Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации
Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Кафедра Системного программирования

ЛАБОРАТОРНАЯ РАБОТА №4
по дисциплине
ОПЕРАЦИОННЫЕ СИСТЕМЫ
на тему:
«РЕАЛИЗАЦИЯ ОБМЕНА ДАННЫМИ МЕЖДУ ПРОЦЕССАМИ»

Выполнил:
студент ____Шамсутдинов Р.Ф._____
(Ф.И.О.)
группа_____БВТ2201_____

Проверил:
_____Королькова Т. В._____
(Ф.И.О., должность преподавателя)

Оценка_____

Дата _____30.04.2025_____

Москва 2025

Введение

Цель работы:

• изучение системных средств обмена данными между процессами в ОС GNU/LINUX

• получение практических навыков использования механизмов межпроцессного взаимодействия

Задание:

1. Изучите теоретическую часть лабораторной работы.

2. Написать программу, выполняющую с помощью ВСЕХ рассмотренных выше системных средств обмена данными между процессами (разделяемая память, сокеты, каналы) одну из задач (в соответствии с № по журналу), приведенных в Таблице 1. При выполнении задачи необходимо создать как минимум 2 ведомых процесса, выполняющих переданные ведущим процессом подзадачи и возвращающие результаты ведущему процессу. Финальную обработку результатов при необходимости осуществлять в ведущем процессе.

Задача из Таблицы 1.

Реализовать обмен текстовыми сообщениями между несколькими процессами. Обеспечить возможность отправки сообщения сразу нескольким адресатам. Реализовать подтверждение приема сообщения адресатом или, в случае потери сообщения, повторную его передачу.

Ход работы

1) Реализация на каналах.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/select.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

#define NCHILD   2
#define MAXTEXT  256
#define MSG_COUNT 3       // число сообщений для отправки
#define TIMEOUT_SEC 2     // таймаут ожидания ACK в секундах

typedef struct {
    int  msg_id;
    char text[MAXTEXT];
} Message;

void child_proc(int id, int read_fd, int write_fd) {
    Message msg;


    while (1) {

        // ждём сообщение от parent
        ssize_t r = read(read_fd, &msg, sizeof(msg));
        if (r == 0) {
            // канала нет — parent закрыл концы
            break;
        } else if (r < 0) {
            if (errno == EINTR) continue;
            perror("child read");
            exit(1);
        }
        // обработали сообщение
        printf("Child %d received msg #%d: \"%s\"\n", id, msg.msg_id, msg.text);
        // отправляем ACK обратно
        if (write(write_fd, &msg.msg_id, sizeof(int)) < 0) {
            perror("child write ACK");
            exit(1);
        }
    }
    close(read_fd);
    close(write_fd);
    exit(0);
}
```

Рисунок 1. Библиотеки, константы, функция дочернего процесса

```c
int main() {
    int p2c[NCHILD][2], c2p[NCHILD][2];
    pid_t pid;
    int i;

    // создаём пары каналов
    for (i = 0; i < NCHILD; i++) {
        if (pipe(p2c[i]) < 0 || pipe(c2p[i]) < 0) {
            perror("pipe");
            exit(1);
        }
    }

    // форкаем детей
    for (i = 0; i < NCHILD; i++) {
        pid = fork();
        if (pid < 0) {
            perror("fork");
            exit(1);
        } else if (pid == 0) {
            // в child: закрываем ненужные концы
            close(p2c[i][1]);    // не пишем в свой p2c
            close(c2p[i][0]);    // не читаем из c2p
            // закрываем каналы других детей
            for (int j = 0; j < NCHILD; j++) if (j != i) {
                close(p2c[j][0]); close(p2c[j][1]);
                close(c2p[j][0]); close(c2p[j][1]);
            }
            child_proc(i, p2c[i][0], c2p[i][1]);
        }
    }

    // в parent: закрываем ненужные концы
    for (i = 0; i < NCHILD; i++) {
        close(p2c[i][0]);    // parent только пишет в p2c
        close(c2p[i][1]);    // parent только читает из c2p
    }

    // готовим массив сообщений
    Message msgs[MSG_COUNT] = {
        { .msg_id = 1, .text = "Hello, child!" },
        { .msg_id = 2, .text = "How are you?" },
        { .msg_id = 3, .text = "Goodbye!" }
    };
```

Рисунок 2. Функция мейн, создание каналов и дочерних процессов, закрытие ненужных концов, массив сообщений

```c
// отправка сообщений и ожидание ACK
for (int m = 0; m < MSG_COUNT; m++) {
    int acked[NCHILD] = {0};
    int remaining = NCHILD;

    // 1) Первый раз шлём всем
    for (i = 0; i < NCHILD; i++) {
        write(p2c[i][1], &msgs[m], sizeof(Message));
        printf("Parent sent msg #%d to child %d\n", msgs[m].msg_id, i);
    }
```

Рисунок 3. Отправка сообщений всем дочерним процессам

```c
    // 2) Цикл: ждём таймаут или ACK, и только по таймауту шлём повторно
    while (remaining > 0) {
        struct timeval tv = { .tv_sec = TIMEOUT_SEC, .tv_usec = 0 };
        fd_set readfds;
        FD_ZERO(&readfds);
        int maxfd = -1;
        for (i = 0; i < NCHILD; i++) {
            if (!acked[i]) {
                FD_SET(c2p[i][0], &readfds);
                if (c2p[i][0] > maxfd) maxfd = c2p[i][0];
            }
        }

        int ready = select(maxfd + 1, &readfds, NULL, NULL, &tv);
        if (ready < 0) {
            if (errno == EINTR) continue;
            perror("select");
            exit(1);
        }

        if (ready == 0) {
            // полный таймаут — повторяем отправку тем, кто ещё не подтвердил
            printf("Timeout waiting ACK for msg #%d; retransmitting to unacked...\n",
                   msgs[m].msg_id);
            for (i = 0; i < NCHILD; i++) {
                if (!acked[i]) {
                    write(p2c[i][1], &msgs[m], sizeof(Message));
                    printf("Parent re-sent msg #%d to child %d\n",
                           msgs[m].msg_id, i);
                }
            }
        } else {
            // пришли одни или несколько ACK — читаем их, но НЕ шлём новых сообщений
            for (i = 0; i < NCHILD; i++) {
                if (!acked[i] && FD_ISSET(c2p[i][0], &readfds)) {
                    int ack_id;
                    if (read(c2p[i][0], &ack_id, sizeof(int)) > 0
                        && ack_id == msgs[m].msg_id) {
                        acked[i] = 1;
                        remaining--;
                        printf("Parent received ACK for msg #%d from child %d\n",
                               ack_id, i);
                    }
                }
            }
            // сразу же возвращаемся в select: если ещё есть 'remaining',
            // мы либо дождёмся следующего ACK (до истечения tv),
            // либо timeout — и тогда ретранслируем.
        }
    }
}
```

Рисунок 4. Ожидание подтверждений, повторная отправка

```c
    // закрываем каналы parent → child (чтобы дети вышли из read)
    for (i = 0; i < NCHILD; i++) {
        close(p2c[i][1]);
    }

    // ждём завершения детей
    for (i = 0; i < NCHILD; i++) {
        wait(NULL);
    }

    // закрываем оставшиеся каналы
    for (i = 0; i < NCHILD; i++) {
        close(c2p[i][0]);
    }

    printf("Parent: all done.\n");
    return 0;
}
```

Рисунок 5. Закрытие каналов, дочерних процессов.

```
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$ gcc channels.c -o channels
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$ ./channels
Parent sent msg #1 to child 0
Parent sent msg #1 to child 1
Child 0 received msg #1: "Hello, child!"
Parent received ACK for msg #1 from child 0
Child 1 received msg #1: "Hello, child!"
Parent received ACK for msg #1 from child 1
Parent sent msg #2 to child 0
Child 0 received msg #2: "How are you?"
Parent sent msg #2 to child 1
Parent received ACK for msg #2 from child 0
Child 1 received msg #2: "How are you?"
Parent received ACK for msg #2 from child 1
Parent sent msg #3 to child 0
Parent sent msg #3 to child 1
Child 0 received msg #3: "Goodbye!"
Child 1 received msg #3: "Goodbye!"
Parent received ACK for msg #3 from child 0
Parent received ACK for msg #3 from child 1
Parent: all done.
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$
```

Рисунок 6. Сборка и вывод программы

2) Реализация с общей памятью

```c
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>        // shm_open
#include <sys/mman.h>     // mmap, shm_unlink
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>
#include <time.h>
#include <semaphore.h>

#define NCHILD     2
#define MAXTEXT    256
#define MSG_COUNT  3
#define TIMEOUT_SEC 2

typedef struct {
    int  msg_id;
    char text[MAXTEXT];
} Message;

typedef struct {
    Message msg;
    sem_t    sem_msg;   // сигнал от parent: "у тебя новое msg"
    sem_t    sem_ack;   // сигнал от child:  "я принял msg"
} Slot;

void child_proc(int id, Slot *slot) {
    for (;;) {
        // ждём, пока parent выдаст новое сообщение
        sem_wait(&slot->sem_msg);

        // читаем и обрабатываем
        printf("Child %d received msg #%d: \"%s\"\n",
               id, slot->msg.msg_id, slot->msg.text);

        // подтверждаем получение
        sem_post(&slot->sem_ack);

        // последний msg_id == MSG_COUNT — выходим
        if (slot->msg.msg_id == MSG_COUNT)
            break;
    }
    exit(0);
}
```

Рисунок 7. Библиотеки, константы, функция дочернего процесса

```c
int main() {
    const char *shm_name = "/shm_sem_chat";
    int shm_fd = shm_open(shm_name, O_CREAT | O_RDWR, 0666);
    if (shm_fd < 0) { perror("shm_open"); exit(1); }
    // резервируем место под NCHILD слотов
    if (ftruncate(shm_fd, sizeof(Slot)*NCHILD) < 0) {
        perror("ftruncate"); exit(1);
    }
    // мапим
    Slot *slots = mmap(NULL, sizeof(Slot)*NCHILD,
                       PROT_READ | PROT_WRITE,
                       MAP_SHARED, shm_fd, 0);
    if (slots == MAP_FAILED) { perror("mmap"); exit(1); }

    // инициализируем семафоры и поля
    for (int i = 0; i < NCHILD; i++) {
        slots[i].msg.msg_id = 0;
        slots[i].msg.text[0] = '\0';
        sem_init(&slots[i].sem_msg, 1, 0);  // pshared=1, init=0
        sem_init(&slots[i].sem_ack, 1, 0);
    }

    // форкаем детей
    for (int i = 0; i < NCHILD; i++) {
        pid_t pid = fork();
        if (pid < 0) {
            perror("fork"); exit(1);
        } else if (pid == 0) {
            child_proc(i, &slots[i]);
        }
    }

    // parent: массив текстов
    const char *texts[MSG_COUNT] = {
        "Hello, child!",
        "How are you?",
        "Goodbye!"
    };
```

Рисунок 8. Создание общей памяти, семафоры, создание дочерних процессов

```c
// буфер абсолютного времени для sem_timedwait
struct timespec ts;

// рассылка и ожидание ACK
for (int m = 1; m <= MSG_COUNT; m++) {
    int acked[NCHILD] = {0}, remaining = NCHILD;

    // 1) первый раз шлём всем
    for (int i = 0; i < NCHILD; i++) {
        slots[i].msg.msg_id = m;
        strncpy(slots[i].msg.text, texts[m-1], MAXTEXT);
        sem_post(&slots[i].sem_msg);
        printf("Parent sent msg #%d to child %d\n", m, i);
    }

    // 2) цикл: ждём ACK с таймаутом, и по таймауту — ретранслируем
    while (remaining > 0) {
        // ждём по каждому не-acked child
        for (int i = 0; i < NCHILD; i++) {
            if (acked[i]) continue;

            // готовим abs timeout
            clock_gettime(CLOCK_REALTIME, &ts);
            ts.tv_sec += TIMEOUT_SEC;

            if (sem_timedwait(&slots[i].sem_ack, &ts) == 0) {
                // пришёл ACK
                acked[i] = 1;
                remaining--;
                printf("Parent received ACK for msg #%d from child %d\n", m, i);
            } else if (errno == ETIMEDOUT) {
                // таймаут именно для этого child — ретранслируем ВСЕМ не-acked
                printf("Timeout waiting ACK for msg #%d; retransmitting to unacked...\n", m);
                for (int j = 0; j < NCHILD; j++) {
                    if (!acked[j]) {
                        sem_post(&slots[j].sem_msg);
                        printf("Parent re-sent msg #%d to child %d\n", m, j);
                    }
                }
                break;  // выйдем из for(i) и начнём ждать заново
            } else {
                perror("sem_timedwait");
                exit(1);
            }
        }
    }
}
```

Рисунок 9. Отправка сообщений и ожидание подтверждений, повторная отправка.

```c
    // ждём всех детей
    for (int i = 0; i < NCHILD; i++)
        wait(NULL);

    // cleanup
    for (int i = 0; i < NCHILD; i++) {
        sem_destroy(&slots[i].sem_msg);
        sem_destroy(&slots[i].sem_ack);
    }
    munmap(slots, sizeof(Slot)*NCHILD);
    shm_unlink(shm_name);

    printf("Parent: all done.\n");
    return 0;
}
```

Рисунок 10. Ожидание завершения дочерних процессов, очистка общей памяти.

```
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$ gcc shared_memory.c -o shared_memory -lrt -pthread
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$ ./shared_memory
Parent sent msg #1 to child 0
Parent sent msg #1 to child 1
Child 0 received msg #1: "Hello, child!"
Parent received ACK for msg #1 from child 0
Child 1 received msg #1: "Hello, child!"
Parent received ACK for msg #1 from child 1
Parent sent msg #2 to child 0
Parent sent msg #2 to child 1
Child 0 received msg #2: "How are you?"
Child 1 received msg #2: "How are you?"
Parent received ACK for msg #2 from child 0
Parent received ACK for msg #2 from child 1
Parent sent msg #3 to child 0
Parent sent msg #3 to child 1
Child 0 received msg #3: "Goodbye!"
Child 1 received msg #3: "Goodbye!"
Parent received ACK for msg #3 from child 0
Parent received ACK for msg #3 from child 1
Parent: all done.
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$
```

Рисунок 11. Сборка и вывод программы

3) Реализация на сокетах.

```c
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stddef.h>       // offsetof
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <sys/time.h>     // struct timeval

#define NCHILD      2
#define MAXTEXT     256
#define MSG_COUNT   3
#define TIMEOUT_SEC 2

typedef struct {
    int  msg_id;
    char text[MAXTEXT];
} Message;

typedef struct {
    int child_id;
    int msg_id;
} AckMessage;

static void cleanup_socket(const char *path) {
    unlink(path);
}
```

Рисунок 12. Библиотеки, константы

```c
static void child_proc(int id) {
    int sock;
    struct sockaddr_un src, dst;
    char path[64];

    sock = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sock < 0) { perror("child socket"); exit(1); }

    // свой адрес
    snprintf(path, sizeof(path), "/tmp/child%d.sock", id);
    cleanup_socket(path);
    memset(&src, 0, sizeof(src));
    src.sun_family = AF_UNIX;
    strcpy(src.sun_path, path);
    if (bind(sock, (struct sockaddr*)&src,
             offsetof(struct sockaddr_un, sun_path) + strlen(path)) < 0) {
        perror("child bind"); exit(1);
    }

    // адрес родителя
    memset(&dst, 0, sizeof(dst));
    dst.sun_family = AF_UNIX;
    strcpy(dst.sun_path, "/tmp/parent.sock");

    Message msg;
    while (1) {
        ssize_t r = recvfrom(sock, &msg, sizeof(msg), 0, NULL, NULL);
        if (r < 0) {
            if (errno == EINTR) continue;
            perror("child recvfrom"); exit(1);
        }
        printf("Child %d received msg #%d: \"%s\"\n",
               id, msg.msg_id, msg.text);

        // отправляем ACK с child_id и msg_id
        AckMessage ack = { .child_id = id, .msg_id = msg.msg_id };
        if (sendto(sock, &ack, sizeof(ack), 0,
                   (struct sockaddr*)&dst,
                   offsetof(struct sockaddr_un, sun_path) + strlen(dst.sun_path)) < 0) {
            perror("child sendto ACK"); exit(1);
        }
        if (msg.msg_id == MSG_COUNT) break;
    }

    close(sock);
    cleanup_socket(path);
    exit(0);
}
```

Рисунок 13. Функция дочернего процесса

```c
int main() {
    int sock;
    struct sockaddr_un parent_addr;
    pid_t pid;

    sock = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (sock < 0) { perror("parent socket"); exit(1); }

    cleanup_socket("/tmp/parent.sock");
    memset(&parent_addr, 0, sizeof(parent_addr));
    parent_addr.sun_family = AF_UNIX;
    strcpy(parent_addr.sun_path, "/tmp/parent.sock");
    if (bind(sock, (struct sockaddr*)&parent_addr,
             offsetof(struct sockaddr_un, sun_path) + strlen(parent_addr.sun_path)) < 0) {
        perror("parent bind"); exit(1);
    }

    struct timeval tv = { .tv_sec = TIMEOUT_SEC, .tv_usec = 0 };
    if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)) < 0) {
        perror("setsockopt SO_RCVTIMEO"); exit(1);
    }

    for (int i = 0; i < NCHILD; i++) {
        pid = fork();
        if (pid < 0) {
            perror("fork"); exit(1);
        } else if (pid == 0) {
            child_proc(i);
        }
    }

    sleep(1);

    Message msgs[MSG_COUNT] = {
        {1, "Hello, child!"},
        {2, "How are you?"},
        {3, "Goodbye!"}
    };
```

Рисунок 14. Создание сокетов и дочерних процессов.

```c
    struct sockaddr_un child_addr[NCHILD];
    socklen_t child_len[NCHILD];
    char path[64];
    for (int i = 0; i < NCHILD; i++) {
        memset(&child_addr[i], 0, sizeof(child_addr[i]));
        child_addr[i].sun_family = AF_UNIX;
        snprintf(path, sizeof(path), "/tmp/child%d.sock", i);
        strcpy(child_addr[i].sun_path, path);
        child_len[i] = offsetof(struct sockaddr_un, sun_path) + strlen(path);
    }

    for (int m = 0; m < MSG_COUNT; m++) {
        int acked[NCHILD] = {0}, remaining = NCHILD;

        for (int i = 0; i < NCHILD; i++) {
            if (sendto(sock, &msgs[m], sizeof(msgs[m]), 0,
                       (struct sockaddr*)&child_addr[i], child_len[i]) < 0) {
                perror("parent sendto"); exit(1);
            }
            printf("Parent sent msg #%d to child %d\n",
                   msgs[m].msg_id, i);
        }

        while (remaining > 0) {
            AckMessage ack;
            ssize_t r = recvfrom(sock, &ack, sizeof(ack), 0, NULL, NULL);
            if (r < 0) {
                if (errno == EAGAIN || errno == EWOULDBLOCK) {
                    printf("Timeout waiting ACK for msg #%d; retransmitting...\n",
                           msgs[m].msg_id);
                    for (int i = 0; i < NCHILD; i++) {
                        if (!acked[i]) {
                            sendto(sock, &msgs[m], sizeof(msgs[m]), 0,
                                   (struct sockaddr*)&child_addr[i], child_len[i]);
                            printf("Parent re-sent msg #%d to child %d\n",
                                   msgs[m].msg_id, i);
                        }
                    }
                    continue;
                } else if (errno == EINTR) {
                    continue;
                } else {
                    perror("parent recvfrom"); exit(1);
                }
            }
            if (ack.msg_id == msgs[m].msg_id && ack.child_id >= 0 && ack.child_id < NCHILD && !acked[ack.child_id]) {
                acked[ack.child_id] = 1;
                remaining--;
                printf("Parent received ACK for msg #%d from child %d\n",
                       ack.msg_id, ack.child_id);
            }
        }
    }
```

Рисунок 15. Отправка сообщений.

```c
    for (int i = 0; i < NCHILD; i++) wait(NULL);
    close(sock);
    cleanup_socket("/tmp/parent.sock");
    printf("Parent: all done.\n");
    return 0;
}
```

Рисунок 16. Завершение программы, очистка.

```
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$ gcc sockets.c -o sockets
sypoo@pop-os:~/mtuci/mtuci-os/labs/4$ ./sockets
Parent sent msg #1 to child 0
Parent sent msg #1 to child 1
Child 0 received msg #1: "Hello, child!"
Child 1 received msg #1: "Hello, child!"
Parent received ACK for msg #1 from child 0
Parent received ACK for msg #1 from child 1
Parent sent msg #2 to child 0
Parent sent msg #2 to child 1
Child 0 received msg #2: "How are you?"
Parent received ACK for msg #2 from child 0
Child 1 received msg #2: "How are you?"
Parent received ACK for msg #2 from child 1
Parent sent msg #3 to child 0
Child 0 received msg #3: "Goodbye!"
Parent sent msg #3 to child 1
Child 1 received msg #3: "Goodbye!"
Parent received ACK for msg #3 from child 0
Parent received ACK for msg #3 from child 1
Parent: all done.
```

Рисунок 17. Сборка и вывод программы

Заключение

Вывод: В ходе лабораторной работы были успешно реализованы три механизма межпроцессного взаимодействия (IPC) — сокеты, разделяемая память с семафорами и каналы с select(). Каждый метод показал свою эффективность: сокеты обеспечили гибкость, разделяемая память — высокую скорость, а каналы — простоту реализации. Наибольшую надежность продемонстрировал подход с сокетами благодаря встроенной поддержке таймаутов и переотправки, что делает его оптимальным для задач, требующих гарантированной доставки сообщений.