", error);
        dlclose(handle);
        return -1;
    }

    plugins[plugin_count].handle = handle;
    plugins[plugin_count].interface = interface;
    plugin_count++;

    printf("Loaded plugin: %s v%s - %s\n",
           interface->name, interface->version, interface->description);
    return 0;
}

void unload_all_plugins() {
    for (int i = 0; i < plugin_count; i++) {
        dlclose(plugins[i].handle);
    }
    plugin_count = 0;
}

void list_plugins() {
    printf("\n=== Available Plugins ===\n");
    for (int i = 0; i < plugin_count; i++) {
        printf("%d. %s v%s\n", i + 1, plugins[i].interface->name,
               plugins[i].interface->version);
        printf("   %s\n", plugins[i].interface->description);
```

```
        }
        printf("========================\n\n");
    }

    int main() {
        printf("=== Plugin Manager ===\n");
        printf("Loading plugins from current directory...\n\n");

        // Загрузка всех плагинов из текущей директории
        const char *plugin_names[] = {"libdouble.so", "libsquare.so",
"libaddself.so"};
        for (int i = 0; i < 3; i++) {
            load_plugin(plugin_names[i]);
        }

        if (plugin_count == 0) {
            printf("No plugins loaded.\n");
            return 1;
        }

        list_plugins();

        int input;
        printf("Enter a number to process: ");
        scanf("%d", &input);

        printf("\n=== Results ===\n");
        for (int i = 0; i < plugin_count; i++) {
            int output;
            plugins[i].interface->process(input, &output);
```

```
        printf("%s: %d -> %d\n", plugins[i].interface->name, input, output);
    }

    unload_all_plugins();
    printf("\nAll plugins unloaded.\n");
    return 0;
}
```

**Шаг 7. Компиляция и запуск**

```
bash
gcc -Wall -g plugin_manager.c -o plugin_manager -ldl

# Запуск
./plugin_manager
```

**Шаг 8. Пример вывода**

```
=== Plugin Manager ===
Loading plugins from current directory...

Loaded plugin: Double Plugin v1.0 - Multiplies input by 2
Loaded plugin: Square Plugin v1.0 - Squares the input
Loaded plugin: AddSelf Plugin v1.0 - Adds input to itself

=== Available Plugins ===
1. Double Plugin v1.0
   Multiplies input by 2
2. Square Plugin v1.0
   Squares the input
3. AddSelf Plugin v1.0
   Adds input to itself
========================
```

Enter a number to process: 5

=== Results ===

Double Plugin: 5 -> 10

Square Plugin: 5 -> 25

AddSelf Plugin: 5 -> 10

All plugins unloaded.

**Вывод**

Программа демонстрирует:

- Создание динамических библиотек с единым интерфейсом.
- Загрузку плагинов во время выполнения с помощью dlopen.
- Получение адресов структур с функциями через dlsym.
- Гибкую архитектуру, позволяющую добавлять новые плагины без перекомпиляции основной программы.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Создать статическую и динамическую библиотеки для работы с комплексными числами (сложение, вычитание, умножение, деление). Написать тестовую программу. |
| 2 | Создать динамическую библиотеку для работы с массивами (сортировка пузырьком, поиск максимума/минимума, реверс). Использовать dlopen для загрузки. |
| 3 | Реализовать систему плагинов для математических операций (+, -, *, /, ^). Каждый плагин реализует одну операцию. Основная программа загружает плагины и вызывает их. |
| 4 | Создать статическую библиотеку для работы с датами (проверка високосного года, количество дней в месяце, разница в днях). Написать программу, использующую библиотеку. |
| 5 | Создать динамическую библиотеку для работы со строками (функции: concat, substring, find). Написать программу, загружающую библиотеку и демонстрирующую её работу. |
| 6 | Реализовать программу-калькулятор с поддержкой плагинов. Плагины реализуют различные операции. Основная программа сканирует директорию с плагинами и загружает все найденные .so файлы. |

**Обратить внимание:**

– Для динамических библиотек обязательно использовать -fPIC при компиляции.

– При использовании dlopen линковать программу с -ldl.

– Для экспорта символов из динамической библиотеки можно использовать -Wl,-export-dynamic.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – создание статической библиотеки и её использование;
   – создание динамической библиотеки и её использование;
   – динамическая загрузка библиотек с dlopen/dlsym;
   – (по варианту) реализация по индивидуальному заданию.
4. Скриншоты (обязательно):
   – компиляция и запуск программы со статической библиотекой;
   – компиляция и запуск программы с динамической библиотекой;
   – работа plugin_test с dlopen;
   – сравнение размеров исполняемых файлов;
   – (по варианту) работа программы по индивидуальному заданию.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Что такое статическая библиотека? Как она создаётся?
2. Какая команда используется для создания архива объектных файлов?
3. В чём отличие статической библиотеки от динамической?
4. Что такое -fPIC и зачем он нужен при создании динамических библиотек?
5. Как создать динамическую библиотеку?
6. Как скомпоновать программу с динамической библиотекой?
7. Как указать системе путь к динамическим библиотекам?
8. Какие функции используются для динамической загрузки библиотек?
9. Что возвращает dlsym? Как обработать ошибку?
10. В каких случаях применяется динамическая загрузка (dlopen)?
11. Какие преимущества даёт использование динамических библиотек?
12. Как проверить, какие динамические библиотеки использует программа?

# Лабораторная работа №23

# Профилирование и анализ производительности

**Цель работы:** освоить инструменты профилирования и анализа производительности в Linux, научиться выявлять узкие места в программах, отслеживать системные и библиотечные вызовы, находить утечки памяти и оптимизировать код с использованием strace, ltrace, gprof, valgrind, perf.

**Задачи:**

– Изучить утилиту strace для трассировки системных вызовов.

– Освоить ltrace для отслеживания вызовов библиотечных функций.

– Научиться профилировать программы с помощью gprof.

– Освоить valgrind (memcheck) для поиска утечек памяти и ошибок работы с памятью.

– Изучить perf для анализа производительности (счётчики событий, кэш-промахи, выборка).

– Провести анализ производительности тестовых программ с утечками и узкими местами.

## Теоретические сведения

1. **strace – трассировка системных вызовов**

   strace перехватывает и записывает все системные вызовы, выполняемые процессом. Позволяет увидеть, какие файлы открывает программа, какие сокеты создаёт, сколько времени занимают системные вызовы.

   **Основные опции:**

   - -c – сводная статистика (время, количество вызовов, ошибки)
   - -t – добавить временную метку
   - -T – показать время выполнения каждого системного вызова
   - -e trace=file – трассировать только вызовы, связанные с файлами
   - -p PID – присоединиться к работающему процессу

- -o файл – записать вывод в файл

2. **ltrace – трассировка библиотечных вызовов**

   ltrace отслеживает вызовы динамических библиотек (функции libc, libm, и др.). Позволяет увидеть, какие функции стандартной библиотеки вызывает программа.

   **Основные опции:**

   - -c – сводная статистика
   - -e – фильтр (например, -e printf – только printf)
   - -l /lib – ограничиться библиотеками из указанного каталога

3. **gprof – профилирование по времени**

   gprof требует компиляции с опцией -pg. При выполнении программы создаётся файл gmon.out, который затем анализируется. Показывает, сколько времени было потрачено в каждой функции и сколько раз она вызывалась.

   **Команды:**

   bash

   gcc -pg program.c -o program

   ./program

   gprof program gmon.out > analysis.txt

4. **valgrind (memcheck) – проверка памяти**

   valgrind с инструментом memcheck обнаруживает утечки памяти, некорректное использование malloc/free, доступ за границы массива, использование неинициализированной памяти.

   **Основные опции:**

   - --leak-check=full – полная проверка утечек
   - --track-origins=yes – отслеживание источника неинициализированных значений
   - --log-file=файл – запись в файл

5. **perf – системное профилирование**

perf – мощный инструмент для анализа производительности на уровне процессора и ядра. Позволяет собирать счётчики событий (кэш-промахи, промахи TLB, выполненные инструкции).

**Основные команды:**

- perf stat ./program – статистика счётчиков
- perf record -e cycles ./program – запись выборок
- perf report – просмотр отчёта

6. **Сравнение инструментов**

| Инструмент | Что анализирует | Типичное применение |
|---|---|---|
| strace | Системные вызовы | Поиск проблем с файлами/сокетами |
| ltrace | Библиотечные вызовы | Понимание поведения программы |
| gprof | Время выполнения функций | Поиск узких мест в коде |
| valgrind | Ошибки памяти, утечки | Отладка, обеспечение качества |
| perf | Счётчики CPU, кэша | Оптимизация производительности |

**Методика выполнения работы**

1. **Подготовка рабочего каталога и тестовых программ**

1.1. Создайте каталог для лабораторной работы:

bash

```
mkdir ~/linux_labs/lab23
cd ~/linux_labs/lab23
```

1.2. Создайте тестовую программу slow_program.c:

c

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>

void slow_function() {
    for (int i = 0; i < 1000000; i++) {
        getpid();  // системный вызов
    }
}

void fast_function() {
    volatile int x = 0;
    for (int i = 0; i < 1000000; i++) {
        x++;
    }
}

int main() {
    printf("Starting slow function...\n");
    slow_function();
    printf("Starting fast function...\n");
    fast_function();
    printf("Done!\n");
    return 0;
}
```

1.3. Создайте программу с утечкой памяти memory_leak.c:

```
c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void leak_memory() {
```

```
    char *ptr = malloc(1024);
    strcpy(ptr, "This memory will leak");
    // Нет free(ptr)
}

void bad_access() {
    int *arr = malloc(10 * sizeof(int));
    arr[10] = 42;  // выход за границы
    free(arr);
}

void uninitialized_use() {
    int x;
    if (x > 0) {  // x не инициализирована
        printf("x is positive\n");
    }
}

int main() {
    for (int i = 0; i < 100; i++) {
        leak_memory();
    }
    bad_access();
    uninitialized_use();
    return 0;
}
```

2. **Использование strace**

   2.1. Базовый анализ slow_program:

   bash

   ```
   gcc -Wall -g slow_program.c -o slow_program
   ```

*# Просмотр системных вызовов*

strace ./slow_program 2>&1 | head -50

*# Статистика системных вызовов*

strace -c ./slow_program

*# С временными метками*

strace -t -T ./slow_program 2>&1 | head -20

*# Только вызовы getpid*

strace -e trace=getpid ./slow_program

2.2. Анализ открытия файлов:

bash

*# Трассировка файловых операций*

strace -e trace=file ./slow_program

3. **Использование ltrace**

   3.1. Анализ библиотечных вызовов:

   bash

   *# Установка ltrace (если не установлен)*

   sudo apt install ltrace

   ltrace ./slow_program 2>&1 | head -30

   *# Статистика*

   ltrace -c ./slow_program

   *# Только вызовы printf*

   ltrace -e printf ./slow_program

4. **Профилирование с gprof**

4.1. Компиляция с -pg и анализ:

bash

gcc -pg -Wall -g slow_program.c -o slow_program_pg

./slow_program_pg

ls -la gmon.out

gprof slow_program_pg gmon.out > gprof_report.txt

cat gprof_report.txt

4.2. Просмотр отчёта (обратите внимание на % time, calls, self seconds).

5. **Поиск утечек памяти с valgrind**

5.1. Установка и запуск:

bash

*# Установка valgrind*

sudo apt install valgrind

gcc -Wall -g memory_leak.c -o memory_leak

*# Базовый анализ*

valgrind ./memory_leak

*# Полная проверка утечек*

valgrind --leak-check=full ./memory_leak

*# С отслеживанием источника неинициализированных значений*

valgrind --leak-check=full --track-origins=yes ./memory_leak

5.2. Анализ вывода valgrind (потеряно 100 блоков по 1024 байта, неинициализированное использование).

6. **Анализ производительности с perf**

6.1. Установка и базовое использование:

bash

*# Установка linux-tools (версия должна соответствовать ядру)*

sudo apt install linux-tools-common linux-tools-generic

*# Статистика выполнения*

perf stat ./slow_program

*# Сбор счётчиков кэша*

perf stat -e cache-misses,cache-references ./slow_program

*# Запись выборок*

perf record -e cycles ./slow_program

perf report

6.2. Сравнение slow_function и fast_function по счётчикам.

## Пример выполнения задания

**Задача примера:** разработать программу с искусственными проблемами производительности (много системных вызовов, неэффективные алгоритмы, утечка памяти) и провести полный анализ с использованием strace, ltrace, gprof, valgrind и perf. На основе анализа оптимизировать программу.

**Шаг 1. Создание проблемной программы**

Файл problematic.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

// Проблема 1: много системных вызовов
void inefficient_io() {
    FILE *f = fopen("/tmp/test.txt", "w");
```

```
    for (int i = 0; i < 1000; i++) {
        fprintf(f, "Line %d\n", i);  // 1000 системных вызовов
    }
    fclose(f);
}

// Проблема 2: неэффективный алгоритм (пузырёк на больших данных)
void bubble_sort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Проблема 3: утечка памяти
void leak_memory() {
    char *buffer = malloc(1024);
    strcpy(buffer, "Leaked memory");
    // забыли free(buffer)
}

int main() {
    printf("=== Problematic Program ===\n");

    // Проблема 1
```

```
    printf("1. Running inefficient I/O...\n");
    inefficient_io();

    // Проблема 2
    printf("2. Running bubble sort on 10000 elements...\n");
    int *arr = malloc(10000 * sizeof(int));
    for (int i = 0; i < 10000; i++) {
        arr[i] = rand() % 10000;
    }
    bubble_sort(arr, 10000);
    free(arr);

    // Проблема 3
    printf("3. Leaking memory...\n");
    for (int i = 0; i < 10; i++) {
        leak_memory();
    }

    printf("Done!\n");
    return 0;
}
```

**Шаг 2. Анализ с strace**

bash

```
gcc -Wall -g problematic.c -o problematic

# Анализ системных вызовов
strace -c ./problematic 2>&1 | grep -E "(write|open|fstat)"
```

**Результат:** Много вызовов write (по одному на каждую строку).

**Шаг 3. Анализ с ltrace**

bash

ltrace -c ./problematic 2>&1 | head -20

**Результат:** 1000 вызовов fprintf.

**Шаг 4. Профилирование с gprof**

bash

```
gcc -pg -Wall -g problematic.c -o problematic_pg
./problematic_pg
gprof problematic_pg gmon.out > gprof_report.txt
cat gprof_report.txt | head -40
```

**Результат:** bubble_sort занимает 90% времени.

**Шаг 5. Анализ с valgrind**

bash

```
valgrind --leak-check=full ./problematic 2>&1 | grep -A 5 "lost"
```

**Результат:** 10 блоков по 1024 байта потеряно.

**Шаг 6. Оптимизированная версия**

Файл optimized.c:

c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int compare(const void *a, const void *b) {
    return *(int*)a - *(int*)b;
}

void efficient_io() {
    FILE *f = fopen("/tmp/test.txt", "w");
    char buffer[64];
    for (int i = 0; i < 1000; i++) {
        int len = snprintf(buffer, sizeof(buffer), "Line %d\n", i);
        fwrite(buffer, 1, len, f);  // буферизованная запись
```

```
    }
    fclose(f);
}

int main() {
    printf("=== Optimized Program ===\n");

    printf("1. Running efficient I/O...\n");
    efficient_io();

    printf("2. Running qsort on 10000 elements...\n");
    int *arr = malloc(10000 * sizeof(int));
    for (int i = 0; i < 10000; i++) {
        arr[i] = rand() % 10000;
    }
    qsort(arr, 10000, sizeof(int), compare);
    free(arr);

    printf("3. No memory leak.\n");

    printf("Done!\n");
    return 0;
}
```

**Шаг 7. Сравнение производительности**

```
bash
gcc -Wall -g optimized.c -o optimized

echo "=== Problematic ==="
time ./problematic
```

```
echo "=== Optimized ==="
time ./optimized
```

**Результат:** Оптимизированная версия работает в 10–50 раз быстрее.

## Индивидуальные задания

Номер варианта соответствует номеру в списке группы.

Для вариантов >6– цикл повторяется: вариант 6→ вариант 1, вариант 7→ вариант 2 и т.д.

| Вариант | Задание |
|---|---|
| 1 | Создать программу с большим количеством вызовов malloc/free и проанализировать её с valgrind. Исправить утечки. |
| 2 | Написать программу, выполняющую рекурсивный обход файловой системы, и проанализировать её с strace. Оптимизировать количество системных вызовов. |
| 3 | Создать программу с неэффективным алгоритмом ($O(n^2)$) и сравнить профиль с оптимизированной версией через gprof. |
| 4 | Написать программу с множеством вызовов printf и проанализировать её через ltrace. Заменить на буферизованный вывод. |
| 5 | Создать программу с интенсивными вычислениями с плавающей точкой и проанализировать счётчики perf (кэш-промахи, инструкции). |
| 6 | Написать программу, открывающую много файлов и не закрывающую их. Проанализировать с strace и valgrind. |

**Обратить внимание:**

– Для gprof программа должна завершиться корректно, чтобы был создан файл gmon.out.

– valgrind замедляет выполнение программы – не использовать для production.

– Для perf может потребоваться установка дополнительных пакетов.

## Требования к отчёту

Отчёт оформляется в электронном виде (PDF или DOCX) и содержит следующие разделы:

1. Титульный лист (ФИО, группа, дата, название работы).
2. Цель и задачи (из методички).
3. Ход выполнения (пошаговое описание с командами и пояснениями):
   – анализ программы с strace;
   – анализ с ltrace;
   – профилирование с gprof;
   – проверка памяти с valgrind;
   – анализ производительности с perf;
   – (по варианту) оптимизация программы по индивидуальному заданию.
4. Скриншоты (обязательно):
   – вывод strace -c;
   – вывод ltrace -c;
   – отчёт gprof;
   – вывод valgrind --leak-check=full;
   – вывод perf stat;
   – сравнение времени выполнения до и после оптимизации.
5. Исходный код программы (вставка текстом всех созданных файлов).
6. Выводы по работе.
7. Ответы на контрольные вопросы (письменно).

Отчёт сдаётся преподавателю до следующего лабораторного занятия.

## Контрольные вопросы

1. Для чего используется strace? Какую информацию можно получить?
2. В чём разница между strace и ltrace?
3. Как с помощью strace узнать, какие файлы открывает программа?
4. Как получить статистику системных вызовов в strace?
5. Что такое gprof? Как компилировать программу для профилирования?
6. Как прочитать отчёт gprof? Что означают колонки % time, self seconds, calls?
7. Как valgrind обнаруживает утечки памяти?
8. Что означают сообщения definitely lost, indirectly lost, possibly lost?
9. Для чего используется perf stat? Какие счётчики можно анализировать?
10. Какой инструмент лучше подходит для поиска узких мест в CPU-интенсивных вычислениях?
11. Какой инструмент лучше подходит для поиска проблем с вводом-выводом?
12. Какую информацию даёт опция -T в strace?

# СПИСОК ЛИТЕРАТУРЫ

**Основная литература**

1. Керриск, М. The Linux Programming Interface: A Linux and UNIX System Programming Handbook / М. Керриск. — San Francisco: No Starch Press, 2010. — 1552 с. — ISBN 978-1-59327-220-3.

2. Стивенс, У. Р. Advanced Programming in the UNIX Environment / У. Р. Стивенс, С. А. Раго. — 3-е изд. — Upper Saddle River: Addison-Wesley, 2013. — 1024 с. — ISBN 978-0-321-63773-4.

3. Лав, Р. Linux System Programming / Р. Лав. — 2-е изд. — Sebastopol: O'Reilly Media, 2013. — 456 с. — ISBN 978-1-4493-3953-1.

4. Бовет, Д. Understanding the Linux Kernel / Д. Бовет, М. Чезати. — 3-е изд. — Sebastopol: O'Reilly Media, 2005. — 942 с. — ISBN 978-0-596-00565-8.

**Дополнительная литература**

5. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с. — ISBN 978-5-496-01395-2.

6. Робертс, Э. Программирование на языке С: принципы и практика / Э. Робертс. — М.: Вильямс, 2016. — 640 с. — ISBN 978-5-8459-2027-5.

7. Учебные пособия на русском языке

8. Моренкова, О. И. Операционные системы. Linux: учебное пособие для СПО / О. И. Моренкова. — Саратов: Профобразование, 2024. — 104 с. — ISBN 978-5-4488-1173-9.

9. Елисеев, А. И. Основы администрирования и системного программирования в операционной системе Linux: учебное пособие. В 2 ч. Ч. I / А. И. Елисеев, А. В. Яковлев, А. С. Дерябин. — Тамбов: Тамбовский государственный технический университет, 2020. — 80 с.

**Интернет-ресурсы и документация**

10. The Linux man-pages project — официальная документация по системным вызовам и библиотечным функциям Linux. — Режим доступа: https://www.kernel.org/doc/man-pages/

11. POSIX.1-2017 Specification — спецификация стандарта POSIX. — Режим доступа: https://pubs.opengroup.org/onlinepubs/9699919799/

12. GNU C Library (glibc) Documentation — документация по стандартной библиотеке C. — Режим доступа: https://www.gnu.org/software/libc/documentation.html

13. Valgrind Documentation — документация по инструменту анализа памяти. — Режим доступа: https://valgrind.org/docs/

14. perf: Linux profiling with performance counters — документация по профилированию. — Режим доступа: https://perf.wiki.kernel.org/