**ОБЩАЯ ТЕОРИЯ**

**LAB 7.2**

*2. ТСP - эхо сервер:*

*a. Сделайте TCP-сервер, который принимает соединения от клиентов на*

*заданном ip и port.*

*b. создает новый процесс, в котором:*

*i. читает данные от клиента;*

*ii. пересылает их ему обратно.*

**TCP (Transmission Control Protocol)** — это транспортный протокол сетевого стека TCP/IP, обеспечивающий надёжную, ориентированную на соединение передачу данных между двумя узлами (сервером и клиентом). Он используется в приложениях, где важна целостность и порядок данных, например, в веб-браузерах (HTTP/HTTPS), электронной почте (SMTP) и данном эхо-сервере.

**Ключевые характеристики TCP**:

1. Надёжность:

TCP гарантирует доставку всех данных без потерь, используя механизмы подтверждения (ACK) и повторной передачи.

Если пакет теряется или повреждается, TCP автоматически пересылает его.

2. Ориентированность на соединение:

Перед передачей данных TCP устанавливает соединение между клиентом и сервером с помощью трехэтапного рукопожатия (three-way handshake). Соединение остаётся активным до явного закрытия (четырёхэтапное завершение).

3. Управление потоком:

TCP использует скользящее окно (sliding window) для контроля объёма данных, отправляемых без подтверждения, предотвращая перегрузку сети или получателя.

4. Контроль ошибок:

TCP вычисляет контрольные суммы для проверки целостности данных. Пакеты, не прошедшие проверку, пересылаются.

5. Упорядоченная доставка:

Данные доставляются в том же порядке, в котором были отправлены, даже если пакеты приходят в разном порядке.

**Установление соединения (Three-Way Handshake)**

Чтобы начать передачу данных, клиент и сервер выполняют трехэтапное рукопожатие:

1. SYN: Клиент отправляет сегмент с флагом SYN (synchronize) и начальным порядковым номером (например, x).

2. SYN-ACK: Сервер отвечает сегментом с флагами SYN и ACK, своим порядковым номером (y) и подтверждением (x+1).

3. ACK: Клиент отправляет сегмент с флагом ACK, подтверждая (y+1).

После этого соединение установлено, и стороны могут обмениваться данными.

На уровне кода:

1. Сервер вызывает listen() для ожидания SYN от клиентов.

2. accept() завершает рукопожатие, возвращая новый сокет для клиента.

3. Клиент вызывает connect(), который инициирует SYN и завершает рукопожатие.

listen(server\_socket\_fd, BACKLOG)*listen()* — системный вызов, который переводит TCP-сокет в режим прослушивания, позволяя ему принимать входящие соединения от клиентов.

Указывает ядру, что сокет готов принимать TCP-соединения (переводит сокет в состояние LISTEN).

Определяет, сколько клиентов могут ждать обработки, пока сервер занят (управляется параметром backlog).

Без listen() сокет не может принимать входящие SYN-пакеты от клиентов, и вызов accept() будет некорректным.

accept(server\_socket\_fd, (struct sockaddr \*)&client\_addr, &client\_len)*accept()* — системный вызов, который принимает входящее TCP-соединение из очереди завершённых соединений, созданной listen(), и возвращает новый сокет для общения с клиентом.

Извлекает соединение из очереди завершённых соединений, завершая процесс установления TCP-соединения.

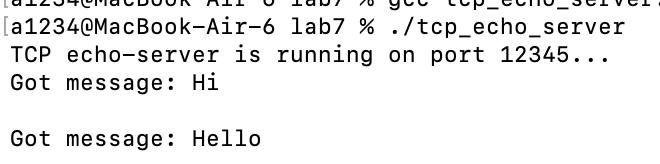
Создаёт новый сокет (client\_sock), который используется для общения с конкретным клиентом (чтение/запись данных). Этот сокет наследует характеристики серверного сокета (например, AF\_INET, SOCK\_STREAM), но привязан к конкретному соединению.

Заполняет client\_addr информацией о клиенте (например, 127.0.0.1:54321), что полезно для логирования или фильтрации.

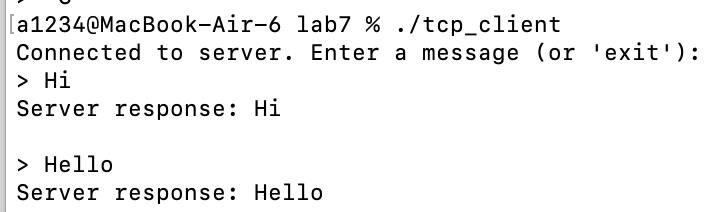
*c. Напишите TCP-клиента для проверки TCP-сервера.*

connect(socket\_fd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr))*connect()* устанавливает TCP-соединение с сервером. Инициирует трехэтапное рукопожатие. Связывает сокет клиента с сервером, переводя его в состояние ESTABLISHED для обмена данными.

**Server**

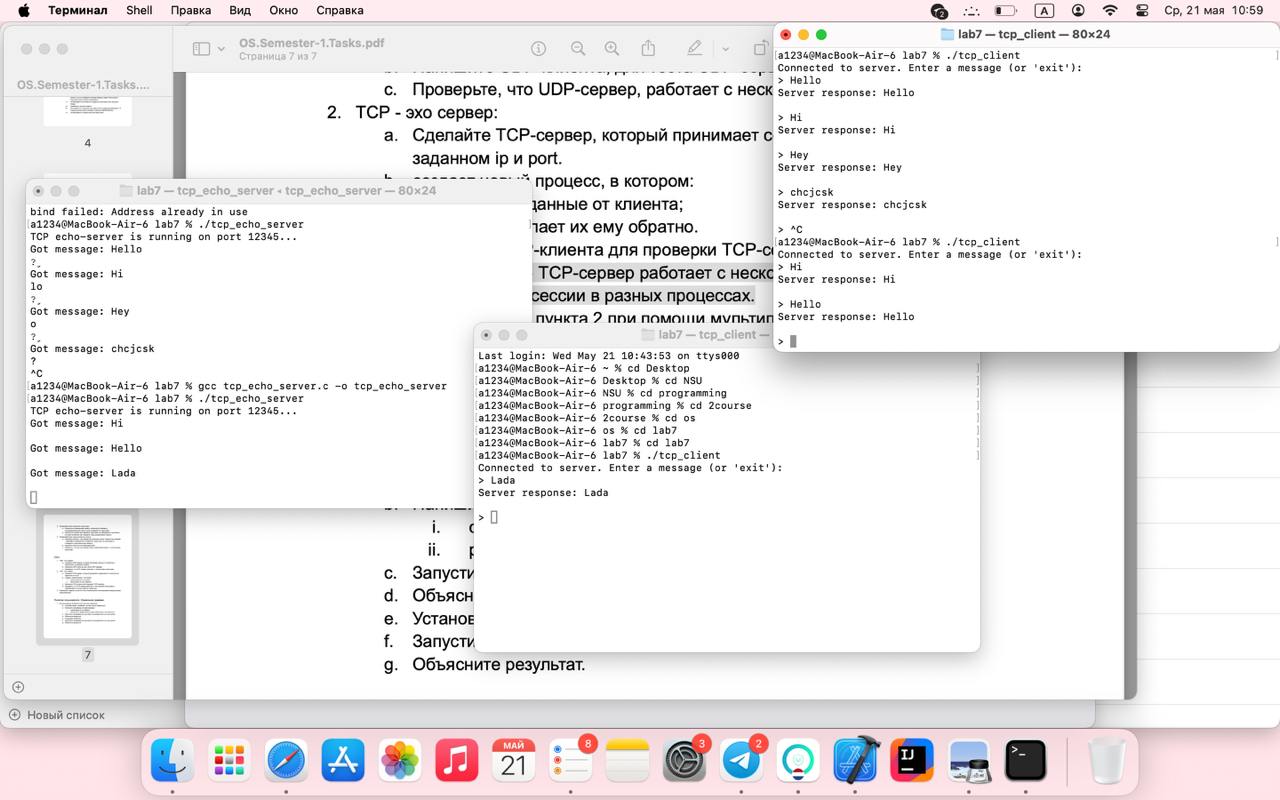
****

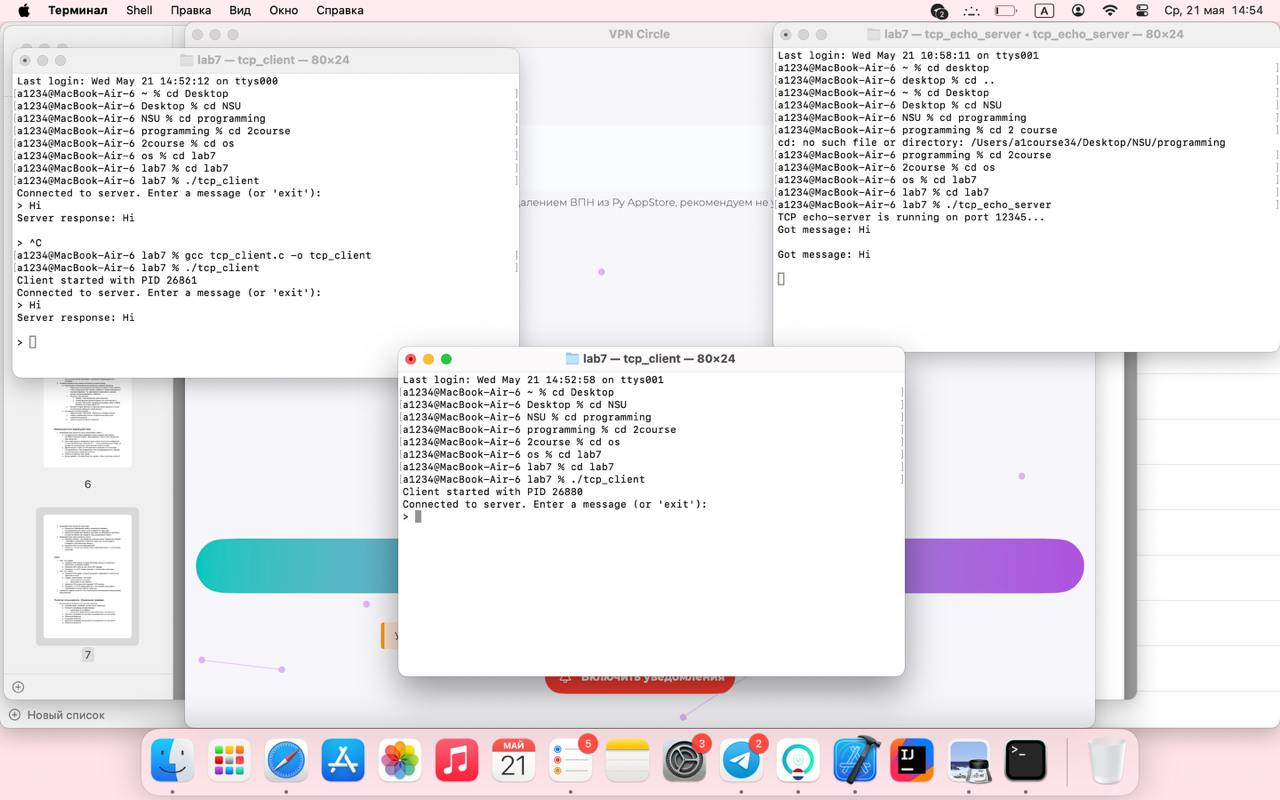
**Client**

****

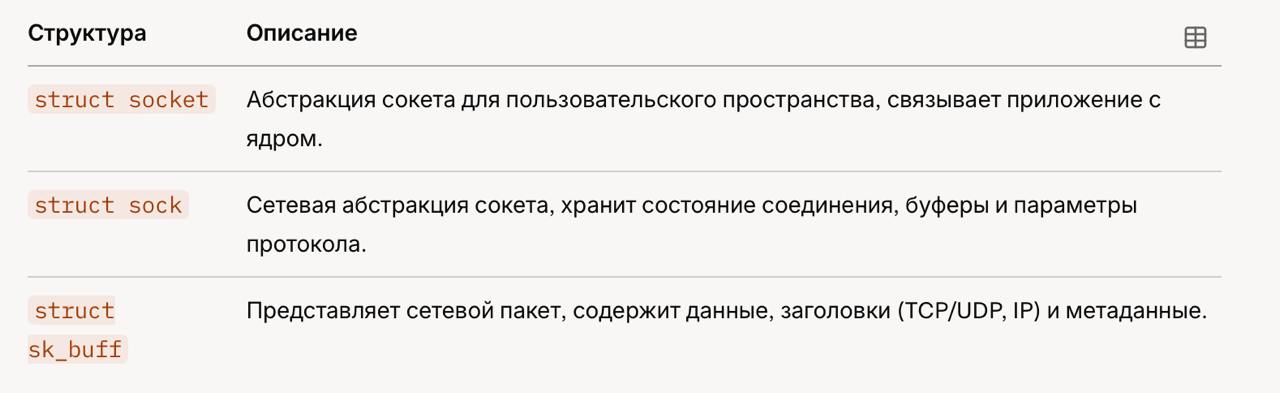
*d. Проверьте, что TCP-сервер работает с несколькими клиентами и*

*обрабатывает сессии в разных процессах.*

****

Добавлен вывод PID:  


КАК ВСЕ РАБОТАЕТ ВНУТРИ ОПЕРАЦИОННОЙ СИСТЕМЫ?



**struct socket:**

Создаётся при вызове socket(AF\_INET, SOCK\_STREAM, 0) (TCP) или SOCK\_DGRAM (UDP).

Хранит тип сокета, указатель на struct sock и функции операций (struct proto\_ops).

Интерфейс для системных вызовов (write, read).

**struct sock:**

Хранит:

IP-адреса и порты (например, 192.168.1.10:54321 → 192.168.1.20:12345).

Буферы отправки (sk\_write\_queue) и приёма (sk\_receive\_queue).

Состояние TCP (например, TCP\_ESTABLISHED).

Для UDP: минимальная информация, так как нет соединения.

**struct sk\_buff:**

Хранит:

Данные (например, "Hello").

Заголовки (TCP/UDP, IP, Ethernet).

Метаданные (интерфейс, временные метки).

Формирует двусвязный список для обработки пакетов

**Системный вызов write для TCP и UDP**

**1.** **TCP**

write(socket\_fd, buffer, strlen(buffer)); // Отправляем "Hello"

1. Системный вызов:

Вызов write(socket\_fd, "Hello", 5) передаёт управление ядру через sys\_write.

Процессор выполняет инструкцию syscall, переключаясь в режим ядра:

2. Слой сокетов:

Ядро находит struct socket по socket\_fd.

Проверяет тип сокета: SOCK\_STREAM

Проверяет, что сокет в состоянии ESTABLISHED (после connect).

Вызывает sock\_sendmsg, который передаёт данные в struct sock.

3. Транспортный слой (TCP):

Копирует данные ("Hello") в sk\_write\_queue через copy\_from\_user.

Создаёт struct sk\_buff:

Данные: "Hello" (5 байт).

TCP-заголовок: порты (54321 → 12345), номер сегмента, флаги.

Проверяет окно передачи (sliding window) и состояние соединения.

Вычисляет контрольную сумму.

4. Сетевой слой (IP):

Получает sk\_buff

Добавляет IP-заголовок: источник, назначение.

Протокол: IPPROTO\_TCP (6) или IPPROTO\_UDP (17).

Выполняет маршрутизацию через ip\_route\_output\_flow: Проверяет таблицу маршрутизации (ip\_route\_output).

Определяет исходящий интерфейс (например, eth0).

Фрагментирует пакет, если он превышает MTU (максимальный размер пакета, обычно 1500 байт). Для "Hello" фрагментация не нужна.

Вычисляет IP-контрольную сумму.

5. Канальный слой:

Добавляет Ethernet-заголовок (MAC-адреса).

Передаёт sk\_buff драйверу сетевой карты.

Драйвер использует DMA для передачи данных в кольцевой буфер карты.

6. Отправка:

Сетевая карта отправляет пакет по сети.

TCP ждёт ACK от сервера, используя таймер повторной передачи (tcp\_retransmit\_timer).

7. Возврат:

Ядро возвращает количество записанных байт (5) или ошибку.

**2. UDP**

sendto(socket\_fd, buffer, strlen(buffer), 0, (struct sockaddr \*)&server\_addr, sizeof(server\_addr));

1. Системный вызов:

Вызов sendto(socket\_fd, "Hello", 5, ...) вызывает sys\_sendto.

2. Слой сокетов:

Ядро находит struct socket и вызывает sock\_sendmsg.

Проверяет адрес сервера (server\_addr).

3. Транспортный слой (UDP):

Копирует данные в struct sk\_buff.

Добавляет UDP-заголовок: порты (54321 → 12345), длина, контрольная сумма (опционально).

Нет состояния соединения, данные сразу передаются вниз.

4. Сетевой и канальный слои:

Аналогично TCP: IP-заголовок, Ethernet-заголовок, DMA, отправка.

5. Отправка:

Пакет отправляется без ожидания подтверждения.

6. Возврат:

Ядро возвращает количество байт или ошибку.

**Системный вызов read для TCP и UDP**

**1. TCP**

n = read(client\_socket\_fd, buffer, BUFFER\_SIZE - 1);

1. Приём пакета:

Сетевая карта сервера (192.168.1.20) получает Ethernet-кадр.

Использует DMA для копирования данных в кольцевой буфер ядра.

Вызывает прерывание, уведомляя ядро о новом пакете.

Драйвер извлекает struct sk\_buff.

2. Канальный слой:

Проверяет Ethernet-заголовок, передаёт sk\_buff в IP-слой.

3. Сетевой слой:

Проверяет IP-заголовок, контрольную сумму.

Протокол (IPPROTO\_TCP или IPPROTO\_UDP).

Удаляет IP-заголовок, передаёт sk\_buff в транспортный слой.

Если пакет фрагментирован, собирает его (ip\_defrag).

4. Транспортный слой (TCP):

Проверяет порты, номера сегментов, контрольную сумму.

Помещает данные в sk\_receive\_queue.

Отправляет ACK клиенту (tcp\_send\_ack).

Если сегменты не в порядке, ядро хранит их до сборки.

5. Системный вызов:

Вызов read(client\_socket\_fd, buffer, BUFFER\_SIZE - 1) вызывает sys\_read.

Ядро проверяет struct socket по client\_socket\_fd.

Ядро проверяет sk\_receive\_queue.

Если данные есть, копирует их в buffer (copy\_to\_user).

Если данных нет, процесс блокируется (wait\_event), пока данные не придут.

6. Возврат:

Ядро возвращает количество байт (n = 5) или 0 (если соединение закрыто).

**2. UDP**

n = recvfrom(socket\_fd, buffer, BUFFER\_SIZE - 1, 0, NULL, NULL);

1. Приём пакета:

Аналогично TCP: DMA, прерывание, sk\_buff.

2. Канальный и сетевой слои:

Проверяют Ethernet и IP-заголовки.

3. Транспортный слой (UDP):

Проверяет порты, контрольную сумму (если есть).

Помещает датаграмму в sk\_receive\_queue.

4. Системный вызов:

recvfrom вызывает sys\_recvfrom, копирует датаграмму в buffer.

Если данных нет, возвращает ошибку или блокируется.

5. Возврат:

Возвращает количество байт или 0.

**Передача между компьютерами**

1. Клиент:

write создаёт sk\_buff, TCP/UDP добавляет заголовки, IP маршрутизирует, сетевая карта отправляет.

2. Сеть:

Пакет проходит через маршрутизаторы, используя IP-адреса.

3. Сервер:

Сетевая карта получает пакет, ядро обрабатывает через стек, данные доходят до read.

Роль процессора

1. Системные вызовы:

Инструкция syscall переключает процессор в режим ядра.

Регистры сохраняются, ядро выполняет код (например, tcp\_sendmsg).

2. DMA:

Сетевая карта использует DMA для передачи данных, минимизируя нагрузку на CPU.

3. Прерывания:

Приём пакета вызывает прерывание (NET\_RX\_SOFTIRQ), процессор обрабатывает его в ядре.