«Разработка многопоточных программ»

Оглавление

Список сокращений

ОС – Операционная Система

IDE – Integrated Development Environment

IPC – Inter Process Commutation

PID – Process ID

PPID – Parent Process ID

FIFO – First Input First Output

1. Введение

Потоки в ОС являются закономерным развитием концепции процессов. Они позволяют создавать производительные программные комплексы со сложной внутренней логикой информационного обмена. Одной из значимых областей применения многопоточности в ОС *Linux* является сетевой обмен данными через механизм сокетов.

Целью данной лабораторной работы является освоение навыка разработки многопоточных программ на языке *C* для сетевого обмена данными. В качестве средства разработки используется *IDE* с возможностью удаленного подключения к лабораторному стенду.

2. Концепция потоков в ОС

2.1. Понятие потока

Концепция потоков в ОС заключается в возможности разделения процесса на ряд «подпроцессов», выполняемых квазипараллельно. Каждый «подпроцесс» принято называть «потоком» или «нитью» («*thread*»). При порождении процесса ОС создает для него только один поток, который в дальнейшем с помощью системных вызов может создавать дополнительные потоки внутри данного процесса [1]. То есть каждый процесс состоит как минимум из одного потока, а поток считается наименьшей единицей обработки информации, исполнение которой может быть запущено ОС.

Несмотря на крайнюю схожесть процессов и потоков, между ними существует ключевое различие – обращение к памяти. Память разделена между процессами и каждый процесс может обращаться только к своему участку, а потоками в рамках одного процесса обращаются к единой области памяти. Иллюстрация приведена на Рисунке Рисунок 1.

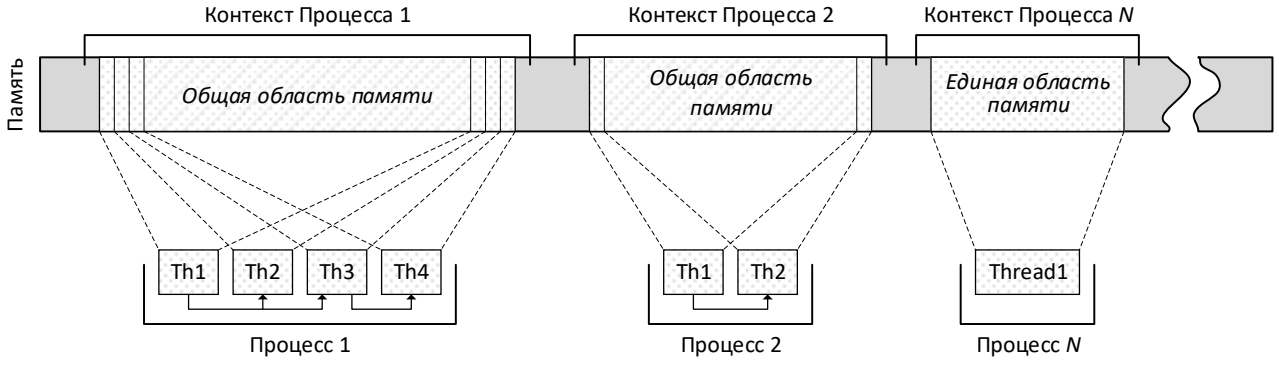

Рисунок 1. Обращение потоков к памяти

За счет работы в едином адресном пространстве потоки могут напрямую обращаться к одним и тем же глобальным переменным, массивам данным, дескрипторам файлов, потокам ввода-вывода и т. д. Однако для квазипараллельного

исполнения и управления потоками ОС *Linux* частично выделяет память под контекст потока, в котором хранятся состояние счетчика команд, значения регистров, стек, обработчики сигналов и т. д.

Существует несколько преимуществ в применении потоков вместо процессов. Во-первых, потоки создаются гораздо быстрее за счет отсутствия необходимости полного выделения нового участка памяти для контекста. Во-вторых, работа потоков в едином адресном пространстве с одними и теме же данным исключает потребность в пересылке данных, что так же экономит время. В-третьих, с помощью потоков возможен запуск фоновой обработки данных в рамках единого процесса. Например, в одном потоке реализован интерфейс пользователя, который ожидает поступления новых данных, а в другом потоке обрабатываются ранее поступившие данные.

Однако обращение к одним и тем же данным несколькими потоками имеет недостаток: для корректной работы между потоками требуется их синхронизация при обращении в одни и те же участки памяти. Отсутствие синхронизации может привести к порче данных. Например, если в один и тот же момент времени в одном потоке происходит запись в массив, а в другом потоке чтение, то поток-читатель получит часть массива со старыми данными, а часть с новыми. Чтобы этого избежать поток-писатель не должен иметь возможности обратиться к массиву во время чтения, а поток-читатель – во время записи. Операции, при которых только один поток имеет право доступа к общему ресурсу называются «атомарными» [2].

2.2. Особенности организации потоков в ОС *Linux*

Несмотря на одинаковую трактовку концепции потоков во всех ОС, их конкретная реализация сильно разнится. Например, в ОС *Windows* процесс воспринимается как некоторый контейнер для потоков, а в ее ядре существуют специальные механизмы поддержки многопоточности.

В ОС *Linux* каждый поток воспринимается исключительно как процесс, который использует совместные ресурсы с другим процессом. В ядре ОС *Linux* за диспетчеризацию процессов и потоков отвечают одни и те же механизмы, поэтому для ОС *Linux* часто вместо понятия «поток» применяют понятие «легковесный процесс», то есть процесс, для которого не требуется полностью создавать новый контекст.

3. Разработка многопоточных программ

3.1. Системные вызовы

В ОС *Linux* легковесные процессы запускаются через системный вызов `clone`:

```
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

Он также, как и «`fork`», создает новый процесс, однако его возможности несколько шире. В отличии от «`fork`», «`clone`» вместо запуска копии процесса с места вызова, запускает на исполнение переданную в аргументе по указателю функцию «`int`

`(*fn)(void *)` с аргументами `«void *arg»`. Аргумент `«void *child_stack»` определяет область памяти для стека процесса. А с помощью аргумента `«int flags»` определяются порядок наследования процессов и те данные, которые будут между ними разделены.

На практике системный вызов `«clone»` крайне редко применяется самостоятельно. Вместо `«clone»` используют стандартизированные библиотечные вызовы на его основе.

3.2. Библиотечные вызовы

Системный *API* для управления, синхронизации и планирования потоков полностью определен в стандарте *POSIX.1c*. В ОС *Linux* библиотечные вызовы, соответствующие данному стандарту, реализованы в библиотеке «*libpthread*». Для подключения данной библиотеки к программе необходимо указать компилятору опцию `«-lpthread»`.

Все вызовы, описанные в «*libpthread*», можно разделить на вызовы для управления потоками и вызовы для синхронизации данных.

3.2.1. Управление потоками

Основными библиотечными вызовами для управления потоками являются:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *),
                                                                                    void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
int pthread_detach(pthread_t thread);
```

Вызов `«pthread_create»` запускает в новом потоке функцию `«void *(*start_routine) (void *)»` с аргументами `«void *arg»`. Аргумент `«const pthread_attr_t *attr»` служит для передачи специфических атрибутов потоку и может быть заменен на `«NULL»`. Если поток был запущен успешно, то возвращаемое значение `«int»` равно `«0»`, а в аргумент `«void *thread»` записывается дескриптор потока. Если во время запуска потока произошла ошибка, то в возвращаемом значении будет код ошибки, а значение `«void *thread»` будет не определено.

Вызов `«pthread_exit»` завершает тот поток, в котором он был использован (аналог `«exit»` для процесса). В аргумент `«void *retval»` записывается код возврата функции потока.

Вызов `«pthread_join»` предназначен для ожидания родительским потоком завершения дочернего потока. В аргумент `«pthread_t thread»` передается дескриптор дочернего потока, а в аргумент `«void **retval»` дочерний поток записывает возвращаемое значение. В случае успеха `«pthread_join»` возвращает `«0»`, в случае ошибки возвращаемое значение содержит ее код.

Вызов `«pthread_detach»` открепляет дочерний поток с дескриптором `«pthread_t thread»` от родительского. После открепления дочерний поток может быть завершен без вызова `«pthread_ join»` в родительском потоке. В случае успеха `«pthread_detach»` возвращает `«0»`, в случае ошибки возвращаемое значение содержит код ошибки.

Рассмотрим простой пример исходного кода, создающего новый поток (Листинг 1).

Листинг 1 – Пример создания нового потока

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

void* threadFunc (void* thread_data) {

	printf("threadFunc was started\n");
	pthread_exit(0);

}

int main () {

	printf("main was started\n");

	void* thread_data = NULL;
	pthread_t thread;

	if (pthread_create(&thread, NULL, threadFunc, thread_data) != 0) {
		fprintf(stderr, "error: pthread_create was failed\n");
		exit(-1);
	}

	if (pthread_join(thread, NULL) != 0) {
		fprintf(stderr, "error: pthread_create was failed\n");
		exit(-1);
	}

	printf("main and threadFunc was finished\n");

	return 0;

}
```

Функция «main» выводит информационное сообщение о старте своей работы и создает дочерний поток с помощью вызова «pthread_create», в котором выполняется функция «threadFunc». Функция «threadFunc» выводит в stdout информационное сообщение и завершается с помощью «pthread_exit». Функция «main» дожидается завершения функции «threadFunc» в системном вызове «pthread_join», после чего выводит информационное сообщение об окончании работы. Переменная «thread_data» показана в качестве примера передачи аргументов в функцию при создании нового потока.

3.2.2. Синхронизация данных

Стандарт *POSIX* определяет несколько механизмов синхронизации потоков: мьютексы, переменные условия, блокировки чтение-запись, спин-блокировки, барьеры и т. д. Их применение в ОС *Linux* осуществляется через специальные библиотечные вызовы, в которых реализованы атомарные операции доступа. В рамках данного лабораторного практикума будут рассмотрены только мьютексы ввиду их наиболее частого применения.

Мьютекс представляет собой некоторую бинарную переменную
pthread_mutex_t mutex

внутренняя структура которой скрыта от пользователя, а ее основные значения ассоциированы с ее состоянием: мьютекс либо «захвачен», либо «свободен».

Каждый поток перед началом работы с общими данными пытается захватить мьютекс (по умолчанию мьютекс свободен). Если поток захватил свободный мьютекс, то он держит его до тех пор, пока не закончит все необходимые операции, а потом освобождает. Другие потоки, при попытке захватить уже захваченный мьютекс, приостанавливают свою работу до тех пор, пока мьютекс не освободится. В результате, в каждый момент времени мьютексом либо не владеет ни один поток (если никто не обращается к данным), либо владеет только один, а остальные потоки либо ждут своей очереди на получение мьютекса, либо работают с другими данными.

Помимо состояния, мьютекс содержит в себе информацию о своем типе, потоке-владельце и т. д. Основные типы мьютекса:

- `PTHREAD_MUTEX_NORMAL` – повторный захват мьютекса тем же потоком не проверяется и поток блокируется;
- `PTHREAD_MUTEX_RECURSIVE` – повторный захват мьютекса тем же потоком допустим, введется счет повторных захватов;
- `PTHREAD_MUTEX_ERRORCHECK` – повторный захват мьютекса тем же потоком возвращает ошибку.

По умолчанию мьютексу назначается состояние `PTHREAD_MUTEX_DEFAULT`, которое ОС может интерпретировать как одно из вышеперечисленных состояний.

Для работы с мьютексами существуют следующие библиотечные вызовы:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Вызов «`pthread_mutex_init`» инициализирует начальное состояние мьютекса «`pthread_mutex_t *restrict mutex`» в состояние «свободен» с возможностью добавления атрибутов «`const pthread_mutexattr_t *restrict attr`» (тип мьютекса и т. д.). Если установка атрибутов не требуется, то вместо данного аргумента можно указать «`NULL`». Помимо вызова «`pthread_mutex_init`» мьютекс так же может быть статически инициализирован при объявлении

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

где «`PTHREAD_MUTEX_INITIALIZER`» – макрос, объявленный в «*pthread.h*».

Вызов «`pthread_mutex_destroy`» деинициализирует мьютекс «`pthread_mutex_t *mutex`». Вызов «`pthread_mutex_lock`» позволяет потоку захватить мьютекс «`pthread_mutex_t *mutex`». Вызов «`pthread_mutex_trylock`» проверяет возможность захвата мьютекса «`pthread_mutex_t *mutex`». Если мьютекс получилось захватить, то в возвращаемом значении типа «`int`» данного вызова будет «`0`», в противном случае – код ошибки. Вызов «`pthread_mutex_unlock`» позволяет потоку освободить мьютекс «`pthread_mutex_t *mutex`».

Все вызовы для работы с мьютексами возвращают значение типа «`int`», которое равно «`0`» в случае успеха или содержат в случае ошибки ее код.

Дополним пример из Листинга 1 получением информации из `stdin` в основном потоке и ее выводом в `stdout` в дочернем потоке (Листинг 2).

Листинг 2 – Пример работы с мьютексами

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
char shared_data[128] = { '\0' };

void* threadFunc (void* thread_data) {

	printf("threadFunc was started\n");

	while (1) {
		pthread_mutex_lock(&mutex);
		if (shared_data[0] != '\0') {
			printf("threadFunc: %s", shared_data);
			shared_data[0] = '\0';
		}
		pthread_mutex_unlock(&mutex);
	}
	pthread_exit(0);

}

int main () {

	printf("main was started\n");

	void* thread_data = NULL;
	pthread_t thread;

	if (pthread_create(&thread, NULL, threadFunc, thread_data) != 0) {
		fprintf(stderr, "error: pthread_create was failed\n");
		exit(-1);
	}

	char tmp[128] = { '\0' };
	while (fgets(tmp, sizeof(tmp), stdin)) {
		pthread_mutex_lock(&mutex);
		for (int i = 0; i < sizeof(shared_data); i++) shared_data[i] = tmp[i];
		pthread_mutex_unlock(&mutex);
	}

	if (pthread_join(thread, NULL) != 0) {
		fprintf(stderr, "error: pthread_create was failed\n");
		exit(-1);
	}

	printf("main and threadFunc was finished\n");

	return 0;

}
```

Для обмена данными между потоками объявлены две глобальные переменные: мьютекс «`pthread_mutex_t mutex`», который инициализируется макросом «`PTHREAD_MUTEX_INITIALIZER`» и общий массив данных «`char shared_data[256]`», первый элемент которого инициализируется символом окончания строки. В основном потоке (функция

«main») полученные из stdin символы с помощью «fgets» записываются во временный массив «tmp». Перед копированием символов из «tmp» в общие данные «shared_data» поток «main» захватывает мьютекс, а после копирования – освобождает.

В дочернем потоке (функция «threadFunc») в бесконечном цикле «while» перед работой с общими данными захватывается мьютекс. Если в «shared_data» находится строка ненулевой длины, то она выводится в stdout, а в нулевой элемент «shared_data» записывается «0», чтобы избежать бесконечной отправки одних и тех же данных. После работы с «shared_data» мьютекс высвобождается.

Приведенный выше пример является одним из часто используемых паттернов организации потоков: в одном потоке с помощью системных вызовов реализуется ожидание данных от пользователя (других процессов, сетевых подключений и т. д.), а в другом потоке реализуется обработка ранее поступивших данных. В результате при обработке данных взаимодействие с процессом не блокируется, а обработка данных не блокируется ожиданием возврата из системных вызовов приема данных.

4. Сетевой обмен данными в ОС *Linux*

4.1. О принципах сетевого обмена в ОС *Linux*

Неотъемлемой частью ОС *Linux* является хорошо проработанная поддержка взаимодействия с другими вычислительными системами по сети (например, через *Ethernet* или *Wi-Fi*). На сетевом обмене данными базируется множество ключевых особенностей ОС *Linux*: удаленное подключение терминала по *SSH*, система репозитариев с ПО, удаленный доступ к файловой системе и т. д.

Основным механизмом для организации сетевого обмена в ОС *Linux* является «*сокет*» (Рисунок Рисунок 2). В ядре ОС он представляет собой драйверы для аппаратного обеспечения сетевого обмена данными (*Ethernet PHY*) с одной стороны, и системный *API* (системные вызовы) для получения запросов от процессов с другой стороны.

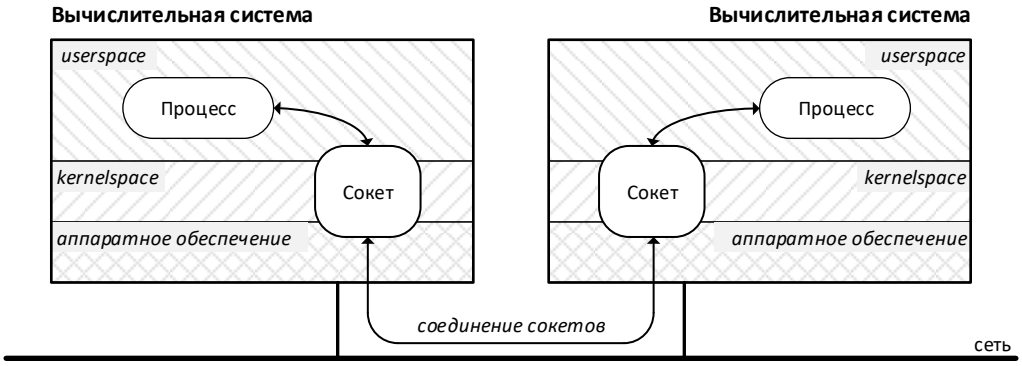

Рисунок 2. Взаимодействие процессов через сокеты

Системное *API* в ОС *Linux* для приемо-передачи данных через сокеты максимально соответствует общей идеологии работы со всеми сущностями вычислительной системы, как с файлами. Однако при работе с сокетами существуют

дополнительные нюансы, связанные с внутренней организацией сетевого стека проколов.

4.2. Сетевой стек протоколов

Для корректного приема данных по сети, помимо самих данных, необходимо наличие дополнительной информации: какое устройство является отправителем, какое устройство является получателем, какое количество данных было отправлено и т. д. Так же на линии передачи могут произойти сбои, искажающие информацию, поэтому часто требуются дополнительная служебная информация (контрольная сумма, бит четности и т.п.). Совокупность отправляемых данных (полезной нагрузки) и всей необходимой дополнительной информации называется «*пакетом*». А конкретный набор правил, определяющих порядок отправки, расположение полей служебной информации и полезной нагрузки в пакете, выбранных средств защиты от ошибок и т. д. называется «*протоколом*».

Для организации сетевого обмена используют «*стек протоколов*»: полезная нагрузка одного пакета, сформированного по одному протоколу, представляет собой пакет, сформированный по другому протоколу, полезная нагрузка которого представляет собой следующий пакет и т. д. Отправляемые данные содержатся в самом вложенном пакете.

В соответствии с моделью *OSI* все протоколы можно разделить на семь уровней, каждый из которых берет на себя определенную функцию в приемо-передаче данных. На Рисунок Рисунок 3 обозначены наиболее популярные фактически используемые протоколы в сетевом взаимодействии и их соответствие модели *OSI*. Физический, канальный и сетевой уровни реализуются с помощью протоколов *Ethernet* (*Wi-Fi*) и *IP*. Наиболее популярным протоколом транспортного уровня является протокол *TCP* (совместно такой стек именуется *TCP/IP*). Вместо *TCP* может быть применен *UDP* протокол.

Рисунок 3. Модель OSI, применяемые сетевые протоколы и их реализация в ОС Linux

Сеансовый уровень, уровень представления и прикладной уровень модели *OSI* реализуют конечную функцию приемо-передачи данных: это может быть протокол

HTTP для реализации web-приложений, *SMTP* для электронной почти, *SSH* для удаленного подключения и т. д.

В ОС *Linux* четыре нижние уровня модели *OSI* реализованы на уровне ядра и аппаратного обеспечения, а три верхних уровня реализуются в пользовательских приложениях.

4.3. Настройка сетевых интерфейсов

Аппаратное обеспечение вычислительной системы может иметь в своем составе несколько физических сетевых интерфейсов (3 нижних уровня модели *OSI*). В ОС *Linux* для просмотра и настройки сетевых подключений используется утилита

```
ifconfig
```

После ее вызова в командной оболочке появится информация о включенных и поддерживаемых в ОС сетевых интерфейсах. Например,

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.36  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::3863:a4e5:cb42:4858  prefixlen 64  scopeid 0x20<link>
        ether dc:a6:32:8b:6b:6d  txqueuelen 1000  (Ethernet)
        RX packets 4915  bytes 577746 (564.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3608  bytes 4078753 (3.8 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Для просмотра всех интерфейсов, в том числе и выключенных, можно воспользоваться опцией *«-a»*:

```
ifconfig -a
```

Каждый сетевой интерфейс имеет свое название. В приведенном примере в ОС определено и включено два сетевых интерфейса с названиями «*eth0*» и «*lo*». Интерфейс *eth0* является физически существующем интерфейсом, а *lo* – интерфейс, эмулируемый ОС для служебных целей. Перед использованием интерфейса в нем должны быть произведены настройки *IP*-протокола. Если интерфейс уже настроен (как в приведенном выше примере), то для него заданы: *IP*-адрес (*inet*, адрес устройства в протоколе *IP*), маска подсети (*netmask*) и опционально широковещательный адрес (*broadcast*) и адрес для *IPv6* (*inet6*), если данный протокол используется в стеке.

Как правило, в сети присутствует устройство, которое самостоятельно настраивает интерфейс для вновь подключившихся абонентов с помощью протокола динамической настройки *DHCP*, и ручной ввод параметров не требуется. Однако при его отсутствии ручной ввод параметров *IP*-протокола можно осуществить с помощью команды:

```
ifconfig <ИМЯ ИНТЕРФЕЙСА> inet <IP-дрес> netmask <МАСКА ПОДСЕТИ>
```

Для подключенного и настроенного интерфейса утилита *ifconfig* отображает статистику по количеству принятых (*TX*) и переданных (*RX*) пакетов и информацию о количестве возникших ошибках при приеме и передаче.

4.4. Алгоритмы работы с сокетами

На Рисунке Рисунок 4 схематично изображены алгоритмы работы с сокетами для двух вычислительных систем: «*сервера*» и «*клиента*». Они взаимодействуют через сетевой стек *TCP/IP*. Клиент-серверное взаимодействие, принятое для данного сетевого стека, определяет, что в актах обмена информацией инициатором всегда выступает *клиент*, который посылает *серверу* запросы и получает от *сервера* ответы. *Сервер*, в свою очередь, при отсутствии запроса, данные не отправляет. На каждом шаге алгоритма указаны используемые системные вызовы ОС *Linux*.

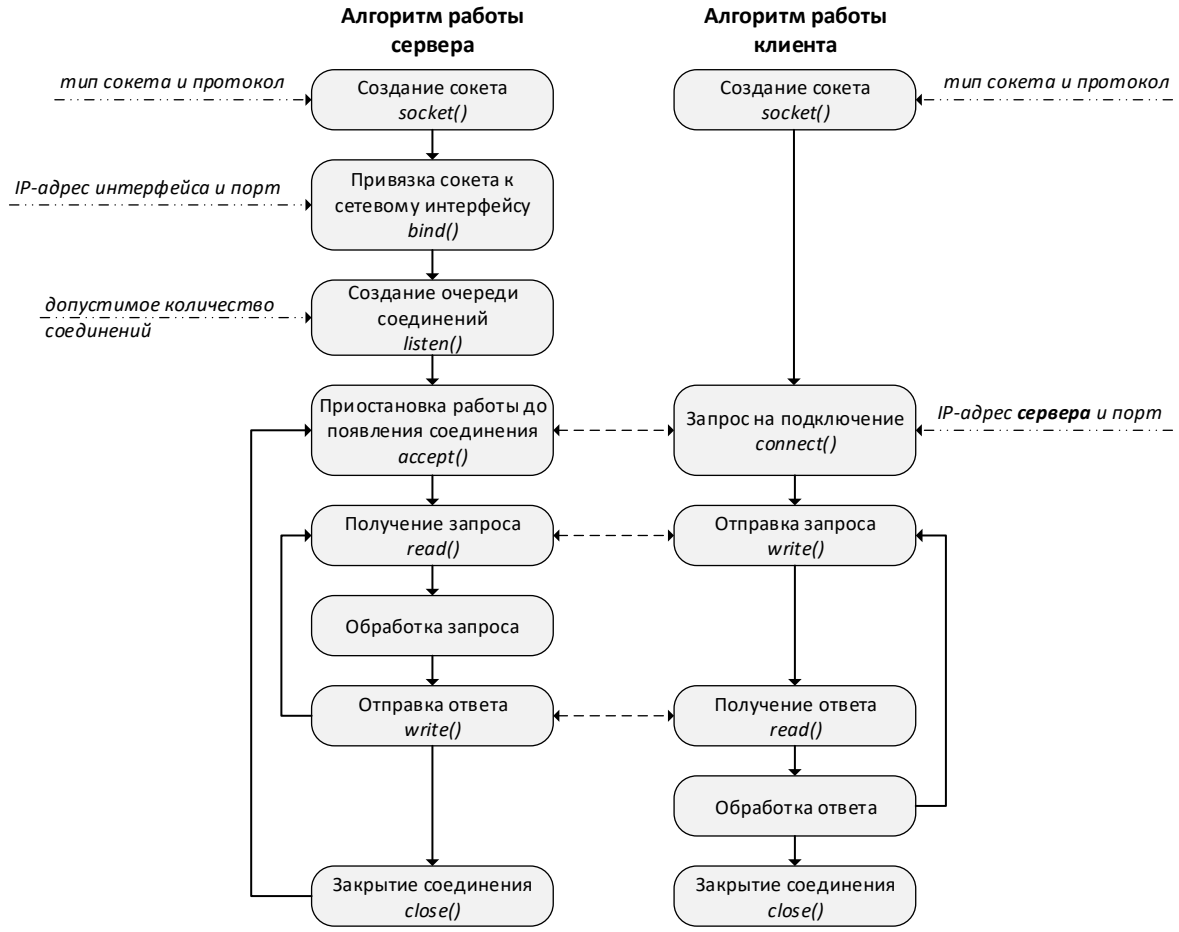

Рисунок 4. Алгоритмы работы сервера и клиента через TCP/IP

Рассмотрим алгоритм работы *сервера*. На первом шаге он создает новый сокет с помощью системного вызова «`socket`», задает его тип и стек протокол. Затем с помощью системного вызова «`bind`» созданный сокет привязывается к определенному сетевому интерфейсу, присутствующему в системе, указывая *IP*-адреса данного интерфейса. Затем сервер создает очередь подключений для клиентов системным вызовом «`listen`»,

указывая максимально допустимое число открытых соединений. После создания очереди сервер уходит в ожидание подключения клиента в системном вызове «accept». После подключения *клиента сервер* с помощью системного вызова «read» получает запрос от клиента, обрабатывает его, и выдает ответ с помощью системного вызова «write». После отправки ответа сервер либо возвращается к ожиданию нового запроса, либо закрывает соединение с помощью системного вызова «close». После закрытия соединения сервер либо переходит в состояние ожидания нового подключения, либо завершает свою работу.

Рассмотрим алгоритм работы *клиента*. На первом шаге он создает новый сокет с помощью системного вызова «socket», задает его тип и стек протоколов. После создания сокета *клиент* использует системный вызов «connect» для установления соединения с сервером, указывая его *IP*-адрес и порт. После установки соединения с помощью системных вызовов «write» и «read» клиент отправляет запросы и получает ответы от сервера. После завершения обмена данными клиент закрывает соединение с помощью системного вызова «close».

В блоках «Обработка запроса» и «Обработка ответа» сервер и клиент реализуют логику для работы либо по одному из существующих протоколов верхнего уровня (*HTTP*, *SSH*, *FTP* и др.), либо могут реализовать собственный протокол.

4.5. Системные вызовы

Системный вызов «*socket*» имеет следующую семантику:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Его возвращаемое значение типа «int» является дескриптором сокета с помощью которого ведется вся дальнейшая работа. Входной аргумент «int domain» выбирает набор из доступных протоколов, которые будут участвовать в соединении, среди них:

- PF_UNIX – для коммуникации локальных процессов;
- PF_INET – для коммуникации по протоколам *TCP/IP* или *UDP*;

Входной аргумент «int type» определяет тип коммуникации через сокет, наиболее часто применяются следующие значения:

- SOCK_STREAM – создает двухстороннее соединение для обмена данными (как правило, *TCP/IP*);
- SOCK_DGRAM – обмен данными происходит без предварительного соединения, размеры пакетов фиксированы (как правило, *UDP*).

Входной аргумент «int protocol» представляет собой расширение типа коммуникации и применяется для уточнения конкретного протокола. Однако обычно каждый тип коммуникации имеет только один протокол и данный аргумент устанавливается в «0».

Если при использовании системного вызова «socket» произошла ошибка, то возвращаемое значение будет равно «-1».

Системный вызов «bind» привязывает созданный сокет к существующему в системе сетевому интерфейсу:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Первым аргументом «int sockfd» является дескриптор сокета, второй аргумент «struct sockaddr *my_addr» представляет собой указатель на структуру для определения сетевого интерфейса. Она имеет следующие поля для настройки:

```
struct sockaddr_in {
                sa_family_t    sin_family;    // Семейство протоколов (AF_INET для IP)
                in_port_t      sin_port;      // 16-битный номер порта в сетевом порядке байт
                struct in_addr sin_addr;      // IP-адрес сетевого интерфейса
};
```

В качестве IP-адреса сетевого интерфейса «struct in_addr sin_addr» допускается указывать директиву INADDR_ANY, который привяжет сокет сразу ко всем доступным сетевым интерфейсам.

Аргумент «socklen_t addrlen» вызова «bind» должен равняться размеру структуры «struct sockaddr *my_addr» байтах. В случае успеха «bind» возвращаемое значение типа «int» равно «0», в случае ошибки – «-1», а код ошибки установится в переменную «errno».

Системный вызов «listen» имеет следующую семантику:

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Его входной аргумент «int sockfd» является дескриптором сокета, а аргумент «int backlog» – количеством максимально допустимых подключений. В случае успеха возвращаемое значение типа «int» равно «0», в случае ошибки – «-1», а код ошибки установится в переменную «errno».

Системный вызов «accept»:

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);
```

позволяет принять подключение клиента к сокету. В случае успешного подключения он возвращает дескриптор соединения типа «int», с помощью которого осуществляется прием (системный вызов «read») и передача (системный вызов «write») данных. В случае ошибки возвращаемое значение равно «-1», а код ошибки установится в переменную «errno». Входной аргумент «int sockfd» является дескриптором сокета, аргумент «struct sockaddr *restrict addr» предназначен для возврата описания адреса клиента. Если оно не требуется, то устанавливается «NULL». Аргумент «socklen_t *restrict addrlen» предназначен для указания размера «struct sockaddr *restrict addr». Если «struct sockaddr *restrict addr» задано как «NULL», то он тоже может указывать на «NULL».

Системный вызов «connect» инициирует соединение на сокете:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

Входной аргумент «int sockfd» является дескриптором сокета, входной указатель «const struct sockaddr *serv_addr» ссылается на структуру, в котором описаны параметры сервера, а аргумент «socklen_t addrlen» указывает на размеры этой структуры. В случае успеха возвращаемое значение типа «int» равно «0». В случае ошибки возвращаемое значение равно «-1», а код ошибки установится в переменную «errno».

Помимо системного вызова «write» для отправки данных в сокет могут использоваться следующие специализированные системные вызовы:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *msg, size_t len, int flags);
ssize_t sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

Системный вызов «send» может использоваться только если соединение установлено, а «sendto» и «sendmsg» в любое время. Входной аргумент «int s» должен являться дескриптором сокета, «const void *msg» – указатель на массив с отправляемыми данными, а «size_t len» – количество отправляемых данных в байтах. С помощью аргумента «int flags» определяются опции и режимы отправки.

Помимо системного вызова «read» данные из сокета могут быть приняты с помощью системных вызовов»

```
#include<sys/types.h>
#include <sys/socket.h>

int recv(int s, void *buf, size_t len, int flags);
int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
int recvmsg(int s, struct msghdr *msg, int flags);
```

Входной аргумент «int s» должен являться дескриптором сокета, «void *buf» – указатель на массив для приема данных, а «size_t len» – максимальное количество принимаемых данных в байтах. С помощью аргумента «int flags» определяются опции и режимы отправки.

4.6. Вспомогательные библиотечные вызовы

Для преобразования *IP*-адреса из строкового формата в формат, пригодный для восприятия системными вызовами, используются библиотечный вызов

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

Он преобразует строку символов «const char *src» в сетевой адрес типа «int af» и записывает по указателю «void *dst». Тип «int af» принимает следующие значения:

- AF_INET – для *IPv4*, строка в формате «ddd.ddd.ddd.ddd»;
- AF_INET6 – для *IPv6*.

В случае успеха «inet_pton» возвращает положительное значение. В случае ошибки преобразования возвращает «0» (неправильный формат «src») или отрицательное значение, если неправильно указан тип «af».

Для преобразований, связанных с разным представлением данных в памяти и при отправке, существуют следующие библиотечные вызовы:

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Вызов «htonl» осуществляет перевод целого *long*-числа «uint32_t hostlong» из порядка *little-endian* в *big-endian*. Вызов «htons» осуществляет перевод целого *short*-числа «uint16_t hostshort» из порядка *little-endian* в *big-endian*. Вызов «ntohl» осуществляет перевод целого *long*-числа «uint32_t netlong» из порядка *big-endian* в *little-endian*. Вызов «ntohs» осуществляет перевод целого *short*-числа «uint16_t netlong» из порядка *big-endian* в *little-endian*.

4.7. Пример программ сервера и клиента

В качестве примера программ, использующих сетевое взаимодействие через стек *TCP/IP* рассмотрим исходный код клиента и «эхо-сервера» – сервера, отправляющим в ответ тоже сообщение, что и было получено.

В Листинге 3 приведен исходный код *эхо-сервера*:

Листинг 3 – Исходный код эхо-сервера

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#define IP     INADDR_ANY
#define PORT   10050

int main() {

        // create a socket:
        int lfd = socket(AF_INET, SOCK_STREAM, 0);
        if (lfd == -1) {
                fprintf(stderr, "error: cannot create a socket. Sorry\n");
                exit (-1);
        }

        // bind socket to interface:
        struct sockaddr_in servaddr;
        memset(&servaddr, 0, sizeof(servaddr));
        servaddr.sin_family           = PF_INET;
        servaddr.sin_addr.s_addr      = htonl(IP);
        servaddr.sin_port             = htons(PORT);

        if (bind(lfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
                fprintf(stderr, "error: cannot bind socket to interface. Sorry\n");
                exit (-1);
        }

        // create a queue for one client:
        if (listen(lfd, 1) == -1) {
                fprintf(stderr, "error: cannot create a queue (system call listen). Sorry\n");
                exit (-1);
        }

        // forever loop for client accept:
        while (1) {

                // wait for client:
                int cfd = accept(lfd, NULL, NULL);
                if (cfd == -1) {
                        fprintf(stderr, "error: cannot accept client. Sorry\n");
                        exit (-1);
```

```
48.                 }
49.
50.                 // loop for data exchange (echo mode):
51.                 char buf[128];
52.                 int n;
53.                 do {
54.                         n = read(cfd, buf, sizeof(buf));      // wait request
55.                         write(cfd, buf, n);                   // send response:
56.                 } while (n > 0);
57.
58.                 close (cfd);           // close client connection
59.
60.         }
61.
62. }
```

Вначале в функции «main» происходит создание сокета с помощью системного вызова «socket», дескриптор сокета записывается в переменную «lfd». При создании сокет настраивается на работу через стек *TCP/IP* с помощью передачи в аргументах AF_INET и SOCK_STREAM. Полученный дескриптор сокета привязывается к любому сетевому интерфейсу, доступному в ОС, поскольку указываемый *IP*-адрес определен директивой INADDR_ANY. Порт подключения установлен как «10050». Это произвольное значение, для реальных протоколов прикладного уровня значение порта, как правило, определено протоколом: для *HTTP* портом по умолчанию является «80», для *SSH* – «22» и т. д. После привязки сокета к любому сетевому интерфейсу с помощью системного вызова «listen» создается очередь подключений. Поскольку пример является демонстрационным, максимальный размер очереди установлен в одного клиента.

В бесконечном цикле в «main» используется системный вызов «accept», в котором процесс находится до тех пор, пока не поступит запрос на подключение от клиента. Когда запрос поступил, «accept» возвращает дескриптор клиента в переменную «cfd». В цикле «do { … } while» сперва ожидается получение запроса от клиента, которое копируется в массив «buf» с помощью системного вызова «read», а затем данные из «buf» передаются обратно клиенту с помощью вызова «write».

В Листинге 4 представлен исходный код клиента:

Листинг 4 – Исходный код клиента

```
1.  #include <arpa/inet.h>
2.  #include <netinet/in.h>
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.  #include <string.h>
6.  #include <sys/socket.h>
7.  #include <sys/types.h>
8.  #include <unistd.h>
9.
10. int main(int argc, char *argv[]) {
11.
12.         // get IP and port from args:
13.         if (argc < 2) {
14.                 printf("call: ./tcpclient IP PORT");
15.                 exit(-1);
16.         }
17.         char *IP       = argv[1];
18.
```

```
19.         char *port     = argv[2];
20.
21.         //create data structure about server:
22.         struct sockaddr_in servaddr;
23.         memset(&servaddr, 0, sizeof(servaddr));
24.         servaddr.sin_family    = PF_INET;
25.         servaddr.sin_port              = htons(atoi(port));
26.
27.         if(inet_pton(PF_INET, IP, &servaddr.sin_addr) <= 0) {
28.                 fprintf(stderr, "error: bad IP address\n");
29.                 exit (-1);
30.         }
31.
32.         // create a socket:
33.         int sfd = socket(PF_INET, SOCK_STREAM, 0);
34.         if (sfd == -1) {
35.                 fprintf(stderr, "error: cannot create a socket. Sorry\n");
36.                 exit (-1);
37.         }
38.
39.         // connect to server:
40.         if (connect(sfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
41.                 fprintf(stderr, "error: cannot connect to server %s:%s\n", IP, port);
42.                 exit (-1);
43.         }
44.
45.         // loop for data exchange:
46.         char buf[128];
47.         while (1) {
48.
49.                 // wait data from user:
50.                 fgets(buf, sizeof(buf), stdin);
51.
52.                 // send data to server:
53.                 if (write(sfd, buf, strlen(buf)) < 0) {
54.                         fprintf(stderr, "error: cannot send data to server %s:%s\n", IP, port);
55.                         exit (-1);
56.                 }
57.
58.                 // wait server answer:
59.                 int n = read(sfd, buf, sizeof(buf));
60.                 if ((n == 2) && (buf[0] == 'q')) break;
61.
62.                 // send data to user:
63.                 printf("server answer: %s", buf);
64.
65.         }
66.
67.         close(sfd);    // close socket
68.
69. }
```

Вначале в функции «main» происходит получение *IP*-адреса и порта подключения сервера из входных аргументов. Затем данные о сервере заносятся в структуру «servaddr». Затем создается сокет с помощью системного вызова «socket», дескриптор сокета записывается в переменную «sfd». Дескриптор сокета «sfd» и структура «servaddr» передаются в системный вызов «connect» для подключения к серверу.

После подключения к серверу в бесконечном цикле вначале происходит ожидание данных из stdin, затем эти данные отправляются на сервер с помощью системного вызова «write», затем ожидается ответ сервера с помощью системного вызова «read» и полученный ответ отправляется в stdout. Если по stdin была получена строка «q\n», то цикл прерывается и сокет закрывается системным вызовом «close».

5. Интеграция *IDE* в процесс разработки программ

5.1. Назначение *IDE*

В предыдущих лабораторных работах был продемонстрирован подход к удаленной работе со встраиваемой системой через терминал пользователя. Для этого использовалась программа *PuTTY*, с помощью которой через интерфейс *Ethernet* по протоколу *SSH* производилось подключение к командной оболочке ОС целевого устройства. Такой подход позволил получить полное управление над ОС, разрабатывать и запускать *bash*-скрипты и пользовательские программы.

С ростом функциональной сложности разрабатываемых программ растет и количество файлов, и объем исходного кода в них. Появляется все большее количество алгоритмов, переменных и функций, которые необходимо помнить и отслеживать. Разработка приложений в командной оболочке с помощью текстового редактора становится весьма трудоемким процессом, требующим предельной концентрации и внимания. Значительно повышается вероятность ошибок в исходных кодах программы как в части синтаксиса, так и в части логики поведения программы.

Для облегчения процесса разработки целесообразно перейти от текстового редактора в командной оболочке к *IDE* с графическим интерфейсом. Современные *IDE* имеют множество функций, облегчающих процесс разработки: подсветка синтаксиса, авто дополнение имен переменных и функций, графически представленное дерево зависимостей файлов исходных кодов, автоматически создаваемый «*Makefile*», наглядная пошаговая отладка, интеграция с системами контроля версий и т. д.

5.2. Способы интеграции *IDE*

Можно выделить несколько способов интеграции *IDE* в задачу разработки программ для встраиваемых систем под управлением ОС *Linux*:

- использовать систему *VNC* (*Virtual Network Computing*);
- использовать кросс-компиляцию;
- использовать удаленную разработку.

Система *VNC* основывается на сетевом клиент-серверном протоколе *RFB* (*Remote Frame Buffer*) и, по сути, представляет собой захват экрана подключенной целевой платформы. Серверная программа и *IDE* устанавливаются на целевую платформу, а клиентское приложение на ПК. Система *VNC* требует поддержку графической оболочки и дополнительного объема памяти для установки *IDE*.

Кросс-компиляция – это процесс компиляции исходного кода программы на одной платформе с целью последующего запуска результирующего исполняемого файла на другой платформе. При кросс-компиляции *IDE* и дополнительные библиотеки устанавливаются на ПК, а результирующий файл передается по сети на целевую платформу. Такой подход удобен, когда разработка ПО невозможна на целевой платформе или разработка требуется под множество целевых платформ.

Способ удаленной разработки базируется на протоколе *SSH*. *IDE* устанавливается на машину-клиент, а на целевой платформе должны быть установлены программные инструменты компиляции и отладки. При удаленной разработке на *IDE* возлагается только роль удобного графического интерфейса, помогающему разработчику, а вся непосредственная работа над программой ведется полностью на целевой платформе.

5.3. Apache *NetBeans IDE*

Для разработки многопоточных программ в лабораторной работе будет использоваться способ удаленной разработки с помощью *IDE Apache NetBeans 12.4*. Данная *IDE* является свободно распространяемой интегрируемой средой разработки приложений.

Проект *NetBeans IDE* активно поддерживается компанией *Oracle*, однако разработка ведется независимым сообществом разработчиков *Apache NetBeans Community* [3]. Среда разработана на языке *Java*, а значит требует для своей работы установленную виртуальную машину *Java* [4], но за счет этого является кроссплатформенной, то есть может быть запущена на множестве ОС: *Microsoft Windows, Linux, MacOS, Solaris* и других. Для установки и запуска среды возможно использовать инсталляторы или *portable*-версию. Полная инструкция по работе с *Apache NetBeans IDE* представлена в [5].

6. Упражнения

6.1. В соответствии с п. 4.1 Лабораторной работы №1 подключитесь к лабораторному стенду.

6.2. Создайте поддиректорию «*lab4*» в директории со своим именем и фамилией.

6.3. Разработка многопоточной программы

6.3.1. В качестве примера многопоточной программы, работающей с сокетами, поэтапно разработаем простейший веб-сервер, который будет получать данные из *stdin* и отображать их на веб-странице, и получать данные от веб-страницы и выводить их в *stdout*.

6.3.2. Создание потоков

6.3.2.1. Откройте *NetBeans IDE* и подключитесь к *Raspberry Pi*.

6.3.2.2. Создайте новый проект «Services» – «Узлы сборки на C/C++» – «pi@IP» – «Новый проект C/C++» – «C/C++» – «Приложение на C/C++».

6.3.2.3. Задайте имя проекта «webserver», расположение файла «/home/pi/IVT31_Ivanov_Ivan/lab4/»

6.3.2.4. Создадим каркас из двух функций: «main» и «server». Каждая функция будет работать в отдельном потоке

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

void *server(void *arg) {
```

```
    // print info:
    fprintf(stderr, "server was started\r\n");

    /* ТУТ БУДЕТ СЕРВЕР */

    // print info:
    fprintf(stderr, "server was finished\r\n");

}

int main() {

    // print info:
    fprintf(stderr, "main was started\n");

    // create new thread for server:
    pthread_t thread;
    if (pthread_create(&thread, NULL, server, NULL) != 0) {
        fprintf(stderr, "error: pthread_create was failed\r\n");
        exit(-1);
    }

    /* ТУТ БУДЕТ ПОЛУЧЕНИЯ ДАННЫХ ИЗ STDIN */


    // wait server thread exit:
    pthread_join(thread, NULL);

    // print info:
    fprintf(stderr, "main was finished\r\n");

    return (0);

}
```

В функции «main» с помощью «pthread_create» создается поток, выполняющий функцию «server», дескриптор потока передается в «thread» по ссылке. В конце функции «main» в вызове «pthread_join» ожидается завершения дочернего потока. Каждая функция пока что только выводит информационные сообщения о старте и завершении.

6.3.2.5. Добавьте в опции компилятора флаг для работы с потоками: ПКМ по проекту – *Properties* – Компилятор C – Дополнительные параметры – «-lpthread» и то же самое для конфигурации «*Release*» (переключение вверху всплывающего окна).

6.3.2.6. Скомпилируйте проект: Run – Build Project (F11)

6.3.2.7. Запустите проект Run – Run Project (F6)

6.3.2.8. В окне «Output» (выполнение) появятся сообщения:

```
main was started
server was started
server was finished
main was finished
```

Сначала запустился основной поток main и создал дочерний поток server, затем дочерний поток server завершился и завершился основной поток main.

6.3.2.9. Добавьте результаты в репозиторий git (ПКМ по проекту – «Удаленный репозиторий git» – «Фиксация…») с комментарием:

```
a skeleton of project was created
```

Не забудьте указать автора при коммите.

6.3.3. Обмен данными между потоками

6.3.3.1. Добавим тестовый обмен данными между потоками. Создадим буфер памяти и в потоке `main` будем записывать в него тестовую строку, а в потоке `server` считывать и выводить в `stdout`.

6.3.3.2. Перед описанием функций добавим подключение «`string.h`» для использования библиотечных вызовов по работе со строками, добавим директиву, в которой укажем размер буфера и создадим структуру для буфера с двумя полями: указатель на буфер и его размер в байтах:

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

// size of buffer (bytes) for data exchange between main and server:
#define SHAREDBUF_SIZE 1024

// shared buffer for data exchange between main and server:
struct st_shbuf {
    char            *buf;
    ssize_t         bufsize;
};
….
….
```

6.3.3.3. Доработаем функцию «`server`»: добавим в нее прием аргумента со структурой для буфера и бесконечный цикл, в котором будем проверять наличие данных в буфере, выводить содержимое буфера на экран и очищать буфер:

```
void *server(void *shbuf) {

    // print info:
    fprintf(stderr, "server was started\r\n");

    // get arg:
    struct st_shbuf *sharedbuf = (struct st_shbuf *) shbuf;        // преобразуем тип
аргумента

    // forever loop:
    while (1) {
        if (strlen(sharedbuf->buf) != 0) {                         // если в буфере есть
данные
            fprintf(stderr, "%s\r\n", sharedbuf->buf);            // выводим их на экран
            memset(sharedbuf->buf, 0, sharedbuf->bufsize);        // и очищаем буфер
        }
    }

    // print info:
    fprintf(stderr, "server was finished\r\n");

}
```

6.3.3.4. Доработаем функцию `main`. Добавим в нее инициализацию памяти под буфер, освобождение памяти под буфер перед завершением, передачу буфера в качестве аргумента в поток server при создании и бесконечный цикл, в который будет записываться тестовая строка:

```
int main() {

    // print info:
    fprintf(stderr, "main was started\r\n");

    // create shared buffer for data from stdin between main and server:
    struct st_shbuf *sharedbuf = malloc(sizeof(struct st_shbuf));       // создаем в куче
    sharedbuf->bufsize  = SHAREDBUF_SIZE;                              // запоминаем размер
буфера
```

```
            sharedbuf->buf      = malloc(sharedbuf->bufsize);              // память в куче под
буфер
            memset(sharedbuf->buf, 0, sharedbuf->bufsize);              // заливаем буфер нулями

            // create new thread for server:
            pthread_t thread;
            if (pthread_create(&thread, NULL, server, (void *)sharedbuf) != 0) {
                fprintf(stderr, "error: pthread_create was failed\r\n");
                exit(-1);
            }

            // forever loop:
            char tststr[] = "This is a test string between threads";
            while (1) {
                for (int i=0; i < strlen(tststr); i++) sharedbuf->buf[i] = tststr[i]; // запись
                usleep(1000000/4);                                          // снижение частоты
записи
            }

            // wait server thread exit:
            pthread_join(thread, NULL);

            // print info:
            fprintf(stderr, "main was finished\r\n");
            free(sharedbuf->buf);                                       // освобождаем память
            free(sharedbuf);                                            // освобождаем память

            return (0);

        }
```

6.3.3.5. Компилируем (F11) и запускаем программу (F6). После запуска передаваемая через буфер строка будет выводиться неполностью:

```
This is a test string be
This is a test
This is a test string between threa
This is a test string between threads
This is a test string between threads
This is a test string between
This is a test st
This is a test str
This is a test strin
```

Это происходит из-за отсутствия синхронизации между потоками при доступе к «sharedbuf»: данные считываются во время записи.

6.3.3.6. Сделаем синхронизацию потоков для доступа к «sharedbuf» через мьютекс.

6.3.3.7. Для этого добавим поле с указателем на мьютекс в структуру «struct st_shbuf»:

```
// shared buffer for data exchange between main and server:
 struct st_shbuf {
    char            *buf;
    ssize_t          bufsize;
    pthread_mutex_t *mutex;
};
```

6.3.3.8. В функции main добавим инициализацию мьютекса:

```
// create shared buffer for data from stdin between main and server:
struct st_shbuf *sharedbuf = malloc(sizeof(struct st_shbuf));  // создаем в куче
sharedbuf->bufsize   = SHAREDBUF_SIZE;                              // запоминаем размер
буфера
sharedbuf->buf      = malloc(sharedbuf->bufsize);             // память в куче под буфер
memset(sharedbuf->buf, 0, sharedbuf->bufsize);                 // заливаем буфер нулями
sharedbuf->mutex    = malloc(sizeof(pthread_mutex_t));         // память под мьютекс
pthread_mutex_init(sharedbuf->mutex, NULL);                   // инициализация мьютекса
```

и освобождение памяти под мьютекс при завершении потока «main»:

```
// print info:
fprintf(stderr, "main was finished\r\n");
free(sharedbuf->buf);                           // освобождаем память
free(sharedbuf->mutex);                         // освобождаем память
```

```
free(sharedbuf);                                    // освобождаем память
```

а в бесконечном цикле добавим захват мьютекса до обращения к общему буферу и освобождение после:

```
// forever loop:
char tststr[] = "This is a test string between threads";
while (1) {
    pthread_mutex_lock(sharedbuf->mutex);
    for (int i=0; i < strlen(tststr); i++) sharedbuf->buf[i] = tststr[i];
    pthread_mutex_unlock(sharedbuf->mutex);
    usleep(1000000/4);                                  // снижение частоты записи
}
```

6.3.3.9. В функции `server` так же добавим захват мьютекса до обращения к общему буферу и высвобождение после:

```
// forever loop:
while (1) {
    pthread_mutex_lock(sharedbuf->mutex);                   // захватили мьютекс
    if (strlen(sharedbuf->buf) != 0) {                      // если в буфере есть данные
        fprintf(stderr, "%s\r\n", sharedbuf->buf);          // выводим их на экран
        memset(sharedbuf->buf, 0, sharedbuf->bufsize);      // и очищаем буфер
    }
    pthread_mutex_unlock(sharedbuf->mutex);                 // освободили мьютекс
}
```

6.3.3.10. Скомпилируем (F11) и запустим программу (F11):

```
main was started
server was started
This is a test string between threads
This is a test string between threads
This is a test string between threads
This is a test string between threads
This is a test string between threads
This is a test string between threads
…
```

теперь в один момент времени к буферу «sharedbuf» обращается только один из потоков и данные не считываются во время записи.

6.3.3.11. Добавьте код в репозиторий git (ПКМ по проекту – «Удаленный репозиторий git» – «Фиксация…») с комментарием:

```
data transfer from main thread to server thread was added
```

6.3.4. Ввод данных из stdin

6.3.4.1. Добавим в функцию `main` вместо отправки тестовой строки получение данных из `stdin` и их последовательную запись в буфер. Для этого изменим код внутри цикла `while`:

```
…
        // forever loop:
        char indata[STDIN_DATA_SIZE];

        while (1) {

            // 3.1. get data from stdin:
            fgets(indata, sizeof(indata), stdin);

            // 3.2. push to buffer:
            pthread_mutex_lock(sharedbuf->mutex);
            if ((sharedbuf->bufsize - strlen(sharedbuf->buf) - 1) > strlen(indata)) {
                strcat(sharedbuf->buf, indata);
            }
            pthread_mutex_unlock(sharedbuf->mutex);

            // 3.3. clear input buf:
            memset(indata, 0, sizeof(indata));

        }
```

…

Данные, получаемые из `stdin` с помощью `fgets` будут записываться в промежуточный буфер `indata`, затем дописываться в общий с `server` буфер, а затем промежуточный буфер `indata` будет обнуляться. При копировании данных из `indata` в `sharedbuf` проверяется оставшийся размер в буфере. Так же добавим директиву для размера локального буфера:

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

// maximum bytes from stdin:
#define STDIN_DATA_SIZE 128
…
```

6.3.4.2. Скомпилируйте (F11) и запустите программу в терминале:

```
./webserver/dist/Debug/GNU-Linux/webserver
```

Убедитесь, что отправляемые с клавиатуры данные выводятся на экран.

6.3.4.3. Добавьте код в репозиторий git с комментарием:

```
transfer data from stdin to server was added
```

6.3.5. Разработка сервера

6.3.5.1. Перенесем код функции `server` в отдельный файл «server.c», структуру для буфера в «server.h», а директивны с размерами массивов в отдельный файл «settings.h». Файлы будут иметь следующий вид:

settings.h:

```
#ifndef SETTINGS_H
#define SETTINGS_H

// maximum bytes from stdin:
#define STDIN_DATA_SIZE 128

// size of buffer (bytes) for data exchange between main and server:
#define SHAREDBUF_SIZE 1024

#endif /* SETTINGS_H */
```

server.h:

```
#ifndef SERVER_H
#define SERVER_H

#include <pthread.h>

// shared buffer for data exchange between main and server:
struct st_shbuf {
    char            *buf;
    ssize_t         bufsize;
    pthread_mutex_t *mutex;
};

extern void *server(void *shbuf);

#endif /* SERVER_H */
```

server.c:

```
#include <stdio.h>
#include <string.h>
#include "server.h"

void *server(void *shbuf) {

    // print info:
    fprintf(stderr, "server was started\r\n");
```

```
            // get arg:
            struct st_shbuf *sharedbuf = (struct st_shbuf *) shbuf;        // преобразуем тип
аргумента

            // forever loop:
            while (1) {
                pthread_mutex_lock(sharedbuf->mutex);                    // захватили мьютекс
                if (strlen(sharedbuf->buf) != 0) {                        // если в буфере есть
данные
                    fprintf(stderr, "%s\r\n", sharedbuf->buf);           // выводим их на экран
                    memset(sharedbuf->buf, 0, sharedbuf->bufsize);       // и очищаем буфер
                }
                pthread_mutex_unlock(sharedbuf->mutex);                  // освободили мьютекс
            }

            // print info:
            fprintf(stderr, "server was finished\r\n");

        }
```

main.c:

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include <unistd.h>
        #include <pthread.h>
        #include "settings.h"
        #include "server.h"

        int main() {

            // print info:
            fprintf(stderr, "main was started\r\n");

            // create shared buffer for data from stdin between main and server:
            struct st_shbuf *sharedbuf = malloc(sizeof(struct st_shbuf));   // создаем в куче
            sharedbuf->bufsize  = SHAREDBUF_SIZE;                            // запоминаем размер
буфера
            sharedbuf->buf      = malloc(sharedbuf->bufsize);                // память в куче под
буфер
            memset(sharedbuf->buf, 0, sharedbuf->bufsize);                      // заливаем буфер
нулями
            sharedbuf->mutex    = malloc(sizeof(pthread_mutex_t));          // память под мьютекс
            pthread_mutex_init(sharedbuf->mutex, NULL);                          // инициализация
мьютекса

            // create new thread for server:
            pthread_t thread;
            if (pthread_create(&thread, NULL, server, (void *)sharedbuf) != 0) {
                fprintf(stderr, "error: pthread_create was failed\r\n");
                exit(-1);
            }

            // forever loop:
            char indata[STDIN_DATA_SIZE];

            while (1) {

                // 3.1. get data from stdin:
                fgets(indata, sizeof(indata), stdin);

                // 3.2. push to buffer:
                pthread_mutex_lock(sharedbuf->mutex);
                if ((sharedbuf->bufsize - strlen(sharedbuf->buf) - 1) > strlen(indata)) {
                    strcat(sharedbuf->buf, indata);
                }
                pthread_mutex_unlock(sharedbuf->mutex);

                // 3.3. clear input buf:
                memset(indata, 0, sizeof(indata));

            }
```

```
    // wait server thread exit:
    pthread_join(thread, NULL);

    // print info:
    fprintf(stderr, "main was finished\r\n");
    free(sharedbuf->buf);                                   // освобождаем
память
    free(sharedbuf->mutex);                                 // освобождаем
память
    free(sharedbuf);                                        // освобождаем
память

    return (0);

}
```

6.3.5.2. Добавьте созданные файлы в *git* с комментарием

```
server.c/.h for server functions was added
settings.h for settings was added
```

6.3.5.3. Скомпилируйте (F11) и запустите программу в терминале, убедитесь, что работоспособность сохранилась.

6.3.5.4. Добавим в *server.c* функцию инициализации сервера в соответствии с шагами 1 – 3 Рисунка Рисунок 4:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include "server.h"
#include "settings.h"

int server_init (char *ip, int port, int numofclients);

void *server(void *shbuf) {
        …
}

int server_init (char *ip, int port, int numofclients) {

    // create a socket:
    int lfd = socket(AF_INET, SOCK_STREAM, 0);
    if (lfd == -1) {
        SERVER_ERROR("cannot create a socket");
    }

    // bind socket to interface:
    int anyip = (in_addr_t *) ip == (in_addr_t *)INADDR_ANY;
    struct sockaddr_in servaddr;
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family       = PF_INET;
    servaddr.sin_addr.s_addr  = anyip? INADDR_ANY : inet_addr(ip);
    servaddr.sin_port         = htons(port);

    if (bind(lfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
        SERVER_ERROR ("cannot bind socket to interface with IP: %s", anyip? "any" : ip);
        exit(-1);
    }

    // create a queue for clients:
    if (listen(lfd, numofclients) == -1) {
        SERVER_ERROR("cannot create a queue (system call listen)");
    }

    fprintf(stderr, "server was started IP: %s, PORT: %d\n", ip, port);
```

```
    return lfd;

}
```

Функция «server_init» принимает на вход *IP*-адрес интерфейса, к которому будет привязан сокет, порт и максимальное количество подключенных клиентов. Возвращаемое значение является дескриптором сокета.

6.3.5.5. Добавим в *settings.h* настройки сервера:

```
// server settings:
#define SRV_INFO        1           // print info msgs to stderr (0 - dis, 1 - en)
#define IP              INADDR_ANY
#define PORT            80
#define WEBBUF_SIZE     1024*32     // size of buffer for data of client
```

6.3.5.6. Добавим в *server.h* две директивы для вывода сообщений от сервера

```
// macro for error events:
#define SERVER_ERROR(fmt, ...) do { fprintf(stderr, "server error: " fmt "\n", ##__VA_ARGS__); exit(-1); } while (0)

// macro for info messages:
#define SERVER_INFO(fmt, ...) do { if (SRV_INFO) fprintf(stderr, "server info: " fmt "\n", ##__VA_ARGS__); } while (0)
```

6.3.5.7. Скомпилируйте проект и убедитесь, что он не содержит ошибок.

6.3.5.8. Добавим в функцию server инициализацию сокета и обмен данными с клиентом в соответствии с шагами 4 – 7 Рисунка Рисунок 4:

```
void *server(void *shbuf) {

    // print info:
    fprintf(stderr, "server was started\r\n");

    // get arg:
    struct st_shbuf *sharedbuf = (struct st_shbuf *) shbuf;         // преобразуем тип аргумента

    // server initialization:
    int lfd = server_init(IP, PORT, 1);

    // forever loop for client accept:
    while (1) {

        // wait for client:
        int cfd = accept(lfd, NULL, NULL);
        if (cfd == -1) {
            fprintf(stderr, "error: cannot accept client. Sorry\n");
            exit (-1);
        } else {
            SERVER_INFO("client was connected");
        }

        // loop for data exchange (echo mode):
        char *request   = malloc(WEBBUF_SIZE);
        char *response  = malloc(WEBBUF_SIZE);

        while (1) {

            // recieve request from server:
            memset(request, 0, WEBBUF_SIZE);
            if (recv(cfd, request, WEBBUF_SIZE, 0) == -1) {
                SERVER_INFO("connection was closed");
                break;
            } else {
                SERVER_INFO("client request:\n%s", request);
            }

            // parse request:
            memset(response, 0, WEBBUF_SIZE);
            /* ТУТ БУДЕТ ОБРАБОТКА ЗАПРОСА */
```

```
                // send answer:
                if (send(cfd, response, strlen(response), 0) == -1) {
                    SERVER_INFO("connection was closed");
                    break;
                } else {
                    SERVER_INFO("server response:\n%s", response);
                }

            }

            // close client connection:
            free (request);
            free (response);
            close (cfd);

        }

        // print info:
        fprintf(stderr, "server was finished\r\n");

    }
```

В первом бесконечном цикле вызывается функция `accept`, ожидающая подключение клиента. Когда клиент подключился, поток переходит во вложенный цикл «`while`», в котором выполняет три шага: ожидание запроса от клиента (копируется в память `request`), обработка запроса и отправка ответа (из памяти `response`).

6.3.5.9. Скомпилируйте проект и запустите программу в терминале:

```
sudo ./webserver/dist/Debug/GNU-Linux/webserver
main was started
server was started
server was started IP: (null), PORT: 80
```

Запуск через «`sudo`» необходим для доступа к порту «80»

6.3.5.10. Откройте браузер и зайдите на страницу http://IP. Убедитесь, что в терминале появился запрос, который отправил браузер:

```
server info: client was connected
server info: client request:
GET / HTTP/1.1
Host: 192.168.1.36
...
```

6.3.5.11. Добавьте проект в git с комментарием:

```
TCP/IP workflow was added in server.c
```

6.3.6. Добавление протокола HTTP

6.3.6.1. Создадим файлы http.c/.h для набора примитивных функций работы с протоколом HTTP:

http.h:

```
#ifndef HTTP_H
#define HTTP_H

enum HTTP_REQUEST_TYPE {
    HTTP_REQ_INDEX_HTML,
    HTTP_REQ_READDATA,
    HTTP_REQ_WRITEDATA,
    HTTP_REQ_OTHER
};

enum HTTP_REQUEST_TYPE  http_parse_request          (char *request);
void                    http_200OK                  (char *response, char *data);
int                     http_find_data              (char *request, char **indata);

#endif /* HTTP_H */
```

http.c:

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

```
#include "http.h"

// parse header of input HTTP-packet from client:
enum HTTP_REQUEST_TYPE http_parse_request (char *request) {

    const char http_index[]       = "GET / HTTP/1.1";
    const char http_readdata[]    = "GET /readdata HTTP/1.1";
    const char http_writedata[]   = "POST /writedata HTTP/1.1";

    if (strncmp(request, http_index, strlen(http_index)) == 0) return
HTTP_REQ_INDEX_HTML;
    if (strncmp(request, http_readdata, strlen(http_readdata)) == 0) return
HTTP_REQ_READDATA;
    if (strncmp(request, http_writedata, strlen(http_writedata)) == 0) return
HTTP_REQ_WRITEDATA;

    return HTTP_REQ_OTHER;

}

// find data in input HTTP-packet from client:
int http_find_data (char *request, char **indata) {

    // find start of input data:
    for (int i = 0; i < (strlen(request)-4); i++) {
        if (strncmp(request+i, "\r\n\r\n", 4) == 0) {
            *indata = request+i+4;
            break;
        }
    }

    // check data:
    int len = strlen(*indata) - 4;
    if (strncmp(*indata+len, "\r\n\r\n", 4) == 0) {
        return len;
    }

    return -1;

}

// create output HTTP-packet to client:
void http_200OK (char *response, char *data) {

    strcpy(response, "HTTP/1.1 200 OK\r\n");
    strcat(response, "Status: 200 OK\r\n");
    strcat(response, "Content-Length: ");

    if (strlen(data) == 0) {
        strcat(response, "0\r\n\r\n");
    } else {
        sprintf(response+strlen(response), "%d\r\n\r\n%s", strlen(data), data);
    }


}
```

Код http.c содержит 3 функции. Функция «http_parse_request» определяет один из трех типов входного запроса: клиент запрашивает главную страницу, клиент хочет считать данные или клиент хочет записать данные. Функция «http_find_data» ищет и возвращает указатель и размер данных в теле запроса от клиента. Функция «http_200OK» формирует ответный заголовок и добавляет к нему данные.

6.3.6.2. Добавим созданные функции в server.c для обработки запросов и формирования ответов от клиентов. Для этого подключим заголовочные файлы:

```
...
#include <fcntl.h>
#include "server.h"
```

```
#include "settings.h"
#include "http.h"…
```

Добавим создание служебных переменных после подключения клиента:

```
...
// loop for data exchange (echo mode):
        char *request   = malloc(WEBBUF_SIZE);
        char *response  = malloc(WEBBUF_SIZE);
        char *html      = malloc(WEBBUF_SIZE);
        char *data;
        int f, datalen;
...
```

Вместо комментария

```
/* ТУТ БУДЕТ ОБРАБОТКА ЗАПРОСА */
```

добавим логику работы с запросами:

```
...
switch (http_parse_request(request)) {

        // 3.2.2.1. send index.html to web:
        case HTTP_REQ_INDEX_HTML:

            // read index.html:
            f = open(INDEXHTML_PATH, O_RDONLY);
            memset(html, 0, WEBBUF_SIZE);
            read(f, html, WEBBUF_SIZE);
            close(f);

            // create http response:
            http_200OK(response, html);

        break;

        // 3.2.2.2. send data from shared buffer to web:
        case HTTP_REQ_READDATA:

            pthread_mutex_lock(sharedbuf->mutex);
            http_200OK(response, shbuf->buf);
            memset(shbuf->buf, 0, shbuf->bufsize);
            pthread_mutex_unlock(sharedbuf->mutex);

        break;

        // 3.2.2.3. get data from web and send to stdout:
        case HTTP_REQ_WRITEDATA:

            // get data from web and send to stdout:
            datalen = http_find_data (request,  &data);
            if (datalen != -1) {
                data[datalen] = '\n';
                data[datalen] = '\0';
                write(STDOUT_FILENO, data, datalen+2);
            }

            // response to web:
            http_200OK(response, "");

        break;

        // 3.2.2.4. If receive unknown HTTP-packet:
        case HTTP_REQ_OTHER:
            http_200OK(response, "");
        break;

    }
...
```

А в settings.h добавим директиву пути к html странице:

```
// server settings:
…
#define INDEXHTML_PATH "index.html"
```

6.3.6.3. Создадим страницу index.html и добавим в нее следующий код:

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>webserver</title>
        <style>
            label { display: block; min-width: 50px; text-align: center; }
            div { display: flex; }
            textarea { width: 90%; min-height: 200px; }
        </style>
    </head>
    <body>
        <h1>Веб-сервер лабораторного стенда</h1>
        <br>

        <h2>Температура Raspberry Pi</h2>
        <div id="temp">? °C</div>

        <h2>Светодиоды</h2>
        <div>
            <label id="LED0">LED0</label>
            <label id="LED1">LED1</label>
            <label id="LED2">LED2</label>
            <label id="LED3">LED3</label>
        </div>

        <h2>Кнопки</h2>
        <input type="button" value="0" onclick="send_data('BUTTON0: clicked');">
        <input type="button" value="1" onclick="send_data('BUTTON1: clicked');">
        <input type="button" value="2" onclick="send_data('BUTTON2: clicked');">

        <h2>Сообщения</h2>
        <textarea id="log"></textarea>
    </body>
</html>

<script>

    function data_view(key, value) {

        switch (key) {

            case "LED0":
            case "LED1":
            case "LED2":
            case "LED3":
                document.getElementById(key).style.backgroundColor = value;
            break;

            case "temp":
                document.getElementById("temp").innerHTML = value + " °C";
            break;

            case "log":
                var d = new Date();
                var time = String(d.getHours()).padStart(2, '0') + ":";
                time += String(d.getMinutes()).padStart(2, '0') + ":";
                time += String(d.getSeconds()).padStart(2, '0');
                document.getElementById("log").value = time + ": " + value + "\n" +
document.getElementById("log").value;
            break;

        }
```

```
    }

    function data_request() {

        const xhr = new XMLHttpRequest();
        xhr.open('GET', "readdata");

        xhr.onreadystatechange = function() {

            if (xhr.readyState !== 4 || xhr.status !== 200) return;

            var response    = xhr.responseText;
            var msgs        = response.split("\n");
            for (var i = 0; i < msgs.length-1; i++) {
                var msg = msgs[i].split(": ");
                data_view(msg[0], msg[1]);
            }

            window.setTimeout(data_request, 500);

        };

        xhr.send();

    }

    data_request();

    function send_data (data) {
        const xhr = new XMLHttpRequest();
        xhr.open("POST", "writedata", true);
        xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
        xhr.send(data+ "\r\n\r\n");
    }

</script>
```

6.3.6.4. Перейдите в терминал и выполните следующие команды для запуска веб-сервера:

```
cd webserver/
cp dist/Debug/GNU-Linux/webserver webserver
mkfifo data2web
exec 4<>data2web
sudo ./webserver < data2web &
```

6.3.6.5. Откройте браузер и зайдите на страницу http://IP. Убедитесь, что веб-страница отобразилась. Нажмите на кнопки и убедитесь, что в терминале появляются сообщения. Введите в терминале следующие команды:

```
echo “log: Hello!” > data2web
echo “LED0: green” > data2web
echo “LED1: gray” > data2web
```

Убедитесь, что данные отображаются на веб-странице.

6.3.6.6. Выберете в *NetBeans IDE* конфигурацию *Release* и скомпилируйте проект.

6.3.6.7. Создайте новый скрипт «webserver_start.sh» для запуска веб-сервера:

```
#!/bin/sh

# change path to webserver:
path=`dirname "$0"`
cd $path

# Create FIFO-file for data transer from stdin to web:
if [ -e data2web ] ; then
        rm data2web
```

```
fi
mkfifo data2web
exec 4<>data2web

# run webserver:
sudo ./webserver < data2web &
```

6.3.6.8. Создайте новый скрипт «webserver_stop.sh» для остановки веб-сервера:

```
#!/bin/bash

ps -aux | grep "netbeans/webserver" | awk '{print $2}' | xargs sudo kill
```

6.3.6.9. Присвойте права на исполнение созданным скриптам:

```
chmod +x webserver/webserver_start.sh
chmod +x webserver/webserver_stop.sh
```

6.3.6.10. Скопируйте исполняемый файл веб-сервера в рабочую директорию:

```
cp webserver/dist/Release/GNU-Linux/webserver webserver/webserver
```

6.3.6.11. Добавьте результаты в git с комментарием:

```
webserver was created
```

6.3.7. Тест веб-сервера

6.3.7.1. Запустите веб-сервер:

```
webserver/webserver_start.sh
```

6.3.7.2. Создайте скрипт «temperature.sh», получающий температуру процессора и отправляющий ее в stdout один раз в секунду:

```
nano temperature.sh

#!/bin/bash

while [ true ] ; do
        T=`cat /sys/class/thermal/thermal_zone0/temp`
        temp=`echo "scale=2;$T/1000" | bc`
        echo "temp: $temp"
        sleep 1
done
```

6.3.7.3. Назначьте скрипту права на исполнение, запустите его в терминале и убедитесь, что на экране появляются значения температуры:

```
chmod +x temperature.sh
./temperature.sh
```

6.3.7.4. Остановите выполнение скрипта и перенаправьте его потока в веб-сервер через именованный канал:

```
./temperature.sh > webserver/data2web
```

Убедитесь, что на веб-странице значение температуры обновляется.

6.3.7.5. Добавьте созданный скрипт в репозиторий git:

```
git add temperature.sh
git commit -m "script for temperature of MPU was created"
```

7. Контрольные вопросы

1. Что такое поток?
2. Чем отличаются поток и процесс?
3. Для чего необходима синхронизация между потоками?
4. Для чего применяется механизм сокетов?
5. Для чего нужен системный вызов «*accept*»?
6. Что такое удаленная разработка?

8. Список литературы

[1] Б. А. Н. Н. Синицын С.В., Операционные системы: учебник для студ. учреждений высш. проф. образования, М.: Издательский центр "Академия", 2013, p. 304.

[2] Б. Х. Таненбаум Э., Современные операционные системы, СПб.: Питер, 2015, p. 1120.

[3] «Apache NetBeans Community,» [В Интернете]. Available: https://netbeans.apache.org/community/index.html.

[4] «Java | Oracle,» [В Интернете]. Available: https://www.java.com/ru/.

[5] «NetBeans Tutorials,» [В Интернете]. Available: http://netbeans.apache.org/kb/.