Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

В.Г. Резник

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Методические указания по лабораторным работам

Томск
2024

УДК 004.2
ББК 32.97
Р-344

Рецензенты:
Горитов А.Н., профессор кафедры автоматизированных систем управления ТУСУР, доктор техн. наук

Резник, Виталий Григорьевич
Р-344 Архитектура вычислительных комплексов: Методические указания по лабораторным работам для студентов уровня магистратуры технических направлений подготовки / В.Г. Резник. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2024. – 131 с.

Методические указания содержат требования к лабораторным работам студентов по дисциплине «Архитектура вычислительных комплексов» уровня магистратуры технических направлений подготовки.

Одобрено на заседании каф. АСУ протокол № 11 от 23 ноября 2023 года

УДК 004.2
ББК 32.97

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Методические рекомендации содержат учебный материал для восьми лабораторных работ по дисциплине «Архитектура вычислительных комплексов» (АВК) уровня основной образовательной программы магистратуры технических направлений подготовки.

Основной целью дисциплины является обучение студентов базовым понятиям и принципам построения архитектур вычислительных комплексов, содержащих множество процессоров сложной структуры.

В период обучения магистранты должны совершенствовать знания об архитектурном строении современных вычислительных систем, которые они получили ранее на уровне бакалавриата при изучении дисциплины «Операционные системы», должны научиться определять основные тенденции развития предметной области данного направления знаний, а также овладеть новейшими технологическими достижениями в этой области.

Лабораторные и самостоятельные работы по дисциплине ориентированы на закрепление теоретического материала и формирование навыков самостоятельной работы с конкретной системой вычислительного кластера кафедры АСУ.

В результате изучения дисциплины студент должен:

***Знать**:*

- архитектурные методы построения вычислительных комплексов, алгоритмы обработки компьютерных данных и способы разработки научных моделей, предназначенных для создания информационных систем.

***Уметь**:*

- разрабатывать системное программное обеспечение операционных систем вычислительных комплексов, предназначенных для обеспечения требуемого функционала научных информационных систем.

***Владеть**:*

- методиками разработки сосредоточенных и распределённых систем, предназначенных для обработки и хранения данных научных исследований.

В процессе обучения дисциплине проводятся следующие виды учебной работы: лекции, лабораторные работы, самостоятельная работа.

Лабораторные работы проводятся в учебных классах кафедры АСУ на базе «Учебного программного комплекса (УПК АСУ)».

Методическое обеспечение данного курса опирается на литературные источники [1-5].

В процессе выполнения лабораторных работ, студент должен заполнять единый отчет, который является обязательной частью процесса обучения.

Содержание и качество материала отчёта влияет на общую оценку, выставляемую магистранту во время промежуточной аттестации по данной дисциплине.

1 СОСТОЯНИЕ И ТЕНДЕНЦИИ РАЗВИТИЯ АВК

Первая тема является вводной частью изучаемой дисциплины. В ней рассматриваются общие вопросы, включающие общий обзор предметной области и тематики изучаемой дисциплины, а также обзор учебного материала, приведенного в источниках [1, 2, 5]. Теоретический материал закрепляется выполнением трёх лабораторных работ проводимых как в среде ОС УПК АСУ, так и в операционной среде кластера ЭВМ кафедры АСУ.

1.1 Лабораторная работа №1. Тестирование рабочей области студента

По традиции первая лабораторная работа содержит описание среды ОС, в которой они проводятся. Такая среда содержит:

1) компьютеры учебных классов кафедры АСУ, полностью обеспечивающие индивидуальное обучение магистранта;
2) общее программное обеспечение каждого компьютера с установленной ОС MS Windows, подключённой к общей локальной сети кафедры АСУ;
3) специальное программное обеспечение: ОС УПК АСУ, содержащее учебно-методический материал по изучаемой дисциплине.

Для выполнения данной и последующих лабораторных работ, магистрант должен изучить:

1) состав тем вопросов и лабораторных работ изучаемой дисциплины, описанных в руководстве [1];
2) местоположение и файловую структуру ***ОС УПК АСУ*** в пределах файловой системы ***ОС MS Windows***, используя методическое руководство [5];
3) местоположение и состав учебно-методического материала данной и последующих лабораторных работ, используя файл ***avk-selfworks.pdf***, размещённый на рабочем столе пользователя ***upk***;
4) подготовить личный накопитель flashUSB для выполнения лабораторных работ, хранения личных данных и отчёта о проделанных работах.

1.1.1 Структура учебной части дистрибутива ОС УПК АСУ

Общая структура ОС УПК АСУ достаточно подробно описана в методическом руководстве [5], которым первоначально и следует воспользоваться. Далее, необходимо изучить:

1) правила общей загрузки ОС и вход в систему пользователем ***asu***;
2) местоположение рабочей области студента на его личном flashUSB;
3) подключение рабочей области пользователя ***upk*** к среде ОС;
4) переход в сессию пользователя ***upk***;
5) правила завершения работы студента, включающие: возврат в сессию пользователя ***asu***, отключение рабочей области, сохранение рабочей области на личном flashUSB;
6) отключение flashUSB от среды ОС и завершение работы (выключение системы).

Рабочий стол сессии пользователя ***upk***, для изучаемой дисциплины, имеет стилизованную фоновую заставку, показанную на рисунке 1.1.

Рисунок 1.1 — Рабочий стол пользователя upk для изучаемой дисциплины «Архитектура вычислительных комплексов»

На рабочем столе имеются:

1) значки доступных файловых систем (возможно отключённых);
2) avk-selfworks.pdf — перечень тем и вопросов по обучаемой дисциплине;
3) Отчет.doc — шаблон отчёта студента;
4) upk_asu.pdf — методическое руководство [5];
5) значок перехода в директорию с учебным материалом;
6) значок запуска системы разработки языка С на базе IDE Eclipse.

Основной учебный материал по данной дисциплине расположен в директории *~/Документы*. Здесь располагаются:

1) *публикации* и другой учебный материал;
2) *темы* — каталог для размещения методических пособий, содержащие как теоретический материал, так и описания лабораторных работ.

Примечание — В связи с обновлением версий ОС УПК АСУ и другого (*сопутствующего*) программного обеспечения, студенту следует убедиться в работоспособности всех компонент системы и (*при необходимости*), устранить имеющиеся недостатки, руководствуясь указаниями преподавателя.

1.1.2 Настройка сети ОС УПК АСУ

Современные компьютерные комплексы широко используют сети ЭВМ, в среде которых они установлены. В связи с этим, необходимо настроить сетевое ПО ОС УПК АСУ, используя знания, полученные при изучении курса «Современные операционные системы».

Дополнительно, магистр, проходящий обучение по дисциплине «*Архитектура вычислительных комплексов*» должен:

- иметь личные данные для авторизации на компьютерах кафедры АСУ;
- уметь включать компьютеры, расположенные в учебных классах кафедры АСУ, а также проходить процедуры авторизации на ОС УПК АСУ;
- иметь личные данные для авторизации на сервере кластера кафедры АСУ.

Примечание — Первоначальная установка дистрибутива ОС УПК АСУ не имеет привязки к локальной сети кафедры АСУ, поэтому необходимо научится делать это самостоятельно!

Современные дистрибутивы ОС имеют достаточно развитые средства настройки и работы с сетями ЭВМ. Используемый для обучения дистрибутив ОС УПК АСУ создан на базе дистрибутива ***Arch Linux***, который имеет сетевое ПО, основанное на сетевом пакете ***netctl***, совместимым с управляющим менеджером ОС — ***systemd***.

Замечание.

Старые версии ОС Linux, для обозначения сетевых интерфейсов устройств, использовали *статические* имена:

1) ***ethX*** — проводное устройство Ethernet;
2) ***wlanX*** — беспроводное устройство Wi-Fi,

где ***X*** — номер устройства, начинающийся со значения ***0***.

Новые версии ОС Linux, для обозначения сетевых интерфейсов устройств, используют *динамические* имена, привязанные к узлам шины *pci* (см. рисунок 1.2):

1) ***enpYYYY*** — проводное устройство Ethernet;
2) ***wlpYYYY*** — беспроводное устройство Wi-Fi,

где ***YYYY*** — обозначение узла шины ***pci***.

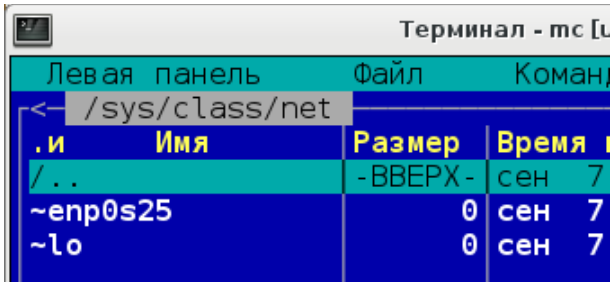

Рисунок 1.2 — Интерфейсы компьютера с одним адаптером сети Ethernet

Для проверки готовности компьютера к работе в сети следует воспользоваться утилитами ***ip*** или ***ifconfig***. На рисунке 1.3 представлен пример вывода утилиты ***ip***, которая показывает, что сетевой интерфейс ***enp0s25*** готов к работе.

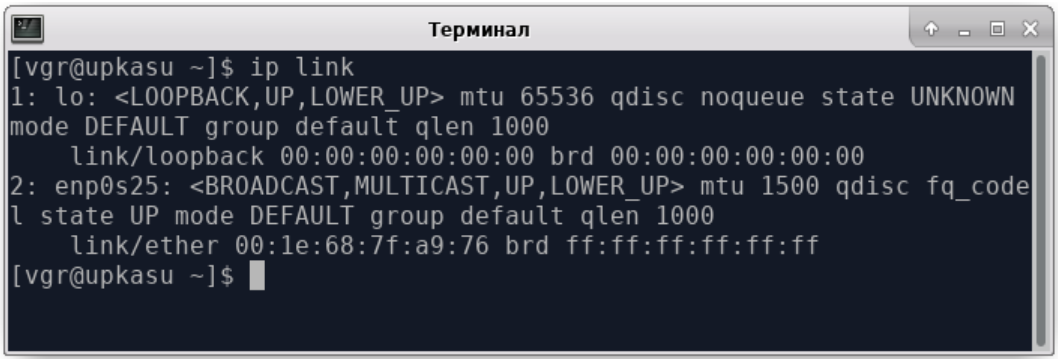

Рисунок 1.3 — Проверка наличия и рабочего состояния сетевых интерфейсов в среде ОС УПК АСУ

1.1.3 Контроль выполнения лабораторной работы №1

В результате выполнения лабораторной работы №1 магистрант должен:

1) проверить запуск работу и полный комплект ПО по изучаемой дисциплине, а также соответствие установленного ПО требованиям методического пособия [5];
2) изучить и освоить работу утилит ***ip*** и ***ifconfig***, а также правила настройки сети ОС УПК АСУ с помощью средств пакета ***netctl***;
3) описать результаты проделанной работы в личном отчёте;
4) отметить завершение работы у преподавателя;
5) сохранить резервную копию отчёта.

1.2 Лабораторная работа №2. Работа со средой кластера кафедры АСУ

Цель лабораторной работы №2 — освоить консольный и графический интерфейсы доступа к кластеру ЭВМ кафедры АСУ.

Учитывая поставленную цель, выполнение данной лабораторной работы выполняется под общим рабочим названием «***Работа со средой кластера кафедры АСУ***».

1.2.1 Работа со средой кластера ЭВМ кафедры АСУ

Цели и назначение кластера ЭВМ кафедры АСУ достаточно хорошо описаны в статье И.В. Бойченко, представленной в рабочей области студента файлом ***HPC-TUSUR-START-2008.pdf***. Студенту следует познакомиться с этой статьёй, чтобы иметь общее представление о конкретном варианте архитектуры вычислительного комплекса.

Все сетевое ПО кафедры АСУ настроено таким образом, что компьютеры получают ***IP-адрес*** автоматически по протоколу ***DHCP***.

При старте ОС УПК АСУ должен использоваться протокол DHCP, обеспечивающий получение адреса, маски и основного маршрутизатора компьютера. Если используются другие альтернативные настройки, то результирующим критерием может быть «видимость» сайта: ***cluster.asu.tusur.ru***.

Примечание — Если при запуске ОС УПК АСУ сеть не установилась, следует обратиться к преподавателю, обеспечивающему проведение лабораторных работ.

1.2.2 Настройка сетевого обеспечения ОС

Проверка работоспособности настроек сети проводится разными способами, один из вариантов которых выполняется в два этапа.

Этап 1. Запустить из меню рабочего стола «Эмулятор терминала» и выполнить команду ***ifconfig***. Появится информация, например, как показано на рисунке 1.4.

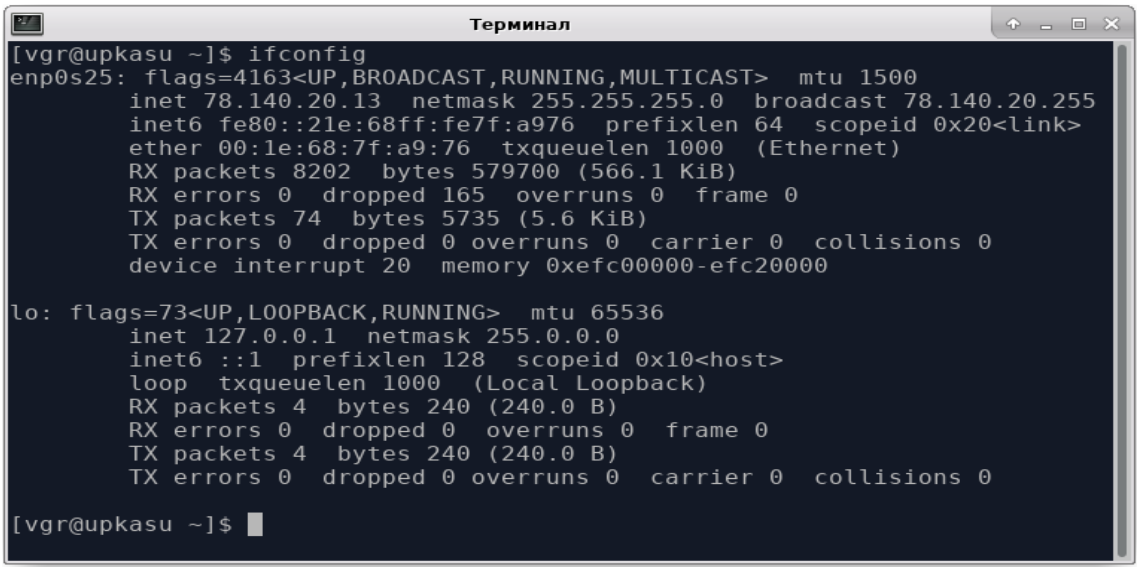

Рисунок 1.4 — Сообщения команды ifconfig

Хорошо видно, что сообщение, выделенное как ***enp0s25***, показывает IP адрес компьютера, его маску и другие параметры.

Этап 2. Убедившись подключении рабочей станции к сети, следует проверить доступность кластера ЭВМ кафедры АСУ. Для этого необходимо выполнить команду **ping** по адресу кластера, что показано выражением (1.1).

sudo ping cluster.asu.tusur.ru **(1.1)**

На рисунке 1.5 представлены результаты выполнения команды (1.1).

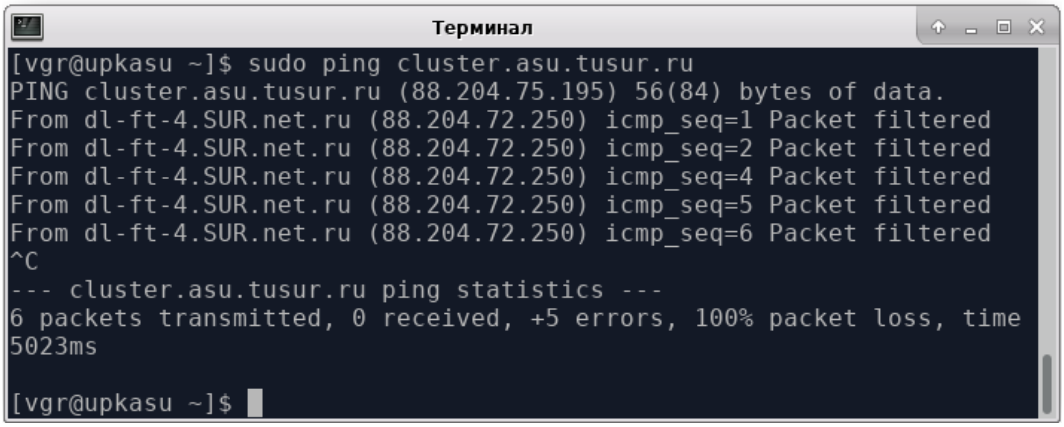

Рисунок 1.5 — Проверка наличия кластера командой ping

Убедившись, что кластер обнаружен, необходимо остановить работу ***ping*** посредством комбинации клавиш ***<Ctrl>+C***.

Примечание — Если кластер ЭВМ кафедры АСУ не обнаруживается, то следует обратиться к преподавателю, обеспечивающему проведение лабораторных работ.

1.2.3 Консольный доступ к кластеру

Консольный доступ к удаленным ЭВМ — устаревшая технология взаимодействия компьютеров известная как *система телекоммуникаций*. На базе этой технологии разрабатывалась модель терминала ОС.

Учебная цель данного пункта — установление защищённого соединения с кластером ЭВМ и настройка переменных среды окружения удалённой ОС.

Для обеспечения такого доступа к удалённому компьютеру существует множество программ как для ОС MS Windows, так и для ОС Linux. Наиболее известной из них является программа ***PuTTY***, которая имеется для обоих ОС.

В данной работе используется способ, осуществляемый с помощью «Эмулятора терминала» ОС Linux посредством вызова программы ***ssh*** (*security shell*).

Чтобы выполнить такое подключение, необходимо набирать команду согласно выражению (1.2).

ssh <имя пользователя>@asu.local@cluster.asu.tusur.ru **(1.2)**

где ***cluster.asu.tusur.ru*** — адрес главного компьютера кластера кафедры АСУ; ***<имя пользователя>*** — имя пользователя для регистрации на сервере кластера.

Пример такого действия показан на следующем рисунке 1.6.

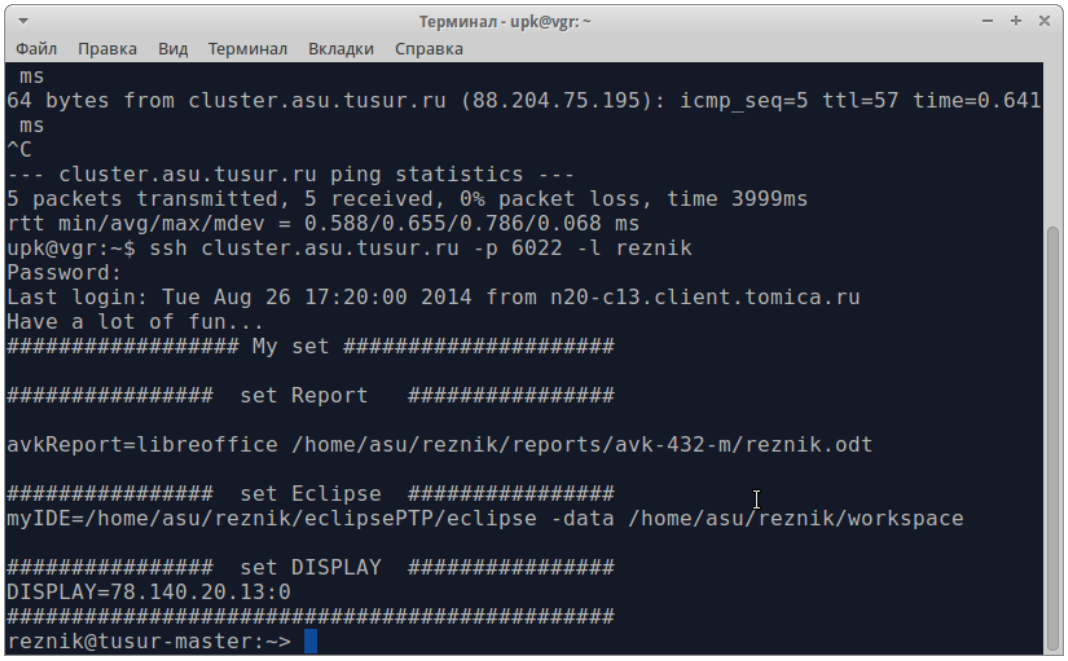

Рисунок 1.6 — Пример терминального соединения с кластером

Примечание — Удалённый сервер может запросить согласие на генерацию ключей. Нужно ответить: **yes**.

Затем, сервер запросит пароль, который студент должен правильно ввести и нажать клавишу «***Enter***».

Примечание — ***Имя*** и ***пароль*** для доступа на кластер кафедры АСУ студент получает у преподавателя, ведущего лабораторные занятия.

Далее командой ***mc*** следует запустить утилиту ***Midnight Commander*** и изучить домашнюю директорию удалённого пользователя, как показано на рисунке 1.7.

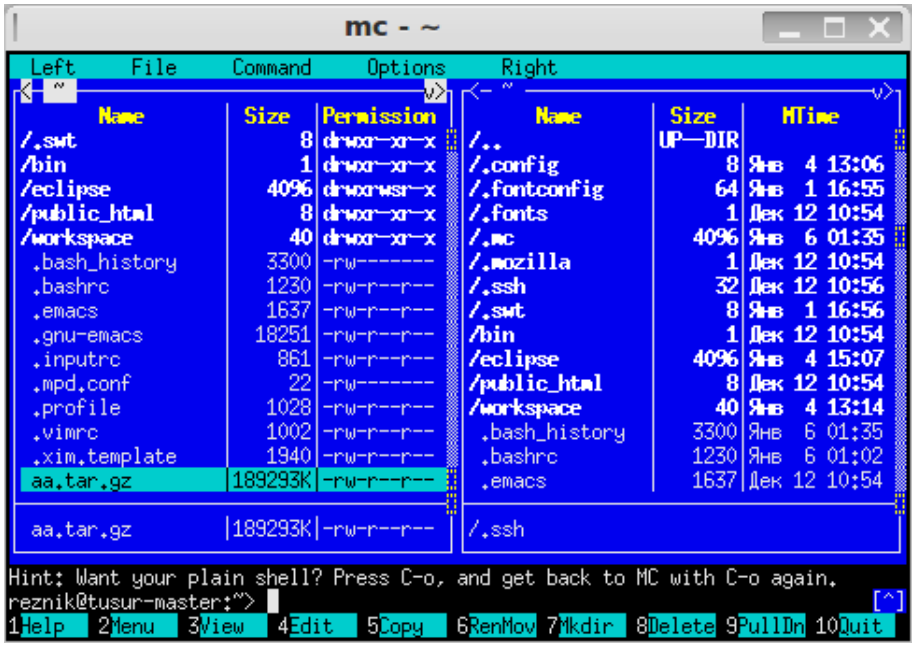

Рисунок 1.7 — Домашняя директория пользователя на кластере кафедры АСУ

Примечание — Если студент неправильно введёт пароль, то удалённый сервер предложит повторный ввод. После нескольких неудачных попыток, сервер разрывает соединение. Если пароль введён правильно, то сервер выдаёт подсказку, характерную для консоли (терминала) ОС.

Получив доступ к ЭВМ кластера, следует изучить среду пользователя, под именем которого был осуществлён вход.

Наиболее важными для изучения являются файлы сценариев ***.profile*** и ***.bashrc***. Эти сценарии запускаются, когда пользователь «успешно сделал login» в системе. Эти сценарии создают переменные окружения (переменные среды) ОС.

Основной список переменных среды студент изучал в третьей теме дисциплины «Операционные системы». Наиболее важными являются переменные среды ОС, которые представлены в таблице 1.1.

Таблица 1.1 — Наиболее важные переменные среды ОС

DISPLAY	Указывает адрес и экран локальной или удалённой ЭВМ, на которую буде осуществляться графический вывод приложений. Подробно рассмотрена в следующем подразделе.
HOME	Директория пользователя.
PATH	Список путей, в которых ищутся запускаемые программы.
USER	Имя пользователя, под которым осуществлён вход в систему.

Чтобы проверить значения этих переменных, можно воспользоваться командами, которые представлены выражениями (1.3) — (1.5).

echo $USER<Enter> **(1.3)**
echo $<Enter> **(1.4)**
echo $PATH<Enter> **(1.5)**

В большинстве случаев, чтобы установить нужные значения переменных среды, достаточно провести редактирование файла ***.bashrc***. Закончив его редактирование, необходимо выполнить команду выражения (1.6).

. ./.bashrc **(1.6)**

Затем, следует повторно проверить значения этих переменных и завершить удалённый доступ к кластеру.

Примечание — Закончив изучение файловой системы удалённого компьютера, студент должен корректно выйти из удалённого соединения (*просто закрывать окно терминала не следует!*). Для этого, обычно достаточно последовательности команд ***exit***.

Например, в рассмотренном случае достаточно двух команд ***exit***:

1) ***первая*** — закроет приложение ***Midnigth Commander***;
2) ***вторая*** — разорвёт соединение с удаленным сервером.

1.2.4 Графический доступ к кластеру кафедры АСУ

Учебная цель этого пункта:

1) *изучить* архитектуру X Window System;
2) *научится* запускать X-сервер с нужными параметрами доступа;

3) *проверить* возможность подключения к X-серверу программ, запущенных на сервере кластера.

Примечание - В отличие от MS Windows, имеющей встроенное в ядро ОС графическое ПО, без которого она не может работать, Linux для этой цели имеет набор приложений, взаимодействующих на основе модели ***«клиент/сервер»***.

Базовая часть графического ПО Linux называется ***X Window System*** *(*или просто - ***X Window****)*.

Начиная с 1988 года, этот стандарт поддерживался консорциумом X, созданным с целью унификации графического интерфейса для ОС UNIX. В 1997 году консорциум X был преобразован в *X Open Group (**http://www.x.org**)*.

X Window — достаточно сложная система и подробно описана большим количеством первоисточников. Мы рассмотрим только основные элементы архитектуры этой системы, опираясь на материал Интернет статьи: «Костромин В. Графический интерфейс Linux: ***http://archive.li/QOuGM***».

Согласно этой статье, общая архитектура X Window может быть представлена рисунком 1.8.

Рисунок 1.8 — Общая архитектура X Window System

«Сердцем» этой графической системы является программа X-сервер, которая через драйверы устройств взаимодействует с видеоплатой, клавиатурой, мышью и монитором компьютера. Именно X-сервер устанавливает и переключает графические режимы видеоплаты, рисует элементы изображений, определяет координаты мыши и формирует программные прерывания при нажатии кнопок мыши и клавиатуры. Все остальные программы, включая

менеджер окон, взаимодействуют с X-сервером по особому протоколу, который называется ***X-протокол***, или протокол сетевой связи (*X Network Protocol*).

Для написания программ, поддерживающих ***X-протокол***, имеется базовая библиотека ***X-lib***. На основе этой библиотеки пишутся дополнительные графические библиотеки более высокого уровня, например, ***GTk+***, ***Qt***, ***Motif*** и другие. Обычно, менеджеры окон, рабочие столы и сложные графические приложения пишутся с использованием этих библиотек.

В общем случае, X-сервер обрабатывает ***4 типа сообщений***:

1. **Запрос** – клиент требует нарисовать что-либо в окне или запрашивает у сервера информацию.
2. **Ответ** – сервер отвечает на запрос.
3. **Событие** – сервер сообщает клиенту о событии (например, о нажатии клавиши пользователем).
4. **Ошибка** – сервер сообщает об ошибке.

Замечание — Важной для нас особенностью является возможность X-сервера работать на стеке протоколов TCP/IP с программами, запущенными на удаленных компьютерах. Обычно используется асинхронная связь (протокол UDP), но возможна и синхронная связь по протоколу TCP, которая работает в 30 раз медленнее. Кроме того, на компьютере может быть запущено несколько X-серверов, которые выводят графическую информацию на разные дисплеи с разными номерами: 0, 1, 3 и т.д.

Чтобы прикладные программы знали на каком компьютере и на каком дисплее находится X-сервер, используется переменная окружения ***DISPLAY***, которая содержит информацию в формате выражения (1.7) (см. также таблицу 1.2).

```
<IP-адрес X-сервера>:<номер дисплея>.<номер экрана дисплея>      (1.7)
```

Таблица 1.2 - Примеры задания значений переменной DISPLAY

Значение переменной DISPLAY	**Номер порта соединения**	**Пояснение**
:0	--	X-сервер на локальном компьютере, дисплей №0, экран №0
:1.0	--	X-сервер на локальном компьютере, дисплей №1, экран №0
asu.tusur.ru:4	6004	X-сервер на компютере asu.tusur.ru, дисплей №4, экран №0, номер UDP-порта - 6004
192.168.1.17:2	6002	X-сервер на компьютере 192.168.1.17, дисплей №0, экран №0, номер UDP-порта - 6002

Многие прикладные программы имеют специальный параметр -display, после которого можно указать указать IP-адрес и номер дисплея вывода.

Например, запуск программы xterm можно осуществить в виде выражения (1.8).

```
xterm -display foo:1<Enter>                                      (1.8)
```

В результате, команда выведет консольную программу на дисплей **№1** соответствующего удалённого компьютера **foo**.

Если на компьютере запущено несколько X-серверов, то следует пользоваться «горячими клавишами»:

1) <Ctrl>+<Alt>+<Backspact> - приведет к остановке X-сервера;
2) <Ctrl>+<Alt>+<F#> - переключение между консолями Linux.

Примечание — Запустить X-сервер может только пользователь **root**. Запуск производится с помощью программы **xinit**, общий формат запуска которой имеет вид выражения (1.9), где <**клиент**> - параметры, определяющие клиентскую часть; <**дисплей**> - адрес компьютера и номер дисплея, где загрузится X-сервер; <**сервер**> - серверная часть параметров.

```
xinit <клиент> -- <дисплей> <сервер>                    (1.9)
```

Следует также заметить, что структура запуска программы ***xinit*** — довольно сложна и требует высокой квалификации.

На практике используются два основных варианта запуска X-сервера:

1. **Вариант 1**. X-сервер запускается менеджером дисплея, например, ***gdm*** (*gnome display manadger*), во время загрузки ОС в графическом режиме.
2. **Вариант 2**. X-сервер запускается пользователем из текстового режима посредством скрипта ***startx***.

В любом из используемых вариантов, конечным результатом этих усилий будет запуск программы xinit с нужными параметрами.

Здесь следует учесть, что основными файлами, содержащими параметры запуска X-сервера, являются:

1) /etc/X11/Xsession;
2) /etc/xinit/xinitrc;
3) /etc/xinit/xserverrc.

Примечание — Если эти файлы скопировать в домашнюю директорию пользователя с именами **.Xsession**, **.xinitrc** и **.xserverrc**, а затем отредактировать нужным образом, то X-сервер будет запускаться с параметрами указанными в этих файлах.

Кроме перечисленных свойств, X-сервер имеет набор механизмов различного уровня защиты от несанкционированного доступа. Современные дистрибутивы Linux, такие как ОС Xubuntu, ArchLinux и другие, поставляются с параметрами, запрещающими X-серверу подключать программы удалённых компьютеров.

Примечание — Поскольку подробное обсуждение средств защиты выходит за рамки данного пособия, студенту следует использовать интерактивное справочное руководство **man** с параметром **Xsecurity**.

Во многих дистрибутивах ***Linux*** для запуска X-сервера используется дисплейный менеджер ***LightDM***. Чтобы этот менеджер разрешал X-серверу прослушивание и подключение сети, необходимо:

1) открыть файл **/etc/lightdm/lightdm.conf**;
2) в секцию **[SeatDefaults]** добавить строку согласно выражению (1.10).

```
xserver-allow-tcp=true                                  (1.10)
```

Далее создать ещё одну секцию согласно выражения (1.11).

```
[XDMCPServer]
enabled=true                                            (1.11)
```

В завершение настроек необходимо перезапустить дисплейный менеджер командой выражения (1.12).

```
sudo restart lightdm                                    (1.12)
```

Примечание — В дистрибутиве ОС УПК АСУ дисплейный сервер ***lightdm*** — не используется.

В дистрибутиве ОС УПК АСУ X-сервер запускается сценарием ***startxfce4***, познакомиться с содержимым которого можно в директории ***/bin***. Сам запуск прописан в файле ***/etc/X11/xinit/xserverrc***, как показано на рисунке 1.9.

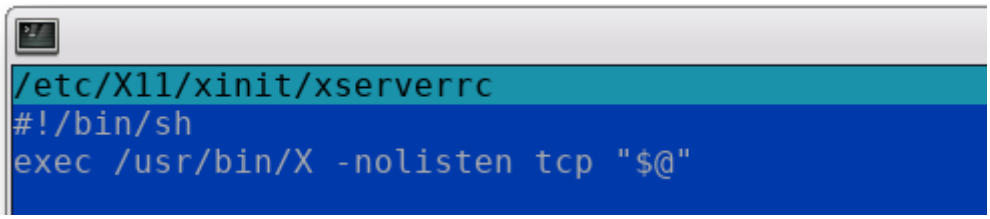

Рисунок 1.9 — Содержимое файла /etc/X11/xinit/xserverrc

Чтобы сервер прослушивал сеть необходимо параметр ***-nolisten*** заменить на параметр ***listen***. Тогда X-сервер запустится, как показано на рисунке 1.10.

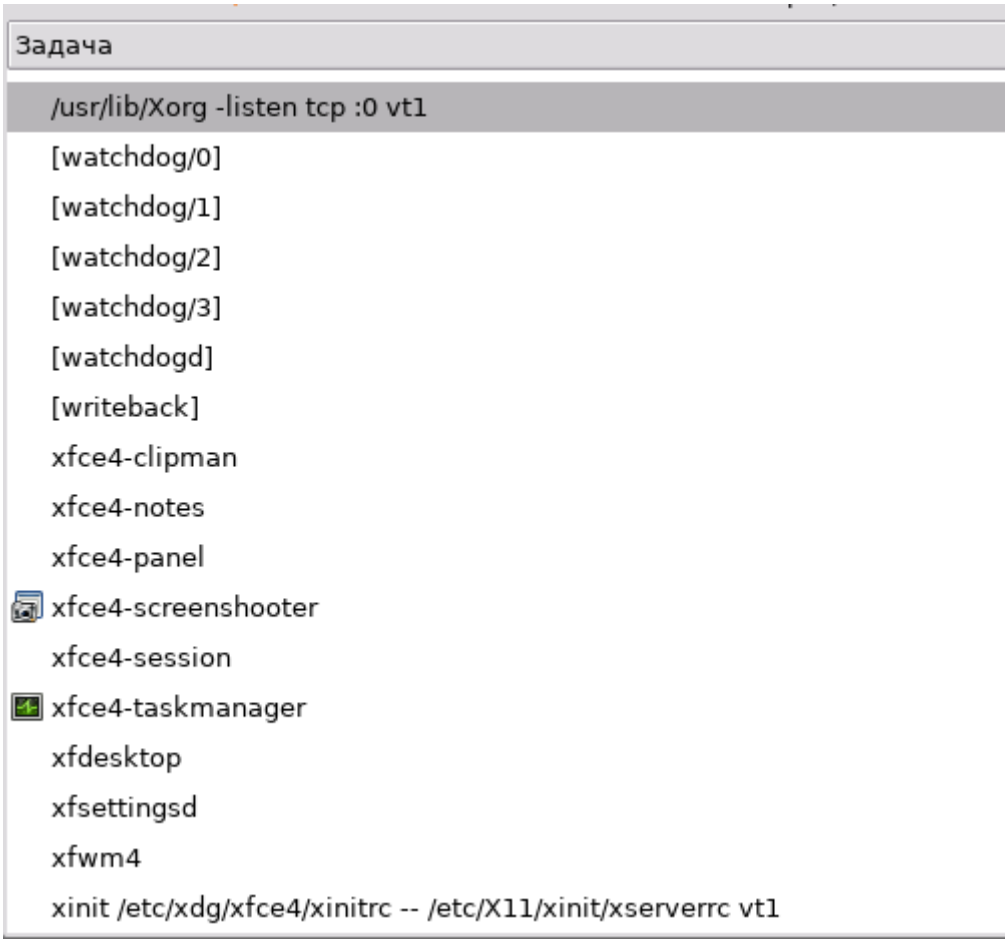

Рисунок 1.10 — Проверка запуска X-сервера из диспетчера задач

Изучив основные элементы архитектуры X Window System, перейдём к пошаговому изложению операций, которые студент должен выполнять на каждой лабораторной работе от

момента запуска дистрибутива ОС УПК АСУ до момента запуска приложения на кластере кафедры АСУ.

Шаг 1. Загрузить ОС УПК АСУ, в графическом режиме, и выполнить все необходимые действия по настройке сети компьютера согласно пунктам 1.1.1 и 1.1.2 данного руководства.

Шаг 2. С помощью команды ***ifconfig*** или ***ip***, определить и запомнить **IP**-адрес компьютера, на котором проводится лабораторная работа.

Шаг 3. В консоли, от имени пользователя **root**, следует выполнить команду (1.13).

```
xhost +cluster.asu.tusur.ru
```
(1.13)

Эта команда разрешит пользователям компьютера ***cluster.asu.tusur.ru*** подключаться к вашему X-серверу.

Примечание — Если в команде ***xhost*** заменить '+' на '-', то доступ с компьютера ***cluster.asu.tusur.ru*** к вашему X-серверу будет запрещен.

Соответственно для проверки, что на компьютере разрешено подключение к X-серверу, необходимо использовать команду (1.14).

```
xhost
```
(1.14)

Эта команда покажет список и уровень защищенности всех разрешённых соединений рабочей станции студента.

Шаг 4. Командой ***ssh*** подключиться к серверу ***cluster.asu.tusur.ru***, используя правила, изложенные в пункте 1.2.2 данного пособия (см. выражение (1.2)).

Шаг 5. Находясь на консоли удалённого сервера, отредактировать файл ***.bashrc***, установив переменную ***DISPLAY***, как показано на рисунке 1.11:

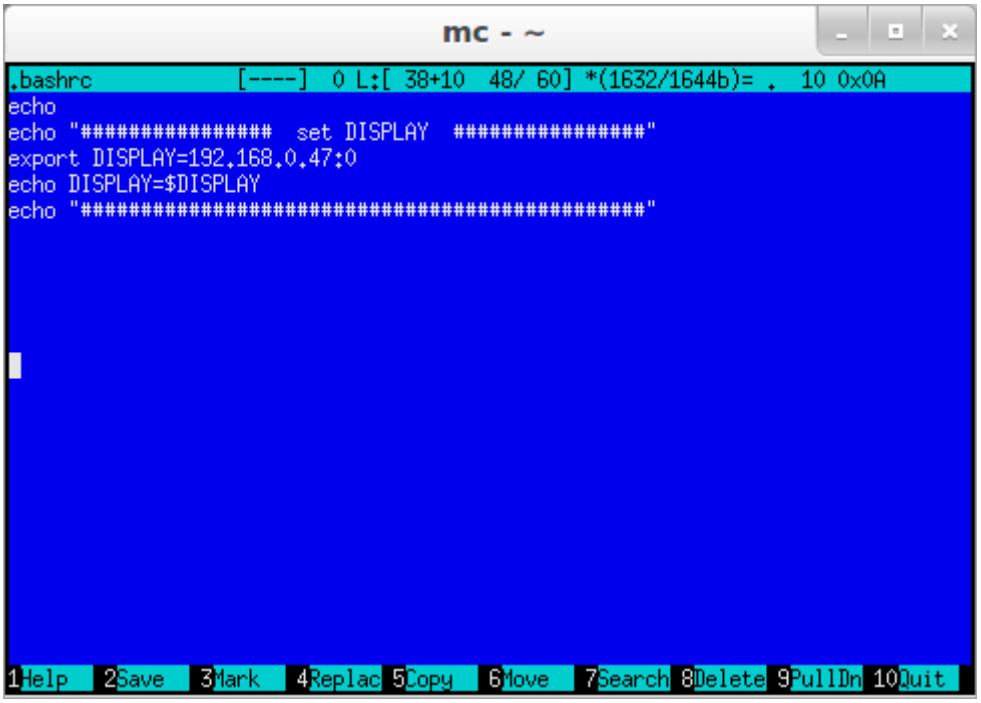

Рисунок 1.11 - Редактирование системной переменной DISPLAY

Примечание — В качестве параметра IP-адреса, студент должен указать ***IP-адрес*** своего компьютера, определённого ранее на ***шаге 1***. Переменную ***DISPLAY*** всегда нужно проверять, поскольку IP-адрес компьютера студента может изменяться.

Далее следует выйти из редактора и перезапустить скрипт ***.bashrc***, как показано на рисунке 3.9, чтобы убедиться в правильности установки переменной ***DISPLAY***.

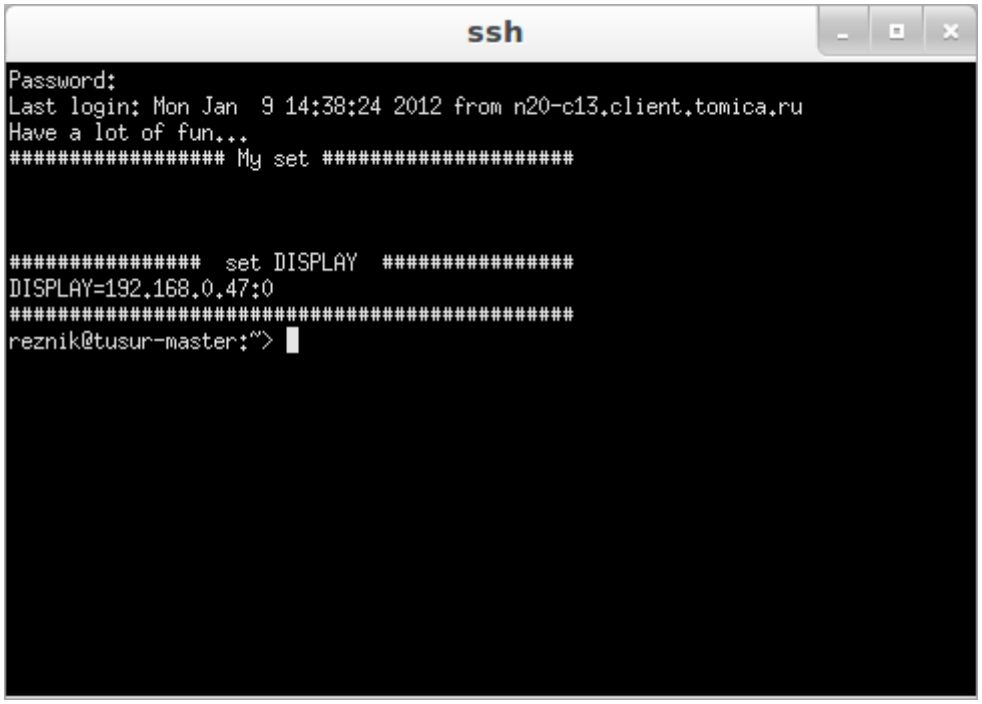

Рисунок 1.12 — Проверка установки переменной DISPLAY

Шаг 6. Нужно дополнительно убедиться в правильности настроек, запустив на удалённом компьютере команду ***xterm***. При правильных настройках, которые были проведены на предыдущих шагах, на рабочем столе должно появиться новое окно консоли, озаглавленное ***xterm***.

Выйти из этой консоли можно командой ***exit***.

Примечание — Все перечисленные ***шесть шагов*** следует выполнять каждый раз перед самым ***началом выполнения работ на кластере кафедры АСУ***.

Примечание - Если на сервере кластера ЭВМ кафедры АСУ не установлен X-сервер, то удалённый запуск графических приложений становится *невозможным!* В таком случае, можно провести имитацию работы с кластером, используя возможности ПО компьютеров учебного класса. Как это сделать, описано в методическом пособии по шестой теме дисциплины «Современные операционные системы». Соответствующее пособие можно взять у преподавателя!

1.3 Лабораторная работа №3. Удалённая разработка приложений

Обычно, разработка приложений осуществляется на локальном компьютере. Для этой цели используются интегрированные среды разработки (***IDE***).

В ряде случаев, такая разработка затруднительна или невозможна по различным причинам. Тогда используются различные варианты удалённой разработки.

Цель лабораторной работы №3 — освоить удалённую разработку приложений на кластере ЭВМ кафедры АСУ.

Учитывая поставленную цель, выполнение данной лабораторной работы выполняется под общим рабочим названием «***Удалённая разработка приложений***».

Обоснованием указанной цели может служить две причины:

1) *невозможность* использования на локальной рабочей станции инструмента ***Eclipse PTP*** — платформы для разработки параллельных вычислений;
2) *возможность* организации удалённой разработки ПО с использованием сетевых возможностей ***X Window System*** доступных в ОС UNIX/Linux.

1.3.1 Удалённая разработка приложений

Основное учебное задание:

1) подключиться к кластеру кафедры АСУ и запустить интегрированную среду разработки (IDE) ***EclipsePTP***;
2) создать типовой проект на языке С, который выводит на консоль строку сообщения: **«Hello, world!!!»**.

Навыки полученные студентом во время выполнения данной лабораторной работы, обеспечат его умениями, необходимыми для реализации программного обеспечения, требующего параллельных вычислений.

Запустив систему ОС УПК АСУ, настроив и подключившись к кластеру кафедры АСУ, как описано в лабораторных работах №1 и №2, студент запускает Midnight Commander и переходит к работе с интегрированной средой разработки IDE ***EclipsePTP***.

Для этого студент заходит в директорию ***/home/asu/reznik/eclipsePTP/*** и запускает на выполнение файл ***eclipse***. На экране монитора появится окно ***«Select a workspase»***, которое предлагает определить директорию рабочей области проектов (см. рисунок 1.13).

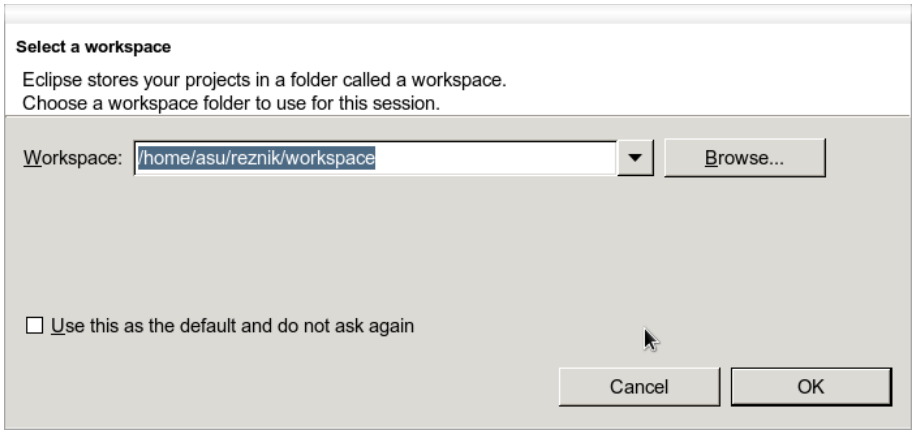

Рисунок 1.13 — Окно выбора директории проектов EclipsePTP

Далее следует отредактировать строку ввода «***Workspace:***», придерживаясь формата ***<Домашняя директория студента>/workspace*** и, воздействуя на кнопку ***OK...*** Загрузить среду разработки EclipsePTP.

Примечание — Если рабочая директория среды разработки выбирается первый раз, то появится окно с предложением выбрать разделы инструментальной среды. Следует выбрать раздел «***Workbench***» и появится ***окно среды разработки***. При последующих выборах этой директории, рабочая среда разработки ***Workbench*** будет показана сразу.

На рисунках 1.14 и 1.15, показаны оба окна запуска среды EclipsePTP.

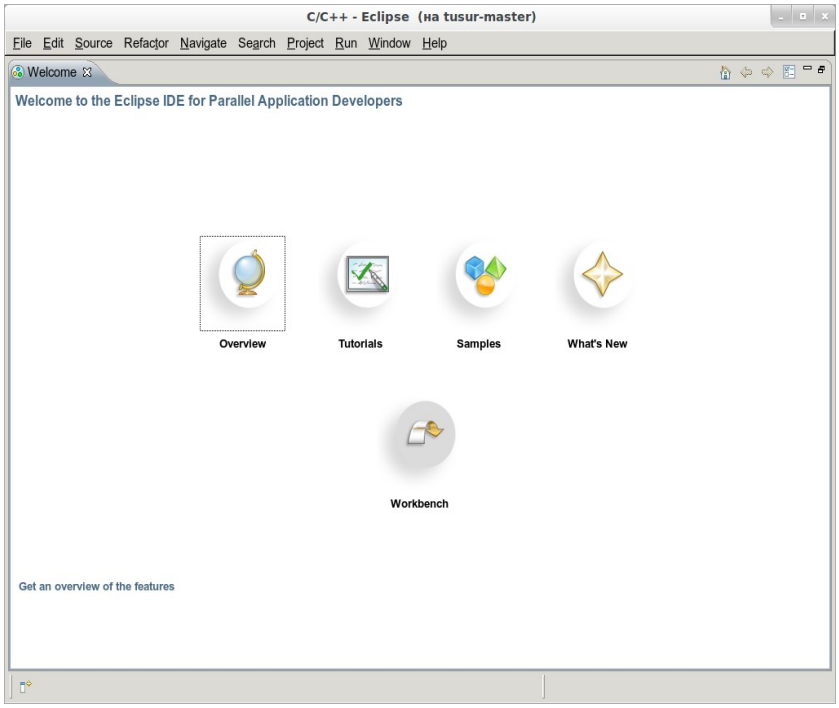

Рисунок 1.14 — Окно EclipsePTP при первом создании рабочей области

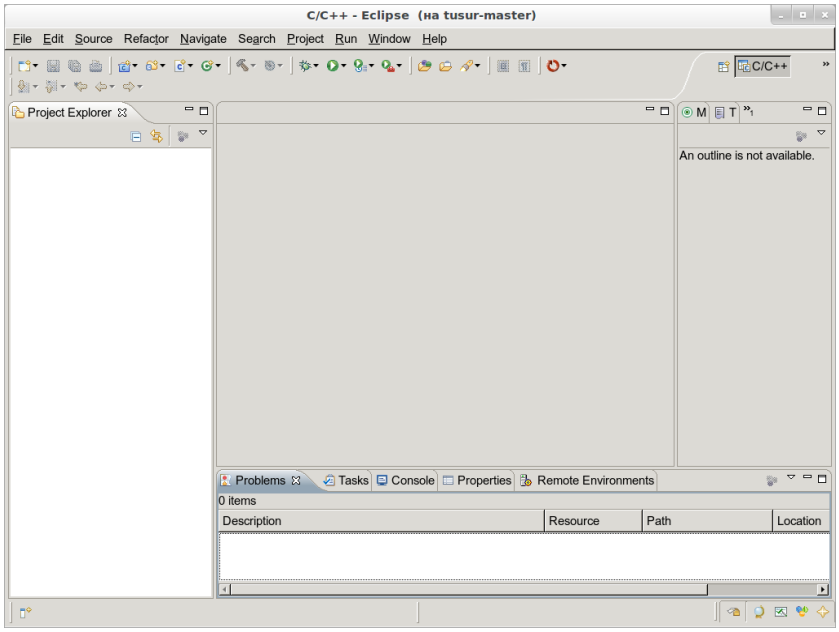

Рисунок 1.15 — Рабочее окно среды разработки EclipsePTP

Примечание — Интегрированная среда разработки IDE EclipsePTP, как и большинство других современных сред разработки, делит создаваемые программные продукты на проекты. Все создаваемые проекты, в пределах рабочей области Workspace, должны иметь уникальные имена.

С целью упорядочения процесса проведения занятий, а также последующей отчётности и контроля, студент на каждой лабораторной работе создаёт один проект. Соответственно, проекты именуются по порядку: ***lab1***, ***lab2***, ***lab3*** и далее.

Задание на разработку ПО.

В данной лабораторной работе, студент должен:

Создать проект с именем ***lab1*** для типовой программы на языке С.

Программа проекта ***lab1*** должна выводить на стандартное устройство вывода (консоль) текст, содержание которого определяется форматом выражения ().

```
Hello, world!!! <Фамилия студента> <Имя студента>          (1.15)
```

Учебная цель данного задания:

1) закрепление знаний работы с интегрированной IDE EclipsePTP;
2) уточнение настроек рабочей среды разработки под локальную ОС;
3) практическое закрепление технологического цикла удаленной разработки ПО.

Чтобы максимально упростить рутинные действия при запуске IDE EclipsePTP, следует дополнительно отредактировать файл ***.bashrc***, как показано на рисунке 1.16. Тогда запуск EclipsePTP можно будет производить с консоли командой: **$myIDE**

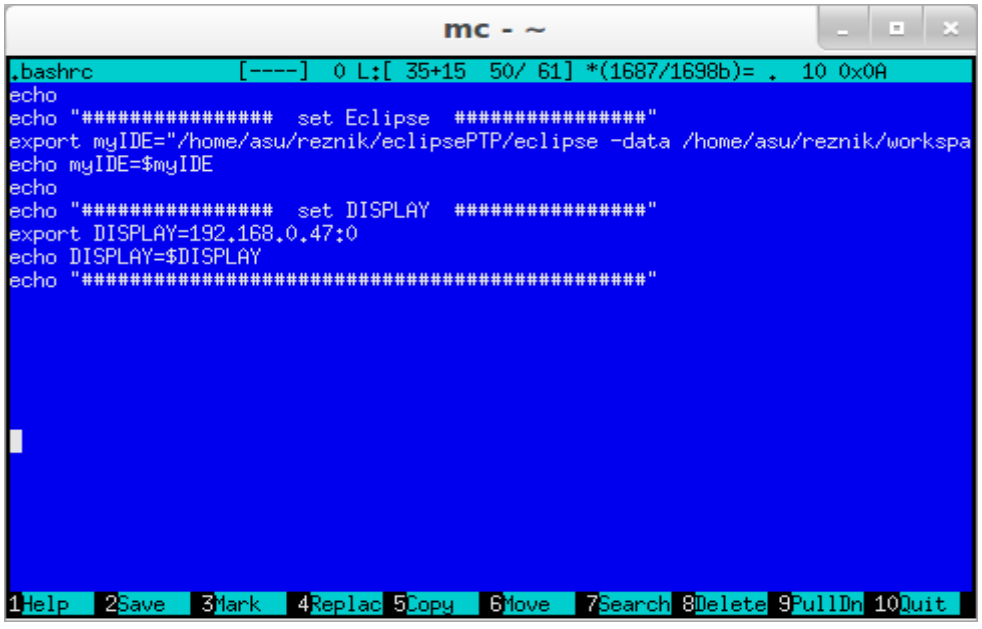

Рисунок 1.16 — Редактирование переменной среды myIDE

Примечание — Программа ***eclipse***, в качестве значения ключа **-data** принимает значение директории, которая будет рассматриваться как ***Workspace***. Студент должен правильно установить значение этого параметра.

Поскольку IDE Eclipse — сложная модульная система, первоначально созданная для разработки ПО на языке Java, то основная масса литературы, посвящённая Eclipse и примерам разработки на ней, содержит контекст для языка Java.

В последнее время, некогда единый проект интегрированной среды Eclipse, возглавляемый Eclipse Foudation, стал делиться на специализированные проекты, среди которых появилась среда разработки для ПО на C/C++ (среда CDT — C Development Tool).

Позже появился ещё один специализированный проект, поддерживающий разработку приложений с параллельными вычислениями (Eclipse IDE for Parallel Application Developers).

Другое название этого проекта — **Eclipse PTP** (***Eclipse Parallel Tools Platform***).

На кластере ЭВМ кафедры АСУ установлена среда разработки Eclipse PTP версии 3.7 (проект Indigo). Эта среда обеспечивает разработку ПО на C, C++ и Fortran.

1.3.2 Контроль выполнения работ по теме 1

Текущий и заключительный контроль выполнения лабораторных работ является обязательной составляющей процесса обучения.

Текущий контроль преподаватель проводит следующими способами:

1) индивидуальная регистрация посещения каждой лабораторной работы каждым студентом;
2) наблюдение общего хода выполнения отдельной лабораторной работы в течение отведённого для ее выполнения времени;
3) просмотр индивидуальных отчетов магистрантом и регистрация объёма выполненной работы;
4) оценка результатов, выполненных каждым магистрантом работ, по двум контрольным точкам.

За время обучения по дисциплине «Архитектура вычислительных комплексов», магистрант должен подготовить единый отчёт.

Шаблон отчёта расположен на рабочем столе пользователя ***upk***. Копию отчёта следует скопировать на личный flashUSB, отредактировать титульную страницу и регулярно заполнять в процессе выполнения всех лабораторных работ по данной дисциплине.

В целом, текст отчёта заполняется в произвольной форме, выбранной самим обучающимся.

2 АРХИТЕКТУРА ПРОЦЕССОРОВ

Вторая тема изучаемой дисциплины посвящена изучению архитектур современных микропроцессоров ЭВМ. Теоретический материал конкретизируется на различных концепциях и подходах, которые достаточно подробно описаны в литературе. Обязательным является изучение материала, изложенного в источнике [3]. Полученные теоретические знания студент закрепляет во время проведения трёх лабораторных работ.

2.1 Лабораторная работа №4. Компоненты аппаратного обеспечения ЭВМ

Лабораторные работы данной темы должны в какой-то мере закрепить теоретические представления об архитектурных особенностях процессоров ЭВМ, как главных компонент систем обработки данных (СОД).

Общие выводы, которые сразу же возникают после изучения теоретической части, можно выразить следующими утверждениями:

1) *само понятие процессор*, в процессе исторического развития, переходит в понятие микропроцессор, что делает его сложным архитектурным сооружением, имеющим все основные черты отдельного компьютера: наличие вычислительных и управляющих элементов, собственной памяти и средств коммуникации (шин), а также интерфейсов для взаимодействия с внешним окружением;
2) *архитектура современных микропроцессоров* имеет тенденцию включения в себя множества функциональных элементов, которые сами претендуют на роль отдельных процессоров, с возможностью самостоятельно обеспечивать вычислительные процессы компьютера (многоядерные архитектуры);
3) *вычислительные элементы современных микропроцессоров* имеют тенденцию к самостоятельному программированию (микропрограммированию), что порождает необходимость их асинхронного их взаимодействия.

Таким образом современную ЭВМ (рабочую станцию) можно рассматривать как многопроцессорный вычислительный комплекс (***МВК***) с общей шиной и общей разделяемой памятью (***ОЗУ***). Такой взгляд позволяет нам выполнить ряд лабораторных работ на отдельной ЭВМ, демонстрируя программные технологии, ориентированные на создание вычислительных комплексов.

В целом лабораторный практикум по данной теме включает три работы:

1) лабораторная работа №4: «***Компоненты аппаратного обеспечения ЭВМ***»;
2) лабораторная работа №5: «***Асинхронное взаимодействие на уровне виртуального терминала***»;
3) лабораторная работа №6: «***Асинхронный композитинг изображений на уровне нитей***».

Непосредственно в данной лабораторной работе, мы:

1) *изучим инструментальные средства* дляопределения состава аппаратного обеспечения компьютера;
2) *познакомимся с характеристиками ряда устройств ЭВМ*, такими как виртуальные терминалы, устройство мыши и устройство фреймбуфера, которые понадобятся нам для реализации программного обеспечения в последующих лабораторных работах.

2.1.1 Компоненты аппаратного обеспечения ЭВМ

Изучая архитектуру ЭВМ или просто занимаясь системным программированием, возникает необходимость в знании как состава, так и некоторых характеристик аппаратного обеспечения компьютера.

Для этих целей, имеется ряд утилит, которые, в той или иной степени, позволяют получить такую информацию:

- ***lscpu*** — информация об используемом процессоре;
- ***lshw*** — информация о всех устройствах ЭВМ;
- ***lspci*** — информация о шине ***pci***;
- ***lsscsi*** — (*не установлена*) — информация о контроллерах SCSI-устройств;
- ***lsusb*** — краткая информация об устройствах USB.

Много полезной информации о системе можно извлечь из файлов директории /proc, просмотрев некоторые из них утилитой ***cat***. Например:

- cat /proc/cpuinfo – информация о процессоре (CPU);
- cat /proc/meminfo – информация о памяти (ОЗУ);
- cat /proc/interrupts – прерывания;
- cat /proc/swaps – вся информация про swap;
- cat /proc/version – версия ядра и другая информация;
- cat /proc/net/dev – сетевые интерфейсы и статистика;
- cat /proc/mounts – смонтированные устройства;
- cat /proc/partitions – доступные разделы;
- cat /proc/modules – загруженные модули ядра.

Примечание — Студенту следует знать об имеющихся возможностях и изучить указанные выше утилиты с помощью руководства ***man*** или из других источников.

Для общего изучения состава аппаратного обеспечения ЭВМ следует воспользоваться утилитой ***lshw***, которая содержит достаточно много возможностей. В частности, эта утилита позволяет сформировать ***html***-страничку, которую затем можно просмотреть в любом браузере. Для этого, необходимо запустить эту утилиту с ключом ***-html***, например, можно создать файл формата ***html*** согласно выражения (2.1).

lshw -html > hw.html **(2.1)**

Файл ***hw.html*** можно просмотреть в браузере, например, как показано ниже на отдельном рисунке 2.1.

Другой способ изучения аппаратных средств ЭВМ — запуск утилиты ***lshw*** командой выражения (2.2) от имени пользователя ***root***.

sudo lshw -X **(2.2)**

В этом случае будет запущена графическая оболочка самой утилиты, что обеспечивает дополнительные возможности изучения аппаратных средств ЭВМ.

Задание на изучение компонент аппаратного обеспечения ЭВМ.

Изучить и отобразить в личном отчёте:

1) перечень компонентов рабочей станции, на которой работает студент;
2) выделить и описать характеристики одной из компонент ЭВМ.

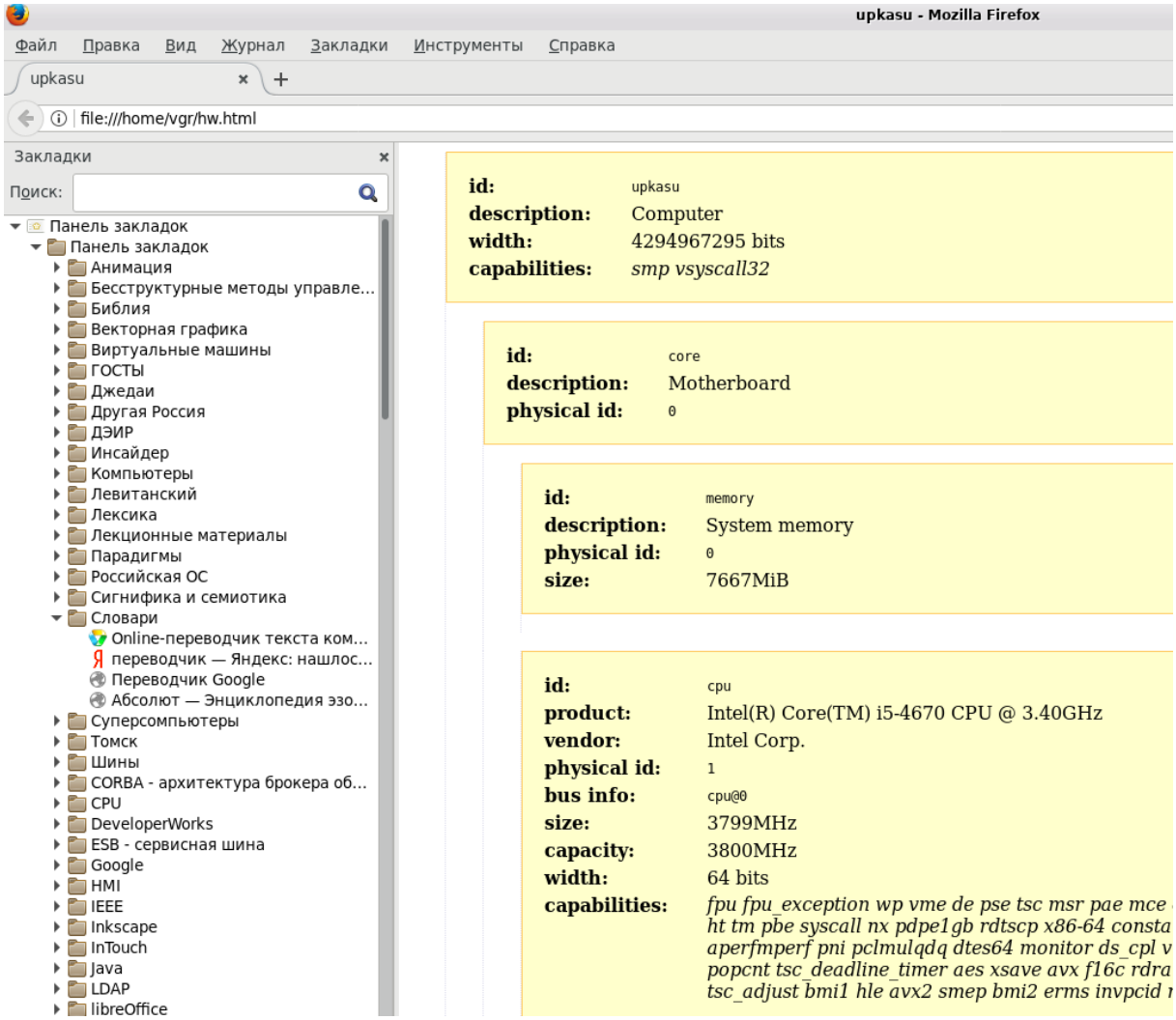

Рисунок 2.1 — Перечень аппаратного обеспечения ЭВМ, представленный html-страницей, полученной с помощью утилиты lshw

2.1.2 Характеристики виртуального терминала

Обычная рабочая станция (компьютер, ЭВМ) имеют: *клавиатуру*, *дисплей* и *графический адаптер*, обеспечивающий вывод текстовой и/или графической информации на дисплей.

Студенту должно быть известно, что перечисленные устройства (клавиатура, дисплей и графический адаптер) системно объединены понятием консоли (терминала).

Современное общее системное программное обеспечение, которое является частью любого комплекса ЭВМ, расширено понятием *виртуальных терминалов*, обеспечивающих взаимодействие с самим комплексом.

Стандартное системное ПО ОС Linux содержит *64 виртуальных терминала*, обозначенных устройствами:

- ***/dev/tty0*** — общий (управляющий терминал);
- ***/dev/tty1*** - ***/dev/tty63*** — виртуальные терминалы для непосредственного взаимодействия конечного пользователя и ПО ОС.

Переключение между виртуальными терминалами обеспечивается ядром ОС и осуществляется следующими способами:

1) *на уровне ядра ОС* — комбинациями клавиш ***Alt-F#*** (***Ctrl-Alt-F#***), где # — номер виртуального терминала;
2) *на уровне интерпретатора shell* — утилитой ***chvt #***, где **#** — номер терминала;
3) *на уровне языка С* (из текущего и уже открытого виртуального терминала) — системным вызовом ioctl(…), согласно синтаксису выражения (2.3).

```
#include <sys/ioctl.h>
int ioctl(0, VT_ACIVATE, #);
```
(2.3)

Чтобы выяснить, на каком терминале работает пользователь, необходимо использовать утилиту ***tty***, которая выведет на экран полное имя устройства терминала, в котором введена команда самой утилиты.

Когда пользователь вошёл в систему под любым именем, ему для взаимодействия с ЭВМ предоставляется некоторый виртуальный терминал, в котором запущен командный интерпретатор (***shell***), например, ***/bin/bash***.

Когда пользователь набирает некоторую строку символов в окне командного интерпретатора, то:

1) сами символы сохраняются во внутреннем буфере терминала, а на экран выводится «эхо» нажатых клавиш;
2) нажатие на клавишу <***Enter***> (символ с десятичным значением 10) является командой терминалу: «Передать введенную строку интерпретатору shell для исполнения».

В целом подобный режим работы с терминалом называется «*каноническим*» и обеспечивает только интерактивное взаимодействие пользователя с ПО ЭВМ на уровне командных строк или на уровне запуска файлов сценариев, содержащих наборы различных команд.

Примечание — Терминал, на котором пользователь провёл вход в систему посредством утилиты ***login***, является для всех программ (процессов) пользователя управляющим терминалом.

Использование конечным пользователем устройств терминала накладывает на запуск и выполнение всех его программ следующие ограничения:

1) *утилита* ***login*** проводит идентификацию и авторизацию пользователя, создаёт сессию и среду для выполнения всех программ, запущенных с устройства терминала; дочерний процесс утилиты ***login*** понижает свой приоритет до уровня пользователя, а затем запускает вместо себя («*меняет тело*») командный интерпретатор;
2) *в случае закрытия терминала*, все программы (процессы) пользователя, запущенные с этого терминала, останавливаются, прекращают свою работу (уничтожаются) и сессия закрывается; нормальное закрытие терминала осуществляется последовательностью команд ***exit***;
3) *каждый процесс*, запущенный в терминале, имеет системный (*стандартный*) ввод/вывод, который доступен через дескрипторы устройств: 0 — ввод с клавиатуры; 1 — нормальный вывод на экран терминала; 2 — канал ошибок.

В общем случае управление режимами терминала представляет далеко не тривиальную задачу, поскольку требует знания многих системных вызовов и работу со структурой ***termios***, описание которой приведено в файле <***bits/termios.h***>. Для изучения структуры и функций работы со структурой termios, а также различных вызовов функции ***ioctl(…)*** следует воспользоваться руководством man, согласно выражений (2.4) — (2.6).

```
man termios
```
(2.4)

```
man ioctl
```
(2.5)

```
man console_ioctl
```
(2.6)

2.1.3 Учебный проект avk_tty

В пределах данной лабораторной работы рассмотрим реализацию на языке С набора функций, который именуем как проект ***avk_tty***.

Цель проекта — демонстрация ПО, обеспечивающего:

1) *переключение* терминала в асинхронный режим взаимодействия с пользователем;
2) *отслеживание* переключений между виртуальными терминалами;
3) *анализ* ввода данных при работе с клавиатурой.

Примечание — Программное обеспечение данного проекта ***avk_tty*** будет использовано в последующих лабораторных работах.

Исходный текст проекта ***avk_tty*** представлен на листинге 2.1.

Листинг 2.1 — Исходный текст проекта avk_tty

```
/*
 ============================================================================
 Name        : avk_tty.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

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

#include <unistd.h>
#include <sys/vt.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <string.h>


/**
 * Объявление структуры состояния терминала.
 */

typedef struct _avk_tty {
      struct termios oldtty;  // Старое состояние терминала.
      struct termios newtty;  // Новое  состояние терминала.
      int         changed;    // Статус состояния терминала.

} avk_tty_t;


/**
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_tty_t *p){
      /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
```

```
      printf("\033[2J\033[H\n");
      /**
       * Включаем аппаратный курсор
       */
     printf("\033[?25h\n");

    if(p->changed)
      /**
       * Возвращаем состояние терминала.
       */
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
    /**
     * Освобождаем память, выделенную структуре.
     */
    free(p);
}

/**
 * Установка нового состояния теринала.
 */
avk_tty_t *
avk_setup_tty(void){
      /**
       * Выделяем память под структуру.
       */
      avk_tty_t *p = malloc(sizeof(avk_tty_t));
    if(p == NULL) return NULL;

    /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Выключаем аппаратный курсор.
       */
     printf("\033[?25l\n");
    /**
     * Сохраняем состояние терминала.
     */
    tcgetattr(STDIN_FILENO, &p->oldtty);
    tcgetattr(STDIN_FILENO, &p->newtty);
    /**
     * Переводим терминал в новое состояние:
     * отключаем канонический режим и эхо.
     */
    p->newtty.c_lflag &= ~(ICANON | ECHO | ECHOE );
    if(tcsetattr(STDIN_FILENO, TCSANOW, &p->newtty) < 0){
        /**
         * Если - проблема!!!
         * Возвращаем старое состояние терминала.
         */
        printf("Не могу установить терминал...\n");
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
        p->changed = 0;
    }else
        p->changed = 1;
    return p;
}
```

```
/**
 * Получить номер текущей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd)
{
      struct vt_stat vs;

      if(ioctl(fd, VT_GETSTATE, &vs) < 0)
            return 0xffff;

      return(vs.v_active);
}

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */

size_t
avk_read_tty(void *buff, size_t length){
    ssize_t L = 0;
    fd_set rfds;
    struct timeval tv;
    int retval;
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /**
     * Ожидаем не более 20ms: 50 раз/сек
     */
    tv.tv_sec = 0; tv.tv_usec = 20*1000;
    /**
     * Проверяем наличие прочитанных данных.
     */
    retval = select(1, &rfds, NULL, NULL, &tv);
    if (!retval) return 0;
    /**
     * Читаем данные, если они доступны.
     */
    if(FD_ISSET(0, &rfds)){
        if((L = read(0, buff, length)) < 1) return 0;
    }
    return (size_t)L;
}


/**
 * Программа циклического асинхронного чтения данных
 * с терминала, при вводе данных с клавиатуры.
 *
 * Байты каждой нажатой клавиши распечатываются
 * отдельной строкой в десятичном виде.
 * Завершение работы программы: Ctrl-x.
 */
```

```
int main(void) {

	/**
	 * Объявляем буфер для чтения данных.
	 */
	unsigned char buff[25];
	/**
	 * Запоминаем стартовый номер
	 * виртуального терминала.
	 */
	unsigned short nv = avk_get_current_vc(0);
	/**
	 * Переводим терминал в асинхронный режим.
	 */
	avk_tty_t * set = avk_setup_tty();
	if(set == NULL){
		printf("Не получилось установить терминал\n");
		return 1;
	}

	printf("Работаем с виртуальным терминалом №%i\n", nv);
	size_t nc;
	int ne = 0; // Когда ne == 1, то завершение работы программы.
	/**
	 * Цикл чтения данных с клавиатуры.
	 */
	while(1){
		/**
		 * Если терминал переключен,
		 * то ожидаем его активации.
		 */
		if(nv != avk_get_current_vc(0)){
		    /**
		     * Запускаем ioctl() для ожидания активности терминала!!!.
		     */
		ioctl(0, VT_WAITACTIVE, nv);
		printf("\nТерминал №%i снова стал активным...\n", nv);
		}
		if((nc = avk_read_tty(buff, 25)) == 0){
			usleep(20*1000);
			continue;
		}
		/**
		 * Печатаем данные клавиатуры.
		 */
		for(int i=0; i<nc; i++)
			if((unsigned int)buff[i] == 24){
				ne = 1;                      // На выход.
				break;
			}
			else  printf(" %i", (int)buff[i]); // Печать байта.
		printf("\n");
		if(ne == 1)
			break;
	}

	/**
	 * Восстанавливаем каноническое
	 * состояние терминала.
	 */
```

```
    avk_restore_tty(set);
    printf("Завершили работу с виртуальным терминалом №%i\n", nv);


    return EXIT_SUCCESS;
}
```

Задание на реализацию проекта avk_tty.

1. По листингу 2.1 изучить назначение и алгоритмы работы функций.
2. В EclipseC открыть проект ***avk_tty*** и перенести в него исходный текст.
3. Провести компиляцию и отладку проекта.
4. Перейти на виртуальный терминал ***/dev/tty3*** и войти на нем в систему от имени пользователя ***upk***.
5. Запустить и исследовать программу работы проекта.
6. Отобразить полученные результаты и знания в личном отчёте.

Примечание — Запуск проекта ***avk_tty*** должен осуществляться на уровне виртуального терминала, что делает невозможным использование файлового менеджера ***Midnight Commander***, поскольку он предоставляет пользователю псевдотерминал. Указанный факт можно проверить, выполнив команду ***tty***.

Чтобы устранить указанное неудобство, можно сделать символьную ссылку на запускаемый файл проекта, например, так:

1) запустить терминал и перейти в директорию ***~/bin*** командой: ***cd ~/bin*** ;
2) сделать символьную ссылку командой выражения ().

```
ln -s ~/workspaceC/avk_tty/Debug/avk_tty  avk_tty          (2.7)
```

Тогда проект будет запускаться из командной строки сразу командой: ***avk_tty***

2.1.4 Классическое устройство мыши. Проект avk_mouse

Другим компонентом ЭВМ, которое легко подключить и использовать, является устройство мыши.

Хотя современный подход к проектирования ОС Linux относит устройство мыши к классу устройств типа ***input*** и рекомендует работать с ним на основе анализа событий (***eventX***). Для этого имеется системное устройство ***/dev/input/mouse0***, которое поддерживает старый протокол для устройств, подключённых к интерфейсу типа ***PS/2***.

Примечание — Устройство ***mouse0*** является первым устройством мыши, которое увидел и создал демон ***udevd***. Следующее подключённое устройство мыши будет называться ***mouse1*** и так далее.

Учебная цель данного пункта работы — изучение работы устройства мыши посредством реализации программного проекта с именем ***avk_mouse***.

Исходные условия для реализации проекта ***avk_mouse***:

1) само устройство ***mouse0*** является символьным устройством доступным для чтения и записи владельцу ***root*** и группе ***input***;
2) пользователь ***upk*** должен быть включён в группу ***input*** ОС, тогда сможет использовать это устройство;
3) как компонент ЭВМ, устройство мыши предназначено для асинхронного взаимодействия с ним;

4) как источник информации, это устройство выдаёт состояние трёх кнопок и относительное перемещение самого устройства в системе координат ***(x,y)***;
5) величины координат ***(x,y)*** поддерживаются стандартным драйвером ОС Linux и сохраняются в структуре ***input_event***.

Исходный текст проекта ***avk_mouse***, учитывающий указанные выше условия работы с устройством мыши, представлен на листинге 2.2.

Листинг 2.2 — Исходный текст проекта avk_mouse

```
/*
 ============================================================================
 Name        : avk_mouse.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

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

#include <fcntl.h>
#include <linux/input.h>
#include <signal.h>
#include <errno.h>

#define MOUSEFILE "/dev/input/mouse0"

/**
 * Дескриптор устройства мыши.
 */
int fd;
/**
 * Функция обработки сигнала SIGINT.
 */
void
sig_handler(int signal){
      printf("\nПолучен сигнал SIGTERM = %i\n",
                  signal);
    close(fd);
}

/**
 * Головная функция проекта.
 */
int main(void)
{
      char *title =
                  "Проект avk_mouse:";
    /**
     * Структура, читаемая с устройства мыши.
     */
    struct input_event ie;
    /**
     * Состояния кнопок устройства мыши.
     */
```

```
unsigned char button,bLeft,bMiddle,bRight;
/**
 * Относительные изменения координат мыши.
 */
char x,y;
/**
 * Абсолюные значения координаты мыши.
 */
int absolute_x = 0, absolute_y = 0;
/**
 * Открываем устройство мыши
 * для чтения с блокировкой.
 */
if((fd = open(MOUSEFILE, O_RDONLY)) == -1) {
    printf("%s\tОшибка открытия устройства мыши...\n",
              title);
    exit(EXIT_FAILURE);
}
else
    printf("%s\tУстройство мыши - открыто...\n",
              title);

/**
 * Активируем обработчик сигнала.
 */
signal(SIGTERM, &sig_handler);
  /**
   * Указатель на читаемую структуру.
   */
unsigned char *ptr = (unsigned char*)&ie;

  /**
 * Цикл вывода нажатия кнопок и
 * относительных координат мыши.
 *
 * Нажатие левой кнопки мыши выводит
 * абсолютные значения координат.
 *
 * Завершение работы программы осуществляется
 * кнопкой останова на экране консоли Eclipse.
 */
while(1){
    /**
     * Засыпаем на 20 ms
     */
    usleep(20*1000);

  /**
   * Читаем структуру данных мыши.
   */
  if(read(fd, &ie, sizeof(struct input_event)) < 0){
      printf("\n%s\tЗавершение работы...\n",
              title);
        break;
  }
    /**
     * Первый байт структуры - состяния кнопок.
     */
    button=ptr[0];
    bLeft = button & 0x1;
```

```
        bMiddle = ( button & 0x4 ) > 0;
        bRight = ( button & 0x2 ) > 0;

        /**
         * Относительные значения координат.
         */
        x=(char) ptr[1];
        y=(char) ptr[2];
        printf("bLEFT:%d, bMIDDLE: %d, bRIGHT: %d, rx: %d  ry=%d\n",
                  bLeft,bMiddle,bRight, x,y);
        /**
         * Вычисление абсолютных координат мыши.
         */
        absolute_x+=x;
        absolute_y-=y;

        if (bLeft == 1) // Если нажата левая кнопка.
            printf("\nАбсолютные координаты TOP_LEFT = %i %i\n\n",
                        absolute_x, absolute_y);
    }

    return 0;
}
```

Задание на реализацию проекта avk_mouse.

1. По листингу 2.2 изучить общий алгоритм работы с устройством мыши.
2. В EclipseC открыть проект ***avk_mouse*** и перенести в него исходный текст.
3. Провести компиляцию и отладку проекта.
4. Запустить и исследовать программу работы проекта, прямо в самой среде разработки, поскольку она не изменяет характеристики терминала.
5. Отобразить полученные результаты и знания в личном отчёте.

2.1.5 Устройство фреймбуфера. Проект avk_fb

Фреймбуфер (*Linux Frame Buffer Device*) — это символьное устройство, представляющее графический аппаратно-независимый уровень абстракций для вывода графики на *консоль* (*терминал*) ОС Linux.

Фреймбуфер является оригинальным устройством ОС Linux. Впервые, он появился в версии ***Linux 2.1.107*** (приблизительно ***1997 — 1999 годы***) и предназначался для эмуляции текстовой консоли на системах ***Apple Macintosh***, у которых не было текстового режима. Позднее он распространился на версии Linux, которые предназначены для компьютеров типа ***IBM PC***.

Для нас это устройство интересно как эмуляция аппаратной компоненты ЭВМ, с которой можно *работать в асинхронном режиме*, наглядно демонстрируя графические результаты взаимодействия с системой без использования жёстких ограничений, присущих доступу к общим разделяемым ресурсам.

> Примечание — Возможности устройства фреймбуфера демонстрируются в учебном пособии [5] на примере реализации подсистемы ***login*** на виртуальных терминалах ***/dev/tty1*** и ***/dev/tty2***. Имеетя также другая демонстрационная программа, которая доступна на виртуальном терминале ***/dev/tty5***.

В среде ОС УПК АСУ фреймбуфер доступен как символьное устройство ***/dev/fb0***, разрешающее чтение и запись владельцу ***root*** и группе пользователей ***video***. Поскольку пользователь ***upk*** уже входит в группу ***video***, то нет никаких ограничений в использовании этого

устройства, как в учебных целях, так и при разработке приложений в среде ОС Linux. Это утверждение демонстрируется примерами использования устройства ***/dev/fb0***, при загрузке системы ОС УПК АСУ:

- показ логотипа кафедры АСУ на чёрном фоне экрана, на первом этапе загрузки, когда ещё недоступна корневая файловая система с его библиотеками;
- показ динамической заставки логотипа кафедры АСУ на синем фоне, на втором этапе загрузки, когда корневая файловая система ОС уже доступна, но система ещё полностью не загружена даже в текстовом режиме доступа.

Являясь символьным устройством, фреймбуфер может быть открыт как обычный файл с возможностью чтения/записи в него обычными функциями ***read()*** и ***write()***. Тем не менее, в отличии от достаточно простого устройства мыши, устройство фреймбуфера требует предварительной настройки и учёта следующих ограничений:

1) фреймбуфер является единственным устройством на все виртуальные терминалы ОС и может быть открыт в каждом из них;
2) фреймбуфер синхронизирован с переключениями на каждый виртуальный терминал ОС, что имеет последствия очистки его содержимого, в момент переключения, и отсутствие восстановления содержимого, при восстановлении активности терминала;
3) характеристики фреймбуфера тесно связаны с графическими режимами виртуальных терминалов, на которые они настроены и включают в себя такие основные параметры как: *пиксельные размеры экрана* по горизонтали и вертикали, *количество байт* для представления одной строки экрана (*stride*), *количество байт для представления одного пикселя* и *формат представления пикселя*; имеются и другие параметры, учитывающие особенности мониторов ЭВМ и графических карт.

Несмотря на кажущуюся излишнюю сложность, в современных ОС Linux, технология работы с фреймбуфером достаточно хорошо отработана:

- все необходимые параметры фреймбуфера сосредоточены в двух базовых структурах с именами: ***fb_var_screeninfo*** и ***fb_fix_screeninfo***;
- указанные структуры извлекаются из системы и устанавливаются в неё с помощью надёжного набора системных вызовов ***ioctl()***, описание которых можно найти в заголовочном файле <***linux/fb.h***>;
- на большинстве современных компьютеров можно ограничиться 32-битным представлением одного пикселя, в котором три байта задают цвета красный, зелёный и синий, а четвёртый байт используется графическими библиотеками для отображения прозрачности изображения.

Учебная задача данного пункта работы — закраски окна монитора синим цветом и отображения на этом фоне динамической картинки циклического перемещения серого прямоугольника.

Реализация учебной задачи — программный проект ***avk_fb***, исходный текст которого представлен на листинге 2.3.

Листинг 2.3 — Исходный текст проекта avk_fb

```
/*
 ============================================================================
 Name        : avk_fb.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint-gcc.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/fb.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <signal.h>

/**
 * Глобальные переменные.
 */
int fd;                 // Дескриптор устройства фреймбуфера.
size_t nbuf;            // Длина фреймбуфера в байтах.
uint32_t * screen;      // Выровненный по целому числу указатель
                        // фреймбуфер.
int if_exit = 0;        // Флаг завершения.

/**
 * Функция обработки сигнала SIGINT (Ctrl-C).
 */
void
sig_handler(int signal){
      printf("\nПолучен сигнал SIGINT = %i\n",
                  signal);
      if_exit = 1;
}

/**
 * Головная программа.
 */
int
main(int argc, char * argv[]) {
      char * title =
                  "Проект avk_fb:";
      printf("\n%s\tНачало работы с фреймбуфером /dev/fb0\n",
                  title);
      /**
       * Параметры фреймбуфера:
       */
      uint32_t nWidth  = 1920;            // Количество пикселей в строке
      uint32_t nHeight = 1200;            // Количество строк
      uint32_t nBits   = 32;              // Количество бит в пикселе
      uint32_t color32 = 0x000000ff;      // Цвет заполнения всего экрана.


      /**
       * Открываем устройство фреймбуфера для
       * чтения и записи.
       */
      fd = open("/dev/fb0", O_CLOEXEC | O_RDWR);
      if(fd < 0){
            printf("%s\tНе могу открыть файл: /dev/fb0\n",
                        title);
            return -1;
      }
```

```
/**
 * Определяем параметры монитора.
 */
struct fb_fix_screeninfo fix;
if(ioctl (fd, FBIOGET_FSCREENINFO, &fix) == -1){
      printf("%s\tНе могу получить информацию о буфере...\n",
                  title);
      close(fd);
      return -1;
}
/**
 * Длина строки экрана в пикселях:
 * по 4 байта на пиксель.
 */
nWidth = fix.line_length/4;

/**
 * Читаю формат пикселя в битах.
 */
struct fb_var_screeninfo var;
var.bits_per_pixel = 0;
ioctl (fd, FBIOGET_VSCREENINFO, &var);
/**
 * Если формат не 32 бита на пиксель, то
 * переустанавливаю это значение.
 */
if(var.bits_per_pixel != 32){
      var.bits_per_pixel = 32;
      ioctl (fd, FBIOPUT_VSCREENINFO, &var);
}
/**
 * Снова (для контроля) читаю структуру.
 */
if(ioctl (fd, FBIOGET_VSCREENINFO, &var) == -1){
      printf("%s\tНе могу получить информацию о буфере...\n",
                  title);
      close(fd);
      return -1;
}
/**
 * Печатаю полученные параметры на экран.
 */
printf("\tШирина экрана:  %d пикселей\n", nWidth);
printf("\tВидимая ширина: %d пикселей\n", var.xres);
printf("\tВысота экрана:  %d пикселей\n", nHeight = var.yres);
printf("\tЧисло  бит:     %d пикселя\n",  nBits   = var.bits_per_pixel);

if(nBits != 32){
      printf("%s\tНа такой экран - не записываю!!!\n",
                  title);
      close(fd);
      return -1;
}
/**
 * С фреймбуфером можно работать как с последовательным
 * символьным файлом, но лучше - как с матрицей целых чисел,
 * поэтому проводим mmap() на память ЭВМ.
 */
nbuf =
```

```
                nHeight*fix.line_length; // Длина фреймбуфера в байтах.
    screen = (uint32_t *)mmap (0, nbuf, PROT_READ |
                PROT_WRITE, MAP_SHARED,
                fd, 0);
    if(screen <= 0){
          printf("%s\tНе могу сделать mmap() на фреймбуфер!!!",
                      title);
          close(fd);
          return -1;
    }
    /**
     * Показываем результат вывода на экран.
     */
    printf("\nЗавершение работы программы %s\n",
                "- комбинация клавиш: Ctrl-C");
    printf("Нажми клавишу Enter...");
    getchar();

/**
 * Активируем обработчик сигнала.
 */
signal(SIGINT, &sig_handler);

    /**
     *  Заполняем фоновым цветом весь экран.
     */
    for(int i=0; i<nWidth*nHeight; ++i)
          screen[i] = color32;

    /**
     * Определяю параметры динамической части рисунка:
     * 5 позиций квадрата 40x40 пикселей каждый.
     */
    uint32_t bg = 0x007f7f7f;     // Цвет фона.
    uint32_t fg = 0x00007f00;     // Цвет заполнения.
    uint32_t t0;                  // Текущий цвет заполнения.
    uint32_t w0 = 40;             // Ширина прямоугольника.
    uint32_t h0 = 40;             // Высота прямоугольника.
    uint32_t n0 = 5;              // Число  прямоугольников.
    uint32_t k0 = 0;              // Текущий прямоугольник.
    uint32_t m0;                  // Ширина запонения в пикселях.
    /**
     * Координаты левого верхнего угла динамического изображения.
     */
    uint32_t x0 = (nWidth  - 5*40)/2;
    uint32_t y0 = nHeight/2;

    /**
     * Циклы отрисовки динамической части изображения.
     */
    while(1){
          if(if_exit)
                break;
          if(k0 == 0){
                t0 = bg;
                m0 = n0*w0;
          }else{
                t0 = fg;
                m0 = k0*w0;
          }
```

```
            for(int i=0; i<h0; ++i){
                  for(int j=0; j<m0; ++j)
                        screen[(y0 + i)*nWidth + x0 + j] = t0;
            }
            k0++;
            if(k0 > n0)
                  k0 = 0;
            /**
             * Задержка 200 ms
             */
            usleep(200*1000);
      }
      /**
       * Завершаю работу.
       */
      printf("%s\tЗавершаю работу...\n",
                  title);
      munmap(screen, nbuf);
      close(fd);

      return EXIT_SUCCESS;
}
```

Задание на реализацию проекта avk_fb.

1. По листингу 2.3 изучить общий алгоритм работы с устройством фреймбуфера, разобрав все этапы его инициализации.
2. В EclipseC открыть проект ***avk_fb*** и перенести в него исходный текст программы.
3. Провести компиляцию и отладку проекта.
4. Запускать и исследовать проект следует на отдельном виртуальном терминале, например, ***/dev/tty3***.
5. Исследовать работу программы проекта, сравнивая ее поведение с характеристиками, приведёнными выше в данном подразделе.
6. Обязательно, в процессе работы программы, произвести переключение на различные виртуальные терминалы.
7. Отобразить полученные результаты и знания в личном отчёте.

После подведения итогов и оформления записей в личном отчёте, лабораторную работу №4 можно считать законченной.

Подводя итог проделанной работы, мы видим:

1) классические представления о командной строке (терминале) сформированы с целью обеспечения нормальной работы интерпретаторов ***shell***; отключая канонический режим терминала и блокирующее чтение, можно организовать взаимодействие с ЭВМ и ее ОС в ***асинхронном режиме***, что соответствует основному подходу в разработке системного ПО;
2) изучение работы таких общедоступных устройств как ***устройство мыши*** и ***устройство фреймбуфера*** имеет целью расширить теоретические представления и обеспечить практические навыки работы с отдельными компонентами компьютера, попутно демонстрируя ряд доступных технологических возможностей работы с устройствами ОС.

2.2 Лабораторная работа №5. Асинхронное взаимодействие на уровне виртуального терминала

Цель данной лабораторной работы — расширение знаний и практических навыков системного программирования на основе знаний, которые получены в предыдущей лабораторной работе.

Методологической основой системного программирования является *синхронизация* (***упорядочение взаимодействия***) *асинхронно работающих компонент*. Мы в качестве объектов взаимодействия будем рассмотривать уже изученные нами устройства *виртуального терминала*, *мыши* и *фреймбуфера*.

> Примечание - Формально устройство терминала состоит из набора других устройств, таких как клавиатура, дисплей и графический адаптер, о чем постоянно говорится в тексте методического пособия, тем не менее, эти устройства упоминаются лишь в контексте, для пояснения проводимых действий или объяснения алгоритмов работы программ.

Реально, мы работаем с устройствами виртуальных терминалов ***/dev/ttyX***, что было продемонстрировано результатами предыдущей работы.

На самом деле, никакого противоречия здесь нет, поскольку, как было показано в предыдущей теме, - архитектура и состав компонент этой архитектуры определяется уровнем абстракции, на котором рассматривается система. Кроме того, на системном уровне ОС модель устройства является самодостаточным объектом, который может рассматриваться как самостоятельный компонент системы с присущими ему свойствами и взаимосвязями с другими устройствами.

Учитывая вышесказанное и главенствующую роль модели терминала, как элемента взаимодействия пользователя с ПО ЭВМ, тема данной лабораторной работы обозначена как «*Асинхронное взаимодействие на уровне виртуального терминала*».

2.2.1 Асинхронное взаимодействие на уровне виртуального терминала

Прежде чем приступать к непосредственному обсуждению заявленной тематики данного подраздела, уточним семантику трёх понятий ***управление***, ***мониторинг*** и ***композитинг***.

Управление.

Управление — это процесс вызванный чьим-то воздействием на объекты и/или субъекты внешнего мира для достижения некоторой цели (***целей***).

В технических науках, понятие управления формализуется до уровня «модели управления с обратной связью», что конкретизировано научной дисциплине «Теория автоматического управления» и даже легло в основу такого научного направления, известного как «Кибернетика».

Развитие средств вычислительной техники и внедрение в процессы управления субъекта (человека) породило термин *АСУ*.

АСУ — автоматизированная система управления, где в контур управления внедрён человек, который в силу своей субъективности берет на себя и часть целевых установок системы.

В России, термин и понятия АСУ были закреплены ГОСТ серии ***24***.

Усиление человеческой целевой компоненты и, как следствие, «размывание» строгой целевой установки породило новый термин *АС*.

АС — автоматизированная система, что закреплено в ГОСТ серии ***34***, фактически убирает строгое и чёткое определение целевой установки, заменяя её *набором функций*, которым должна удовлетворять система.

Именно описание набора функций составляет содержание любого «***Технического задания*** (ТЗ)» на систему, которая по традиции продолжает называться АСУ.

Мониторинг.

Мониторинг — непрерывный процесс наблюдения и регистрации параметров объекта, в сравнении с заданными критериями.

Именно мониторинг является сутью всех информационных систем, включая диспетчерское управление, реализованное в системах, имеющих англоязычную аббревиатуру ***SCADA***. Тем не менее, по традиции, такие системы продолжают называться АСУ, например, «***АСУ технологическими процессами ...***».

Композитинг.

Развитие модельных представлений о системах, в которых интенсивно применяются средства вычислительной техники, привели к понятию модели ***SOA***.

SOA (*Service-Oriented Architecture*) — сервис ориентированная архитектура или модульный подход к разработке программного обеспечения, основанный на использовании распределенных, слабосвязанных компонентов, оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам.

В одном из направлений этой архитектуры, связанной с Web-технологиями, появляются такие экзотические термины как оркестровка и хореография.

Оркестровка — понятие относящееся к исполняемому бизнес-процессу, который может взаимодействовать как с внешними, так и с внутренними Web-сервисами.

Хореография — понятие относящееся к взаимодействию указанных бизнес процессов на уровне протоколов обмена сообщениями.

На фоне указанных определений, термин ***композитинг*** применяется в основном в технологиях обработки изображений.

Композитинг (*compositing*) — комбинированная съёмка (наложение) двух или более изображений, созданных независимо друг от друга.

Применительно к вычислительной технике, имеется даже специальное ПО, называемое композиторами, которое занимается наложением окон, в графической подсистеме компьютера.

Если проанализировать задачи и алгоритмы композитинга, то можно заметить большое их сходство с задачами создания системного ПО:

1) системное ПО работает с отдельными, достаточно независимыми по технологии функционирования устройствами: процессоры, клавиатура, мышь, экран монитора и другие компоненты отдельной ЭВМ или комплекса средств вычислительной техники;
2) задачи системного ПО связаны с согласованием (синхронизацией) работы отдельных компонент системы.

Таким образом, в пределах задач изучаемой дисциплины, будет использоваться следующая семантика термина *композитинг*.

Композитинг — системная программная технология, предназначенная для решения задач синхронизации взаимодействия асинхронно работающих устройств вычислительного комплекса.

2.2.2 Задача раскраски экрана монитора с помощью устройства мыши. Проект fb_monitor

Учебная задача данной лабораторной работы — написание программы закрашивающей экран монитора, некоторым *заданным цветом*, при выполнении следующих дополнительных условий:

- *отслеживание* переключений виртуальных терминалов, с целью предотвращения вмешательства в их деятельность;
- отслеживание положения курсора мыши, с целью зарисовки области ***20x40*** пикселей заданным цветом;
- *отслеживание* нажатий на клавиши устройства клавиатуры, с целью обнаружения команды завершения работы программы, которая соответствует нажатию комбинации клавиш ***Ctrl-X***;
- *вывод* на экран монитора динамического рисунка изображения, как это было продемонстрировано в предыдущей лабораторной работе;
- *реализация* задачи синхронизации всех процессов проектом ***avk_fb_monitor***;
- *проведение* исследования работы данного проекта;
- *описание* в отчёте полученных результатов.

Учебная технология программной реализации поставленной задачи:

1) *выделение* отдельных частей программного обеспечения, соответствующего каждой асинхронно работающей компоненте проекта;
2) *объединение* (композитинг) отдельных частей ПО проекта с помощью главной функции: функции-монитора.

2.2.3 Формализация компонент взаимодействующих устройств

В соответствии с требованиями поставленной задачи, откроем в среде разработки EclipseC проект ***avk_fb_monitor*** и приступим к выделению отдельных компонент программного обеспечения для решения поставленной задачи.

С целью лучшего разделения самого процесса реализации задачи, будем рассуждения и выполняемые действия выделять отдельными этапами.

Этап 1. Формирование структуры контекста задачи.

Программное обеспечение, реализованное в предыдущей лабораторной работе, содержит описание и объявление различных языковых объектов, поэтому в проекте создадим общий заголовочный файл ***avk_monitor.h***, в который включим:

- *общую часть* всех использованных ранее заголовочных файлов;
- *объявления* всех созданных ранее функций, не требующих изменения, кроме функций ***main()***;
- *определение* общей контекстной структуры типа ***avk_context_t***, в которую включим все нужные глобальные переменные, необходимые всему проекту в целом.

Первый вариант заголовочного файла ***avk_monitor.h*** представлен на листинге 3.1, в который также включены объявления функций проекта ***avk_tty***.

Листинг 2.4 — Заголовочный файл проекта avk_fb_monitor

```
/*
 * avk_monitor.h
 *
 *  Created on: 15 авг. 2017 г.
 *      Author: upk
 */

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

#include <sys/vt.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <string.h>

#include <fcntl.h>
#include <linux/input.h>
#include <signal.h>
#include <errno.h>

#include <stdint-gcc.h>
#include <linux/fb.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#ifndef AVK_MONITOR_H_
#define AVK_MONITOR_H_

#define MOUSEFILE "/dev/input/mouse0"
#define FBFILE    "/dev/fb0"

/**
 * Объявление структуры состояния терминала.
 */
typedef struct _avk_tty {
      struct termios oldtty;  // Старое состояние терминала.
      struct termios newtty;  // Новое  состояние терминала.
      int            changed; // Статус состояния терминала.

} avk_tty_t;

/**
 * Обявление общей контекстной структуры проекта.
 */
typedef struct _avk_contect {
      /**
       * Данные виртуального терминала.
       */
      unsigned short    nvt;        // Номер виртуального  терминала.
      avk_tty_t         *tty_status;// Структура состояния терминала.
      char              tbuf[10];   // Буфер терминала.
      /**
       * Данные фреймбуфера.
       */
      int fdf;                // Дескриптор фреймбуфера.
```

```
        int x, y, w, h;         // Размеры окна фреймбуфера.
        int         stride;     // Длина строки окна в байтах.
        uint32_t    color32;    // Цвет заполнения всего экрана  = 0x00ff
        uint32_t    * screen;   // Выровненный по целому числу указатель
                                // на фреймбуфер.
        /**
         * Данные курсора.
         */
        int fdc;                        // Дескриптор курсора.
        int xc, yc;                     // Координаты курсора.
        int wc;                         // Ширина курсора = 20.
        int hc;                         // Высота курсора = 40.
        int bLeft, bMiddle, bRight;     // Состояние кнопок мыши.
        uint32_t colorC;                // Цвет рисования курсором = 0x007f7f00.
        /**
         * Данные динамической части рисунка.
         */
        uint32_t bg;            // Цвет фона = 0x007f7f7f.
        uint32_t fg;            // Цвет заполнения = 0x00007f00.
        uint32_t x0, y0;        // Координаты начала окна.
        uint32_t w0;            // Ширина прямоугольника = 40.
        uint32_t h0;            // Высота прямоугольника = 40.
        uint32_t n0;            // Число  прямоугольников = 5.
        uint32_t k0;            // Текущее состояние.

} avk_context_t;


/** ********************************************************
 * Объявления функций.
 **********************************************************/
/**
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_tty_t *p);

/**
 * Установка нового состояния терминала.
 */
avk_tty_t *
avk_setup_tty(void);

/**
 * Получить номер текущей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd);

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */
```

```
size_t
avk_read_tty(void *buff, size_t length);


#endif /* AVK_MONITOR_H_ */
```

В дальнейшем, данный заголовочный файл должен обновляться описателями новых функций, которые будут созданы на следующих этапах реализации проекта.

Этап 2. Компонента ПО виртуального терминала.

Переносим в наш проект полезную часть ПО, которое уже разработано в рамках проекта ***avk_tty***.

Поскольку проект ***avk_tty*** достаточно хорошо структурирован по функциям, описания которых уже включены в заголовочный файл ***avk_monitor.h***, то набор необходимых действий достаточно прост:

1) создаём в нашем проекте ***Source File*** с именем ***avk_tty.c*** и копируем из проекта ***avk_tty*** содержимое аналогичного файла;
2) удаляем из текста скопированного файла содержимое функции ***main()*** и объявление структуры ***avk_tty_t***;
3) все подключения заголовочных файлов заменяем одним ***avk_monitor.h***;
4) проводим компиляцию проекта и убеждаемся в отсутствии ошибок.

В результате указанных действий, данная компонента ПО примет вид показанный на листинге 2.5.

Листинг 2.5 — Компонента ПО виртуального терминала

```
/*
 ============================================================================
 Name        : avk_tty.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

#include "avk_monitor.h"

/**
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_tty_t *p){
      /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Включаем аппаратный курсор
       */
     printf("\033[?25h\n");

    if(p->changed)
      /**
```

```
       * Возвращаем состояние терминала.
       */
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
    /**
     * Освобождаем память, выделенную структуре.
     */
    free(p);
}

/**
 * Установка нового состояния терминала.
 */
avk_tty_t *
avk_setup_tty(void){
      /**
       * Выделяем память под структуру.
       */
      avk_tty_t *p = malloc(sizeof(avk_tty_t));
    if(p == NULL) return NULL;

    /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Выключаем аппаратный курсор.
       */
     printf("\033[?25l\n");
    /**
     * Сохраняем состояние терминала.
     */
    tcgetattr(STDIN_FILENO, &p->oldtty);
    tcgetattr(STDIN_FILENO, &p->newtty);
    /**
     * Переводим терминал в новое состояние:
     * отключаем канонический режим и эхо.
     */
    p->newtty.c_lflag &= ~(ICANON | ECHO | ECHOE );
    if(tcsetattr(STDIN_FILENO, TCSANOW, &p->newtty) < 0){
        /**
         * Если - проблема!!!
         * Возвращаем старое состояние терминала.
         */
      printf("Не могу установить терминал...\n");
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
        p->changed = 0;
    }else
      p->changed = 1;
    return p;
}

/**
 * Получить номер текущей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd)
{
```

```
      struct vt_stat vs;

      if(ioctl(fd, VT_GETSTATE, &vs) < 0)
            return 0xffff;

      return(vs.v_active);
}

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */

size_t
avk_read_tty(void *buff, size_t length){
    ssize_t L = 0;
    fd_set rfds;
    struct timeval tv;
    int retval;
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /**
     * Ожидаем не более 20ms: 50 раз/сек
     */
    tv.tv_sec = 0; tv.tv_usec = 20*1000;
    /**
     * Проверяем наличие прочитанных данных.
     */
    retval = select(1, &rfds, NULL, NULL, &tv);
    if (!retval) return 0;
    /**
     * Читаем данные, если они доступны.
     */
    if(FD_ISSET(0, &rfds)){
        if((L = read(0, buff, length)) < 1) return 0;
    }
    return (size_t)L;
}
```

Примечание — После переноса или создания отдельной компоненты ПО, следует всегда проводить компиляцию и отладку проекта. Это позволит избежать ошибок в перекрёстных ссылках.

Этап 3. Компонента ПО устройства мыши.

В отличии от проекта ***avk_tty***, проект ***avk_mouse*** не разделен на функции, поэтому следует:

1) в проекте ***avk_fb_monitor*** необходимо открыть файле ***avk_mouse.c*** и скопировать туда исходный текст проекта ***avk_mouse***;
2) как и на предыдущем этапе, необходимо заменить подключаемые файлы заголовков, а потом удалить функцию обработчика прерывания и ее установку в функции ***main()***;
3) далее, действовать как описано ниже.

Следует обратить внимание, что, в отличие от виртуального терминала, перед использованием устройства мыши, следует открыть, а после использования — закрыть это устройство.

Поэтому функцию ***main()***, в файле ***avk_mouse.c***, следует преобразовать в три функции: ***avk_mouse_open()***, ***avk_mouse_close()*** и ***avk_mouse_get()***, как это показано на листинге 2.6.

> Примечание — Все указанные функции должны принимать в качестве аргумента указатель на структуру типа ***avk_context_t***.

Это необходимо для централизованного доступа к общим данным проекта.

Листинг 2.6 — Компонента ПО устройства мыши

```
/*
 ============================================================================
 Name        : avk_mouse.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

#include "avk_monitor.h"

/**
 * Закрытие устройства мыши.
 */
void
avk_mouse_close(avk_context_t *ctx){
      close(ctx->fdc);
      ctx->fdc = -1;
}

/**
 * Открытие устройства мыши:
 *
 *  0 - устройство открыто;
 * -1 - ошибка открытия устройства.
 */
int
avk_mouse_open(avk_context_t *ctx){
      char *title =
                  "Проект avk_mouse_open:";
    /**
     * Открываем устройство мыши для чтения
     * и без блокировки.
     */
    if((ctx->fdc = open(MOUSEFILE, O_RDONLY |
            O_NONBLOCK)) == -1) {
        printf("%s\tОшибка открытия устройства мыши...\n",
                  title);
        exit(-1);
    }

    return 0;
}

/**
```

```
 * Чтение и установка абсолютных координат мыши.
 *
 * Возвращает значения:
 * 0  - данные прочитаны, обработаны и сохранены;
 * -1 - данные отсутствуют.
 */
int
avk_mouse_get(avk_context_t *ctx){
    /**
     * Структура, читаемая с устройства мыши.
     */
    struct input_event ie;
    /**
     * Относительные изменения координат мыши.
     */
    char x,y;

    /**
     * Указатель на читаемую структуру.
     */
    unsigned char *ptr = (unsigned char*)&ie;

      /**
       * Читаем структуру данных мыши.
       * Поскольку чтение осуществляется без блокировки,
       * то получение отрицательного значения говорит,
       * что данные отсутствуют.
       */
      if(read(ctx->fdc, &ie, sizeof(struct input_event)) < 0){
            return -1;
      }
    /**
     * Первый байт структуры - состяния кнопок.
     */
    unsigned char  button=ptr[0];
    ctx->bLeft   = button & 0x1;
    ctx->bMiddle = ( button & 0x4 ) > 0;
    ctx->bRight  = ( button & 0x2 ) > 0;

    /**
     * Относительные значения координат.
     */
    x=(char) ptr[1];
    y=(char) ptr[2];
    /**
     * Вычисление абсолютных координат мыши.
     */
    ctx->xc += x;
    ctx->yc -= y;

    return 0;
}
```

После отладки этой части ПО, следует перенести в файл ***avk_monitor.h*** описания функций ***avk_mouse_open()***, ***avk_mouse_open()*** и ***avk_mouse_get()***.

Этап 4. Компонента ПО устройства фреймбуфера.

Формирование компоненты ПО устройства фреймбуфера происходит аналогично формированию ПО устройства мыши:

1) в проекте создаётся файл ***avk_fb.c*** в который копируется содержимое проекта ***avk_fb***;
2) после преобразований исходного текста касающегося подключаемых заголовков файлов, а также удаления глобальных переменных и обработчика прерывания, следует удалить также код, связанный с формированием динамической части изображения, поскольку он не относится непосредственно к ПО устройства фреймбуфера;
3) далее, функция ***main()*** также преобразуется в следующие три функции: ***avk_fb_open()***, ***avk_fb_close()*** и ***avk_fb_paint()***, причём последняя функция необходима исключительно для восстановления фона экрана после переключений виртуальных терминалов.

Исходный код этой части проекта представлен на листинге 2.7.

После отладки этой части кода, описания функций также переносятся в файл заголовка ***avk_monitor.h***.

Листинг 2.7 — Компонента ПО устройства фреймбуфера

```
/*
 ============================================================================
 Name        : avk_fb.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

#include "avk_monitor.h"

/**
 * Закрытие устройства фреймбуфера.
 */
void
avk_fb_close(avk_context_t *ctx){
      /**
       * Завершаю работу.
       */
      printf("Завершаю работу...\n");
    munmap(ctx->screen, ctx->stride * ctx->h);
      close(ctx->fdf);
      ctx->fdf = -1;
}
/**
 * Открытие устройства фреймбуфера.
 */
int
avk_fb_open(avk_context_t *ctx){
      char * title =
                  "avk_fb_open():";
      /**
       * Открываем устройство фреймбуфера для
       * чтения и записи.
       */
      ctx->fdf = open(FBFILE, O_CLOEXEC | O_RDWR);
      if(ctx->fdf < 0){
            printf("%s\tНе могу открыть файл: %s\n",
                        title, FBFILE);
            return -1;
      }

      /**
```

```
 * Определяем параметры монитора.
 */
struct fb_fix_screeninfo fix;
if(ioctl (ctx->fdf, FBIOGET_FSCREENINFO, &fix) == -1){
      printf("%s\tНе могу получить информацию о буфере...\n",
                  title);
      close(ctx->fdf);
      return -1;
}
/**
 * Длина строки экрана в пикселях:
 * по 4 байта на пиксель.
 */
ctx->stride = fix.line_length;
/**
 * Читаю формат пикселя в битах.
 */
struct fb_var_screeninfo var;
var.bits_per_pixel = 0;
ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var);
/**
 * Если формат не 32 бита на пиксель, то
 * переустанавливаю это значение.
 */
if(var.bits_per_pixel != 32){
      var.bits_per_pixel = 32;
      ioctl (ctx->fdf, FBIOPUT_VSCREENINFO, &var);
}
/**
 * Снова (для контроля) читаю структуру.
 */
if(ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var) == -1){
      printf("%s\tНе могу получить информацию о буфере...\n",
                  title);
      close(ctx->fdf);
      return -1;
}
/**
 * Проверка формата пикселя.
 */
if(var.bits_per_pixel != 32){
      printf("%s\tНа такой экран - не записываю!!!\n",
                  title);
      close(ctx->fdf);
      return -1;
}
/**
 * С фреймбуфером можно работать как с последовательным
 * символьным файлом, но лучше - как с матрицей целых чисел,
 * поэтому проводим mmap() на память ЭВМ.
 */
ctx->w   = var.xres;                // Ширина фреймбуфера в пикселях.
ctx->h   = var.yres;                // Высота фреймбуфера в пикселях.
int nbuf =
           ctx->h*fix.line_length; // Длина фреймбуфера в байтах.
ctx->screen = (uint32_t *)mmap (0, nbuf, PROT_READ |
           PROT_WRITE, MAP_SHARED,
           ctx->fdf, 0);
if(ctx->screen <= 0){
      printf("%s\tНе могу сделать mmap() на фреймбуфер!!!",
```

```
                        title);
                close(ctx->fdf);
                return -1;
        }

        return EXIT_SUCCESS;
}
/**
 * Заполнение окна фреймбуфера цветом.
 */
int
avk_fb_paint(avk_context_t *ctx){
        /**
         *  Заполняем фоновым цветом весь экран.
         */
        for(int i=0; i<ctx->stride * ctx->h; ++i)
                ctx->screen[i] = ctx->color32;
        return 0;
}
```

Этап 5. Компонента ПО динамического рисунка.

Хотя компонента ПО динамического рисунка — достаточно примитивна, выделим ее отдельной частью, имитируя возможное большое приложение:

1) в нашем проекте открываем файл ***avk_dynamic.c***;
2) затем, реализуем функцию рисования с именем ***avk_dynamic_set()***, как показано на листинге 2.8.

Листинг 2.8 — Компонента ПО динамического рисунка

```
/*
 * avk_dynamic.c
 *
 *  Created on: 16 авг. 2017 г.
 *      Author: upk
 */

#include "avk_monitor.h"

/**
 * Рисование динамической части
 * изображения в фреймбуфер.
 */
void
avk_dynamic_set(avk_context_t *ctx){
        /**
         * Параметры динамической части рисунка:
         * 5 позиций квадрата 40x40 пикселей каждый.
         *
        uint32_t bg = 0x007f7f7f;     // Цвет фона.
        uint32_t fg = 0x00007f00;     // Цвет заполнения.
        uint32_t w0 = 40;             // Ширина прямоугольника.
        uint32_t h0 = 40;             // Высота прямоугольника.
        uint32_t n0 = 5;              // Число  прямоугольников.
        uint32_t k0 = 0;              // Текущий прямоугольник.
        uint32_t x0 = (nWidth  - 5*40)/2;
        uint32_t y0 = nHeight/2;
*/
        uint32_t t0;      // Текущий цвет = 0x007f7f7f.
        uint32_t m0;      // Ширина запонения в пикселях.
```

```
        /**
         * Отрисовка динамической части изображения.
         */
        if(ctx->k0 == 0){
              t0 = ctx->bg;
              m0 = ctx->n0*ctx->w0;
        }else{
              t0 = ctx->fg;
              m0 = ctx->k0*ctx->w0;
        }
        for(int i=0; i<ctx->h0; ++i){
              for(int j=0; j<m0; ++j)
                    ctx->screen[(ctx->y0 + i)*ctx->w + ctx->x0 + j] = t0;
        }
        ctx->k0++;
        if(ctx->k0 > ctx->n0)
              ctx->k0 = 0;
}
```

2.2.4 Реализация проекта avk_fb_monitor

Выполнив организацию ПО отдельных компонент нашего проекта, переходим к реализации главной части ПО, проводящей синхронизацию (композицию) всех функциональных элементов системы.

Для этого необходимо:

1) *перейти* к ранее созданному файлу проекта: ***avk_fb_monitor.c***;
2) *включить* в него заголовочный файл ***avk_monitor.h***;
3) *реализовать* в функции ***main()*** целевой алгоритм, обеспечивающий решение поставленной задачи.

Общее типовое решение, при реализации функции ***main()***, состоит в разделении ее кода на три части:

1) первая часть, состоит в *инициализации* всех глобальных переменных и *открытия* используемых устройств;
2) вторая часть, обычно в цикле, *собственно и реализует* алгоритм синхронизации (композитинга) решаемой задачи;
3) третья часть, обеспечивает *согласованное закрытие* всех устройств и *нормальное завершение* работы программы.

Если функционал первой и третьей частей уже реализован нами, как по отдельности в предыдущей лабораторной работе, так и в данной работе, посредством выделения отдельных частей ПО, то алгоритм второй части требует отдельного проектирования.

Прежде всего, необходимо определить: «*Какой набор системных средств будет использован для реализации алгоритма синхронизации?*».

Собственно, ответы на этот вопрос и являются предметом нашей дисциплины.

В пределах возможностей отдельной рабочей станции, ответы могут быть следующие:

1) синхронизация *на уровне прикладного программирования*, когда используется одна (главная) нить отдельного процесса, требующего циклический мониторинг состояний отдельных асинхронно работающих компонент системы, с последующей реакцией на изменения этих состояний;
2) синхронизация *на уровне множества нитей*, когда согласование обеспечивается средствами блокировки, семафорами, мютексами и другими аналогичными средствами;

3) синхронизация *на уровне мультипроцессинга*, когда используются средства явного распределения вычислительных ресурсов системы;
4) синхронизация *на уровне передачи сообщений*, когда реализуются распределённые системы.

Каждая из перечисленных возможностей требует применения различных технологий программирования и навыков их использования, а также привлечения различных системных средств ПО ЭВМ.

> Примечание — В пределах заданной учебной цели, вполне достаточным является уровень прикладного программирования, на котором и реализован наш проект.

Результат такой реализации представлен на листинге 2.9.

Листинг 2.9 — Композитинг на уровне прикладного программирования

```
/*
 ============================================================================
 Name        : avk_fb_monitor.c
 Author      : Reznik V.G., 15.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

#include "avk_monitor.h"

int main(void) {
	char *title =
			"Проект avk_fb_monitor:";
	printf("%s\tЗакрашивание экрана устройством мыши\n",
			title);
	printf("\nКомбинация клавишь Ctrl-X - завершение работы программы!\n");
	printf("Для продолжения - нажми Enter...\n");
	getchar();

	/** ***************************************
	 * Первая часть ПО.
	 * Инициализация данных и открытие устройств.
	 *****************************************/
	/**
	 * Инициализация общей контекстной структуры проекта.
	 */
	avk_context_t ctx;
	/**
	 * Данные виртуального терминала.
	 */
	ctx.nvt = avk_get_current_vc(0);    // Номер виртуального  терминала.
	ctx.tty_status = avk_setup_tty();   // Структура состояния терминала.
	if(ctx.tty_status == NULL || ctx.tty_status->changed == 0){
		printf("%s\tНе могу инициализировать виртуальный терминал\n",
				title);
		usleep(2000*1000);
		return -1;
	}

	/**
	 * Данные фреймбуфера.
	 */
```

```
if(avk_fb_open(&ctx) < 0){
      printf("%s\tНе могу открыть фреймбуфер\n",
                  title);
      usleep(2000*1000);
      avk_restore_tty(ctx.tty_status);
      return -1;
}
ctx.x = 0;
ctx.y = 0;
ctx.color32 = 0x00ff;

/**
 * Данные курсора.
 */
if(avk_mouse_open(&ctx) < 0){
      printf("%s\tНе могу открыть устройство мыши\n",
                  title);
      usleep(2000*1000);
      avk_fb_close(&ctx);
      avk_restore_tty(ctx.tty_status);
      return -1;
}
ctx.xc = ctx.w/2;       // Курсор - в центр экрана.
ctx.yc = ctx.h/2;
ctx.wc = 20;            // Размеры курсора.
ctx.hc = 40;
ctx.colorC = 0x007f7f00; // Цвет курсора.

/**
 * Данные динамической части рисунка.
 */
ctx.bg = 0x007f7f7f;    // Цвет фона = 0x007f7f7f.
ctx.fg = 0x00007f00;    // Цвет заполнения = 0x00007f00.
ctx.x0 = ctx.w/2;       // Координаты начала окна.
ctx.y0 = ctx.h/2;
ctx.w0 = 40;      // Ширина прямоугольника = 40.
ctx.h0 = 40;      // Высота прямоугольника = 40.
ctx.n0 = 5;       // Число  прямоугольников = 5.
ctx.k0 = 0;       // Текущее состояние.

/**
 * Рабочие переменные.
 */
int xt, yt;       // Координаты мыши.
int dirty = 1;    // Нужно полностью перерисовать экран.

/** ****************************************
 * Вторая часть ПО.
 * Цикл композитинга, реализующего
 * программный мониторинг состояний устройств.
 ******************************************/
while(1){

      usleep(20*1000);  // Период засыпания 20 ms.
      /**
       * Проверяю, является ли терминал текущим.
       */
      if(ctx.nvt != avk_get_current_vc(0)){
            dirty = 1;
         /**
```

```
            * Запускаем ioctl() для ожидания активности терминала!!!.
            */
        ioctl(0, VT_WAITACTIVE, ctx.nvt);
        printf("\nТерминал №%i снова стал активным...\n",
                    ctx.nvt);
        }

        /**
         * Проверяю, нужно ли завершить работу.
         */
        if(avk_read_tty(ctx.tbuf, 10) > 0){
              if((unsigned int)ctx.tbuf[0] == 24){
                    /**
                     * Нажата комбинация Ctrl-x, выходим из цикла.
                     */
                    break;
              }
        }

        /**
         * Проверяю, нужно ли перерисовывать экран.
         */
        if(dirty){
              avk_fb_paint(&ctx);     // Перерисовываем.
              ctx.k0 = 0;             // Динамический рисунок переводим
                                      // в начальное состояние.
              dirty  = 0;
        }

        /**
         * Выводим динамический рисунок.
         */
        avk_dynamic_set(&ctx);

        /**
         * Проверяю, нужно ли перерисовывать курсор.
         */
        xt = ctx.xc;            // Сохраняем координаты.
        yt = ctx.yc;
        avk_mouse_get(&ctx);    // Получаем новые.
        /**
         * Курсор должен быть полностью в пределах экрана.
         */
        if(ctx.xc < 0)    ctx.xc = 0;
        if(ctx.yc < 0)    ctx.yc = 0;
        if((ctx.xc+ctx.wc) >= ctx.w)  ctx.xc = ctx.w - ctx.wc;
        if((ctx.yc+ctx.hc) >= ctx.h)  ctx.yc = ctx.h - ctx.hc;
        /**
         * Если позиция курсора изменилась, то
         * рисуем его в новой позиции.
         */
        if(xt != ctx.xc || yt != ctx.yc){
              for(int i=0; i<ctx.hc; i++)
                    for(int j=0; j<ctx.wc; j++)
                          ctx.screen[(ctx.yc+i)*ctx.w + ctx.xc + j] =
                                      ctx.colorC;
        }
}
/** ****************************
 * Третья часть ПО.
```

```
         * Закрытие устройств и завершение
         * работы программы.
         *******************************/
        printf("%s\tЗавершаю работу\n",
                        title);
        usleep(2000*1000);
        avk_mouse_close(&ctx);
        avk_fb_close(&ctx);
        avk_restore_tty(ctx.tty_status);

        return EXIT_SUCCESS;
}
```

Студенту следует исследовать и описать недостатки использованного подхода в данной реализации программы.

2.3 Лабораторная работа №6. Асинхронный композитинг на уровне нитей

Учебная цель данной работы — п рактическое освоение синхронизации параллельно работающих компонент компьютера ***на уровне технологии нитей***.

Методологически данная лабораторная работа построена на базе двух предыдущих работ, что достигается двумя способами:

1) модификацией постановки задачи посредством усложнения требований к ее прикладной целевой функции;
2) применением более совершенных и современных методов ее решения, которые более полно демонстрируют технологии вычислительных комплексов.

По тематике технологического решения, данная работа названа как «***Асинхронный композитинг изображений на уровне нитей***».

2.3.1 Критика синхронизации на уровне прикладного программирования

Хотя синхронизация на уровне прикладного программирования вполне адекватна задаче раскрашивания экрана, реализованной в предыдущей работе, нетрудно выделить ряд недостатков, присущих этой технологии.

Отметим два основных недостатка: *плохая масштабируемость* и *плохая согласованность*.

Плохая масштабируемость обусловлена тем, что в основном цикле алгоритма реализации необходимо учитывать как порядок взаимодействия с каждой асинхронной компонентой, так и программировать реакции на результат такого взаимодействия:

1) при увеличении числа взаимодействующих компонент, резко увеличивается число взаимосвязей, которые необходимо учитывать в реализуемом алгоритме.
2) при добавлении новой компоненты или изменении требований к одной из них, может возникнуть необходимость полной модификации самого алгоритма реализации.

Плохая согласованность обусловлена тем, что само взаимодействие и реакция на него осуществляются *один раз за цикл* с последующим освобождением вычислительного ресурса на некоторый тайм-аут:

1) для некоторых компонент, выделенный тайм-аут может оказаться слишком маленьким, а для некоторых — слишком большим, но требования качества исполнения заставляют использовать наименьший тайм-аут.
2) при изменении числа взаимодействующих компонент изменяется и период времени ожидания для каждого из них, что требует усложнения алгоритма реализации для обеспечения нужного качества.

2.3.2 Асинхронный композитинг изображений на уровне нитей

Как уже было отмечено в предыдущей лабораторной работе, классическая семантика термина композитинг связана с программными средствами, которые манипулируют размещением и комбинированием наложений различных изображений. Такие программные средства называются *композиторами*.

Основная проблематика композиторов — манипулирование большими объёмами данных, что характерно для графического способа представления информации, при условии обеспечения нужного качества.

Другой аспект этой проблематики — необходимость синхронизации работы:

1) входных устройств, к которым относятся клавиатура, мышка и аналогичные устройства ввода информации;
2) выходных устройств, которым относятся графическая карта и монитор (дисплей) компьютера.

Примечание — Задача реализации даже простейшего композитора, является сферой применения самых современных технологических приёмов системного программирования.

Учебная задача данной работы — модификация программного обеспечения предыдущей лабораторной работы с целью *отображения курсора устройства мыши*.

Для успешного решения задачи учтем опыт уже существующих разработок композиторов:

1) основным источником обработки данных являются прямоугольные окна, которые полностью отрисовываются отдельными приложениями;
2) для отрисовки окон используется специальный буфер, в котором и производится композитинг;
3) заполнение всего экрана осуществляется только при восстановлении изображения при переключении виртуальных терминалов;
4) обычно на дисплей отрисовываются только те части окон, которые были изменены за некоторый (системный) интервал времени;
5) для отрисовки курсора используются разные подходы: некоторые композиторы отрисовывают курсор прямо на дисплей, а некоторые — предварительно записав в его буфер, поверх окон;
6) многие приложения сами отрисовывают курсор, в пределах своих окон.

Излишне напоминать, что реальные композиторы являются достаточно сложными программными системами, которые:

1) отслеживают появление новых окон и удаление уже существующих;
2) показ (show) и прятание (hide) окон;
3) размещение фонового окна и обеспечение иерархии других (нормальных) окон;
4) обеспечение режима работы с диалоговыми окнами, меню и окнами системных сообщений;
5) реализацию различных эффектов с окнами: перемещение, изменение размера, fading (динамические эффекты) и другие возможности.

Чтобы непосредственно сосредоточиться на теме нашего обучения, уточним общие черты нашего проекта, выделив только отличия от описания задачи, приведённой в предыдущей лабораторной работе:

1) для композиции будет использоваться ***единственный общий буфер*** размер которого полностью совпадает с размером фреймбуфера; одновременно будем считать, что этот буфер композитора выполняет роль фонового (родительского) окна, которое обычно содержит некоторый фон или изображение рисунка;
2) устройство виртуального терминала не имеет изображения для композиции, но ***реализуется в виде отдельной нити***, отслеживая команду завершения работы программы;
3) устройство мыши имеет ***собственный рисунок***, который используется композитором и ***отдельную нить***, которая полностью вычисляет абсолютные координаты курсора в пределах экрана монитора;
4) динамический рисунок полностью реализуется ***в своём буфере отдельной нитью***; композитор перерисовывает в свой буфер только изменённую часть рисунка, а затем выводит ее в окно фреймбуфера;

5) сам композитор ***реализуется главной нитью*** (функцией ***main()***), в которой выделены части инициализации, рабочего цикла и завершающая часть работы программы проекта.

Ожидаемые преимущества рассматриваемого решения:

1) *программа полностью не блокируется*, при переключении на другой виртуальный терминал; приостанавливается только вывод на экран монитора;
2) *повышается самостоятельность* и *автономность* каждой отдельной компоненты, поскольку теперь их работа не будет зависеть от циклов запуска их композитором;
3) ожидается *повышение масштабируемости* проекта, за счёт переноса функциональности из алгоритма композитинга в алгоритмы компонент;
4) ожидается *повышение согласованности* проекта, за счёт предоставления возможности компонентам работать в нужном им темпе.

Примечание — Обсуждение деталей синхронизации перенесём в описательную часть самого программного обеспечения проекта.

Далее кратко рассмотрим инструментальную часть реализации проекта. К таким средствам инструментальной части относятся:

1) нити (*threads*);
2) мютексы (*mutex*) — упрощённые средства синхронизации;
3) графические средства библиотеки ***cairo***.

Инструментальные средства нитей.

Инструментальные средства нитей (***threads***) являются стандартными для всех ОС UNIX/Linux, в редакции стандарта POSIX. В качестве справки, на листинге 2.10 приведён перечень базовых функций, необходимых для реализации алгоритмов проекта.

Листинг 2.10 — Базовые функции для работы с нитями

```
/**
 * Заголовочный файл нитей.
 */
#include <pthread.h>

/**
 * Устанавливает по умолчанию атрибуты будущей нити.
 */
int pthread_attr_init(pthread_attr_t *attr);

/**
 * Открывает нить с заданными: атрибутом, функцией исполнения
 * и аргументом, передаваемым в функцию исполнения нити.
 */
int pthread_create(pthread_t * thread, pthread_attr_t *attr,
            void *(*start_routine)(void *), void *arg);

/**
 * Функция исполнения нити имеет общий вид:
 */
void *(*start_routine)(void *);

/**
 * Завершает работу текущей нити.
 * Работает аналогично функции exit() для процессов.
```

```
 * Аргумент: код возврата - указатель на внешнюю переменную.
 */
void pthread_exit(void * ret);

/**
 * Предлагает нити с заданным номером завершить работу.
 */
int pthread_cancel(pthread_t thread);

/**
 * Нить проверяет, было ли ей предложение закончить работу.
 * Если такое предложение было, то нить заканчивает работу
 * (следующий оператор уже не выполняется).
 */
void pthread_testcancel(void);

/**
 * Вызывается из кода, ожидающего завершение работы нити.
 * Работает аналогично функции wait() для процессов.
 */
int pthread_join(pthread_t th, void** thread_return);
```

Фактически, **нити** — стандарт распараллеливания алгоритмов задач, которые одинаково работают как в однопроцессорных, так и в многопроцессорных вычислительных комплексах. Само распараллеливание осуществляется ядром ОС и не является прозрачным для пользователя, применяющего эти средства.

Синхронизация нитей средствами мютексов.

Мютексы являются средствами синхронизации параллельно работающих нитей, наподобие семафоров.

Мютексы (***mutual exclusions***) — *взаимные исключения* или *исключающие семафоры*, обеспечивающие блокировку доступа к некоторым общим ресурсам.

Как правило, для реализации большинства задач достаточно небольшого числа функций мютексов, приведённых на листинге 2.11.

Листинг 2.11 — Базовые функции для работы с мютексами

```
/**
 * Заголовочный файл мютекса.
 */
#include <pthread.h>

/**
 * Инициализация мютекса.
 * Если второй аргумент функции равен NULL,
 * то инициализируется быстрый мютекс,
 * который мы и будем использовать.
 */
int pthread_mutex_init(pthread_mutex_t* mutex,
            const pthread_mutexattr_t *mutexattr);

/**
 * Запрос на блокировку ресурса, защищаемого мютексом:
 * - первый вызов функции - предоставляет ресурс;
 * - второй и последующие вызовы - блокируют самого вызывающего.
 */
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

```
/**
 * Разблокирование ранее заблокированного мютекса.
 */
int pthread_mutex_unlock(pthread_mutex_t* mutex);

/**
 * Перевод мютекса в неинициализированное состояние.
 */
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Необходимость синхронизации возникает тогда, когда композитор берет изображение компонента, для композитинга его в своём буфере. Если изображение находится в процессе создания, композитор может взять ошибочные данные. Для исключения подобных ситуаций, *защитим буфер композитора мютексом*, требуя, чтобы все участники генерации новых данных блокировали этот буфер до окончания своей работы и снимали блокировку, когда такая работа завершена.

Примечание — В нашей задаче, источником проблем является генератор динамического изображения.

Что касается курсора мыши, то здесь проблем — нет, поскольку само изображение курсора не меняется, а меняются его координаты. Координаты изменяются двумя записями целых чисел и не требует блокировки буфера композитора.

Графические средства библиотеки cairo.

В предыдущей лабораторной работе, выводимые на экран изображения формировались набором целых чисел, что приемлемо только для очень примитивных решений задачи. В текущей лабораторной работе, формирование изображений будет проводиться средствами библиотеки ***cairo***.

Cairo — библиотека, которая ***с 2003 года*** применяется во многих операционных системах, включая MS Window, для отрисовки двумерной векторной графики:

1) к достоинствам библиотеки следует также отнести удобную работу с тестом, используя национальные языки.
2) желающие изучить этот инструментарий могут воспользоваться Интернет-учебником - http://www.opennet.ru/docs/RUS/tutorial_cairo/ или обратиться к официальной документации - https://www.cairographics.org/manual/.

Основными элементами манипулирования библиотеки ***cairo*** являются объекты типа:

- ***cairo_surface_t*** — прямоугольные области, определённые на массивах данных;
- ***cairo_t*** — контекст для конкретной прямоугольной области, который можно рассматривать как дескриптор при работе с файлами.

Дополнительные ограничения задачи:

1. На каждой созданной прямоугольной области (поверхности) ***surface*** можно рисовать замкнутые кривые, которые затем можно закрашивать некоторым цветом и окантовывать линиями.
2. Выбираемые цвета задаются из набора красный, зелёный и синий, а также можно задавать прозрачность объекта закраски.
3. Полный набор манипуляций с поверхностями ***surface*** — достаточно обширный и включает в себя не только возможность работы со стандартными форматами файлов изображений, но и с устройствами, например, такими как устройство фреймбуфера.

Учитывая, что решаемая нами задача не содержит сложных изображений, правила работы с операторами и функциями библиотеки ***cairo*** должны быть понятны даже неподготовленным специалистам.

2.3.3 Модификация компонент взаимодействующих устройств. Проект avk_fb_compositor

В соответствии с требованиями поставленной задачи, откроем в среде разработки EclipseC проект ***avk_fb_compositor*** и приступим к выделению отдельных компонент программного обеспечения для решения поставленной задачи.

Учитывая, что постановка нашей задачи, почти во многих аспектах, совпадает с постановкой задачи предыдущей лабораторной работы, то в данной работе будем использовать исходный текст ее проекта.

Для этого:

- из проекта ***avk_fb_monitor***, без изменений, переносим в наш проект содержимое файлов ***avk_dynamic.c***, ***avk_fb.c***, ***avk_mouse.c*** и ***avk_tty.c***;
- содержимое файла ***avk_monitor.h*** (без макросов заголовка) переносим в заголовочный файл ***avk_compositor.h*** нашего проекта;
- содержимое файла ***avk_fb_monitor.c*** полностью переносим в созданный ранее файл ***avk_fb_compositor.c*** нашего проекта.

Примечание — Нельзя переносить содержимое файлов копированием из одной директории в другую. Необходимо, в новом проекте создать нужный файл и переносить содержимое копированием нужной части текста, воспользовавшись средствами редактора системы разработки. Кроме того, после переноса содержимого файлов, необходимо поменять название подключаемого файла заголовка. Обязательно, подключить к проекту библиотеку ***pthread***.

Учебная цель такого подхода к реализации проекта — детально разобраться с изменениями ПО, которые порождаются изменением уровня системного программирования.

Примечание — Для лучшего разделения самого процесса реализации задачи, будем рассуждения и выполняемые действия также выделять отдельными этапами.

Этап 1. Изменение структуры контекста задачи.

Добавляется новые заголовочные файлы <***pthread.h***> и <***cairo/cairo.h***>, обеспечивающие нам доступ к функциям нитей, мютексов и библиотеки ***cairo***.

Список объявлений функций остаётся прежним, поскольку добавляемый или изменяемый функционал будет реализован в рамках отдельных компонент.

Основные изменения касаются объявления структуры типа ***avk_context_t***, в которую добавляются:

1) указатели на типы ***cairo_t*** и ***cairo_surface_t***, обеспечивающие работу с поверхностями изображений;
2) данные для работы с нитями, мютексами и другие элементы синхронизации.

Полный исходный текст заголовочного файла ***avk_compositor.h*** приведён на листинге 2.12, где:

- добавляемые элементы файла отмечены комментариями;
- удалённые элементы — просто закомментированы.

Листинг 2.12 — Заголовочный файл avk_compositor.h контекста задачи

```
/*
 * avk_compositor.h
 *
```

```
 *  Created on: 18 авг. 2017 г.
 *      Author: upk
 */

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

#include <sys/vt.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <string.h>

#include <fcntl.h>
#include <linux/input.h>
#include <signal.h>
#include <errno.h>

#include <stdint-gcc.h>
#include <linux/fb.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

/**
 * Добавляем заголовочные файлы для работы с
 * нитями, мютексами и библиотекой cairo.
 */
#include <pthread.h>
#include <cairo/cairo.h>

#ifndef AVK_COMPOSITOR_H_
#define AVK_COMPOSITOR_H_

#define MOUSEFILE "/dev/input/mouse0"
#define FBFILE    "/dev/fb0"

/**
 * Объявление структуры состояния терминала.
 */
typedef struct _avk_tty {
      struct termios oldtty;  // Старое состояние терминала.
      struct termios newtty;  // Новое  состояние терминала.
      int         changed;    // Статус состояния терминала.

} avk_tty_t;

/**
 * Обявление общей контекстной структуры проекта.
 */
typedef struct _avk_context {
      /**
       * Данные виртуального терминала.
       * Терминал отслеживает необходимость завершения работы
       * и возможность вывода в фреймбуфер.
       */
//    unsigned short    nvt;        // Номер виртуального  терминала.
      avk_tty_t         *tty_status;// Структура состояния терминала.
//    char              tbuf[10];   // Буфер терминала.
```

```
      int               if_exit;    // Новое: 0 - продолжаем работу; 1 - выход.
      int               no_fb;      // Новое: 0 - можно, 1 - нельзя выводить в
fb.
      pthread_attr_t  attr_vt;      // Новое: Атрибут  нити  терминала.
      pthread_t         thr_vt;     // Новое: Идентификатор  нити терминала.
      /**
       * Данные фреймбуфера.
       * Добавляем мютекс синхронизации и указатели cairo
       * окна композитора и фреймбуфера.
       * Композитор исполняется главной нитью!!!
       */
      int fdf;                // Дескриптор фреймбуфера.
      int x, y, w, h;         // Размеры окна фреймбуфера.
      int stride;             // Длина строки окна в байтах.
      uint32_t color32;       // Цвет заполнения всего экрана  = 0x00ff
      unsigned char * screen;       // Изменение: выровненный по байту указатель
                                    // на данные фреймбуфера.
      unsigned char * data;         // Новое: Указатель на  буфер  композитора.
      pthread_mutex_t mutex_comp;   // Новое: Мютекс,   для работы композитора.
      cairo_surface_t *surf;        // Новое: Указатель на  окно   фреймбуфера.
      cairo_t         *crf;         // Новое: Контекст  окна фреймбуфера.
      cairo_surface_t *sur;         // Новое: Указатель на  окно композитора.
      cairo_t         *cr;          // Новое: Контекст  окна композитора.
      /**
       * Данные курсора.
       * Курсор: треугольный, большой, черно-белый и полупрозрачный.
       */
      int fdc;                // Дескриптор курсора.
      int xc, yc;             // Координаты курсора.
      int wc;                 // Ширина курсора = 100.
      int hc;                 // Высота курсора = 200.
      int bLeft, bMiddle, bRight;   // Состояние кнопок мыши.
//    uint32_t colorC;        // Цвет рисования курсором = 0x007f7f00.
      unsigned char   *datac;       // Новое: Указатель на  буфер  курсора.
      cairo_surface_t *surc;        // Новое: Указатель на  окно   курсора.
      cairo_t         *crc;         // Новое: Контекст  окна  курсора.
      pthread_attr_t  attr_c;       // Новое: Атрибут   нити  курсора.
      pthread_t       thr_c;        // Новое: Идентификатор   нити курсора.
      /**
       * Данные динамической части рисунка.
       */
//    uint32_t bg;                  // Цвет фона = 0x007f7f7f.
//    uint32_t fg;                  // Цвет заполнения = 0x00007f00.
      uint32_t x0, y0;  //   Изменено:   Координаты   начала   окна   вывода   -
относительные.
      uint32_t w0;      // Изменено: Ширина окна вывода - переменная.
      uint32_t h0;                  // Высота прямоугольника = 40.
      uint32_t n0;                  // Число  прямоугольников = 5.
      uint32_t k0;      // Изменено: Текущее состояние.
      unsigned char * datad;  // Новое: Указатель на  буфер  курсора.
      cairo_surface_t *surd;  // Новое: Указатель на  окно   курсора.
      cairo_t         *crd;   // Новое: Контекст  окна  курсора.
      pthread_attr_t  attr_d; // Новое: Атрибут   нити  курсора.
      pthread_t       thr_d;  // Новое: Идентификатор   нити курсора.

} avk_context_t;

/** ********************************************************
 * Объявления функций.
 ********************************************************/
```

```
/**
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_context_t *ctx);

/**
 * Установка нового состояния терминала.
 * Новое: аргументом является глобальная структура.
 */
avk_tty_t *
avk_setup_tty(avk_context_t *ctx);

/**
 * Получить номер текущей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd);

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */
size_t
avk_read_tty(void *buff, size_t length);

/**
 * Закрытие устройства мыши.
 */
void
avk_mouse_close(avk_context_t *ctx);

/**
 * Открытие устройства мыши:
 *
 *  0 - устройство открыто;
 * -1 - ошибка открытия устройства.
 */
int
avk_mouse_open(avk_context_t *ctx);

/**
 * Чтение и установка абсолютных координат мыши.
 *
 * Возвращает значения:
 * 0  - данные прочитаны, обработаны и сохранены;
 * -1 - данные отсутствуют.
 */
int
avk_mouse_get(avk_context_t *ctx);

/**
 * Закрытие устройства фреймбуфера.
```

```
 */
void
avk_fb_close(void *arg);

/**
 * Открытие устройства фреймбуфера.
 */
int
avk_fb_open(avk_context_t *ctx);

/**
 * Заполнение окна фреймбуфера цветом.
 */
int
avk_fb_paint(avk_context_t *ctx);

/**
 * Рисование динамической части
 * изображения в фреймбуфер.
 */
void *
avk_dynamic_set(void *arg);

#endif /* AVK_COMPOSITOR_H_ */
```

Этап 2. Нить виртуального терминала.

Приступая к преобразованию первой компоненты нашего проекта, следует уточнить основную идею этих преобразований: формирование максимально независимой компоненты ПО, реализуемой отдельной нитью приложения.

Действительно, в нашей задаче, все многообразие возможностей виртуального терминала сводится к анализу входной информации, поступающей с устройства клавиатуры, посредством нажатия ее клавиш. Полезным, для композитора и всего приложения, результатом такой деятельности является обнаружение и фиксация двух событий (состояний):

- необходимость завершения работы программы, что фиксируется единичным значением бинарной переменной ***if_exit***;
- запрет записи в устройство фреймбуфера, что фиксируется единичным значением бинарной переменной ***no_fb***.

Таким образом общая схема преобразований состоит в следующем:

1) создаётся новая функция ***avk_tty_thread()***, реализующая алгоритм нити для определения и фиксации, в глобальной структуре, переменных ***if_exit*** и ***no_fb***;
2) в функцию ***avk_setup_tty()***, в качестве аргумента, передается адрес глобальной структуры данных и добавляются операторы запуска нити компонента;
3) в функции ***avk_restore_tty()***, аргумент заменяется на адрес глобальной структуры данных и добавляются операторы завершения работы нити компонента.

На листинге 2.13 приведён результат проведённых преобразований в ***avk_tty.c***.

Листинг 2.13 — Новая компонента ПО виртуального терминала

```
/*
 * avk_tty.c
 *
 *  Created on: 18 авг. 2017 г.
 *      Author: upk
 */
```

```
#include "avk_compositor.h"

/** ****************************************
 * Новая функция, реализующая нить компоненты.
 * Вызывается из функции avk_setup_tty().
 */
void *
avk_tty_thread(void *arg){
      /**
       * Номер терминала обслуживается нитью vt.
       */
      avk_context_t   *ctxp = (avk_context_t *)arg;
      char              tbuf[10];               // Буфер терминала.
      unsigned short    nvt =
                  avk_get_current_vc(0);        // Запоминаем номер vt.
      /**
       * Цикл обслуживания.
       */
      while(1){
            if(nvt != avk_get_current_vc(0)){// Терминал не является текущим.
                  ctxp->no_fb = 1;  // Выводить в фреймбуфер нельзя.
                  usleep(300*1000); // Засыпаем на 0.3 секунды.
                  continue;
            }
            ctxp->no_fb = 0;  // Выводить в фреймбуфер можно.
            /**
             * Проверяю, нужно ли завершить работу.
             */
            if(avk_read_tty(&tbuf, 10) > 0){
                  if((unsigned int)tbuf[0] == 24){
                        /**
                         * Нажата комбинация Ctrl-X, выходим из цикла.
                         */
                        ctxp->if_exit = 1; // Все выходят!!!
                        break;
                  }
                  /**
                   * Здесь возможна реализация алгоритмов реакции
                   * и на другие комбинации клавиш.
                   */
            }
            /**
             * Проверяем внешнее предложение завершить работу нити.
             */
            pthread_testcancel();   // Возможен выход здесь.
            usleep(20*1000);        // Период обслуживания 20 ms.
      }
      /**
       * Полный выход из нити.
       */
      pthread_exit(0);  // Завершаем работу нити.
}

/**
 * Изменено:
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
```

```
avk_restore_tty(avk_context_t *ctx){
      /**
       * Новое:
       * Если нить стартовала, то ожидаем ее завершение.
       */
      if(ctx->thr_vt > 0){
            pthread_cancel(ctx->thr_vt);
            pthread_join(ctx->thr_vt, NULL);
      }

      avk_tty_t *p =
                  ctx->tty_status;
      /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Включаем аппаратный курсор
       */
     printf("\033[?25h\n");

    if(p->changed)
      /**
       * Возвращаем состояние терминала.
       */
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
    /**
     * Освобождаем память, выделенную структуре.
     */
    free(p);
}

/**
 * Изменено:
 * Установка нового состояния терминала.
 * Новое:
 * - аргументом является глобальная структура.
 * Результат:
 * - если инициализация не успешна, то выставляем if_exit = 1;
 * - если инициализация успешна, то выставляем if_exit = 0
 * и запускаем нить обслуживания.
 */
avk_tty_t *
avk_setup_tty(avk_context_t *ctx){
      /**
       * Для последующего контроля дескриптора нити.
       */
      ctx->thr_vt = 0;
      /**
       * Выделяем память под структуру.
       */
      avk_tty_t *p = malloc(sizeof(avk_tty_t));
    if(p == NULL) return NULL;

    /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
```

```
      /**
       * Выключаем аппаратный курсор.
       */
     printf("\033[?25l\n");
    /**
     * Сохраняем состояние терминала.
     */
    tcgetattr(STDIN_FILENO, &p->oldtty);
    tcgetattr(STDIN_FILENO, &p->newtty);
    /**
     * Переводим терминал в новое состояние:
     * отключаем канонический режим и эхо.
     */
    p->newtty.c_lflag &= ~(ICANON | ECHO | ECHOE );
    if(tcsetattr(STDIN_FILENO, TCSANOW, &p->newtty) < 0){
        /**
         * Если - проблема!!!
         * Возвращаем старое состояние терминала.
         */
      printf("Не могу установить терминал...\n");
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
        p->changed   = 0;
        ctx->if_exit = 1;     // Всем на выход!!!
    }else{
      p->changed   = 1;
        ctx->if_exit = 0;
        /**
         * Далее, запускаем нить.
         *
       * Получаем дефолтные значения атрибутов нити.
       */
      pthread_attr_init(ctx->attr_vt);
      /**
       * Стартуем отдельную нить композитора
       */
      if (pthread_create(ctx->thr_vt, ctx->attr_vt,
                  avk_tty_thread, (void*)ctx) != 0) {
            printf("Не могу запустить нить композитора...\n");
            p->changed   = 0;
            ctx->if_exit = 1; // Всем на выход!!!
      }
    }

    return p;
}

/**
 * Получить номер текущей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd)
{

      struct vt_stat vs;

      if(ioctl(fd, VT_GETSTATE, &vs) < 0)
            return 0xffff;

      return(vs.v_active);
```

```
}

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */

size_t
avk_read_tty(void *buff, size_t length){
    ssize_t L = 0;
    fd_set rfds;
    struct timeval tv;
    int retval;
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /**
     * Ожидаем не более 20ms: 50 раз/сек
     */
    tv.tv_sec = 0; tv.tv_usec = 20*1000;
    /**
     * Проверяем наличие прочитанных данных.
     */
    retval = select(1, &rfds, NULL, NULL, &tv);
    if (!retval) return 0;
    /**
     * Читаем данные, если они доступны.
     */
    if(FD_ISSET(0, &rfds)){
        if((L = read(0, buff, length)) < 1) return 0;
    }
    return (size_t)L;
}
```

После внесения указанных изменений в файл ***avk_tty.c***, следует его откомпилировать в среде системы разработки и убедиться, что:

1) отсутствуют сообщения об ошибках в самом файле;
2) отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_tty.c***;
3) можно, при желании, отредактировать файл ***avk_fb_compositor.c***, вызова и использования модифицированных функций, хотя — это в обязательном порядке будет сделано на последнем этапе преобразований.

Примечание — В файле ***avk_compositor.h***, замечания об ошибках могут оставаться до тех пор, пока не будут устранены все ошибки вызова функций во всех файлах проекта. **Внимание!** Системе разработки потребуется подключение библиотеки ***pthread***.

Этап 3. Нить устройства мыши.

Все концептуальные замечания, высказанные о ПО компоненты виртуального терминала, являются справедливыми и к компоненте устройства мыши.

В компоненте устройства мыши, все изменения ПО проводятся в рамках уже определённых функций:

1) функция ***avk_mouse_get()*** будет реализовывать алгоритм нити; здесь необходимо учесть, что такая функция должна в качестве аргумента иметь указатель на тип ***void*** и возвращать указатель на такой же тип; кроме того, алгоритм этой нити должен самостоятельно вычислять абсолютные координаты мыши, освободив от этой функции алгоритм композитора;
2) функция ***avk_mouse_open()***, кроме обязанности открытия устройства мыши, дополняется функционалом запуска своей нити, а также функционалом создания изображения мыши и занесением соответствующих указателей ***datac***, ***surc*** и ***crc*** в глобальную структуру данных, для последующего использования их композитором;
3) функция ***avk_mouse_close()***, кроме обязанности закрытия устройства мыши, дополняется функционалом завершения работы нити, удаления изображения курсора и освобождения выделенной ранее оперативной памяти компьютера.

Все указанные изменения в файле ***avk_mouse.c*** показаны на листинге 2.14.

Листинг 2.14 — Новая компонента ПО устройства мыши

```
/*
 * avk_mouse.c
 *
 *  Created on: 18 авг. 2017 г.
 *      Author: upk
 */

#include "avk_compositor.h"

/**
 * Изменения:
 * Закрытие устройства мыши.
 */
void
avk_mouse_close(avk_context_t *ctx){
	/**
	 * Удаляем нить, если она есть.
	 */
	if(ctx->thr_c > 0){
		pthread_cancel(ctx->thr_c);
		pthread_join(ctx->thr_c, NULL);
	}
	/**
	 * Удаляем объекты библиотеки cairo.
	 */
	if(ctx->crc != NULL)
		cairo_destroy(ctx->crc);
	ctx->crc = NULL;
	if(ctx->surc != NULL)
		cairo_surface_destroy(ctx->surc);
	ctx->surc = NULL;
	/**
	 * Освобождаем память изображения.
	 */
	if(ctx->datac != NULL)
		free(ctx->datac);
	ctx->datac = NULL;
	/**
	 * Закрываем устройство.
	 */
```

```
      close(ctx->fdc);
      ctx->fdc = -1;
}

/**
 * Изменения:
 * Открытие устройства мыши:
 *
 *  0 - устройство открыто;
 * -1 - ошибка открытия устройства.
 */
int
avk_mouse_open(avk_context_t *ctx){
      char *title =
                  "Проект avk_mouse_open:";
    /**
     * Открываем устройство мыши для чтения
     * и без блокировки.
     */
    if((ctx->fdc = open(MOUSEFILE, O_RDONLY |
            O_NONBLOCK)) == -1) {
        printf("%s\tОшибка открытия устройства мыши...\n",
                  title);
        exit(-1);
    }
    /**
     * Характеристики курсора инициализируются в главной нити
     * программы, до вызова этой функции.
     *
     * Выделяем память для изображения курсора.
     */
      ctx->datac = malloc(ctx->wc*4*ctx->hc);
      if(ctx->datac == NULL){
      printf("%s\t\tОшибка выделения памяти ...\n",
                  title);
            return -1;
      }
    /**
     * Создаем cairo_surface_t - поверхность курсора.
     */
    ctx->surc = cairo_image_surface_create_for_data(ctx->datac,
            CAIRO_FORMAT_ARGB32, ctx->wc, ctx->hc, 4*ctx->wc);

    cairo_status_t ret = cairo_surface_status(ctx->surc);
    if(ret != CAIRO_STATUS_SUCCESS){
      printf("%s\t\tне могу создать cairo_surface ...\n",
                  title);
      printf("%s\t\tномер состояния cairo_surface = %s\n",
                  title, cairo_status_to_string (ret));
    }
    /**
     * Создаем cairo_t - управление рисунком.
     */
    ctx->crc = cairo_create(ctx->surc);
      if(cairo_status(ctx->crc) != CAIRO_STATUS_SUCCESS){
            cairo_destroy (ctx->crc);
            cairo_surface_destroy (ctx->surc);
      printf("%s\t\tне могу создать управление окном ...\n",
                  title);
            if(ctx->datac != NULL)
```

```
                free(ctx->datac);
            ctx->datac = NULL;
            return -1;
        }
        /**
         * Курсор большой, треугольный,
         * черно-белый и полупрозрачный.
         */
        memset(ctx->datac, 0, ctx->wc*4*ctx->hc);       // Обнуляем данные.
        cairo_set_source_rgba(ctx->crc, 0, 0, 0, 0.5); // Цвет фона черный.
        cairo_move_to(ctx->crc, 0,       0);           // Рисуем контур.
        cairo_line_to(ctx->crc, ctx->wc, ctx->hc);
        cairo_line_to(ctx->crc, 0,       ctx->hc);
        cairo_line_to(ctx->crc, 0,          0);

        cairo_fill_preserve(ctx->crc);            // Заполняем цветом.

        /**
         * Обводка контура курсора.
         */
        cairo_set_source_rgba(ctx->crc, 1.0, 1.0, 1.0, 0.5);// Цвет белый.
        cairo_set_line_width(ctx->crc, 2);  // Контур - 2 пикселя.
        cairo_stroke(ctx->crc);             // Выполняем.

        /**
         * Запускаем нить курсора.
         * Получаем дефолтные значения атрибутов нити.
         */
        pthread_attr_init(&(ctx->attr_c));
        /**
         * Стартуем отдельную нить.
         */
        if (pthread_create(&(ctx->thr_c), &(ctx->attr_c),
                    avk_mouse_get, (void*)ctx) != 0) {
            printf("%s\t\tНе могу запустить нить курсора...\n",
                        title);
            return -1;
        }

    return 0;
}

/**
 * Изменения:
 * Чтение и установка абсолютных координат мыши.
 *
 * Возвращает значения: реализует алгоритм нити.
 */
void *
avk_mouse_get(void *arg){
      avk_context_t *ctx =
                  (avk_context_t *)arg;
    /**
     * Структура, читаемая с устройства мыши.
     */
    struct input_event ie;
    /**
     * Рабочие переменные координат мыши.
     */
    char x,  y;
```

```
int  xt, yt;

  /**
   * Указатель на читаемую структуру.
   */
unsigned char *ptr = (unsigned char*)&ie;

  /**
   * Читаем в цикле структуру данных мыши.
   * Поскольку чтение осуществляется без блокировки,
   * то получение отрицательного значения говорит,
   * что данные отсутствуют.
   */
while(1){
        /**
         * Проверяем внешнее предложение завершить работу нити.
         */
        pthread_testcancel();   // Возможен выход здесь.
        if(ctx->if_exit)
              break;            // На выход!!!

        usleep(20*1000);        // Период обслуживания 20 ms.
        if(ctx->no_fb ==1){     // Терминал не является текущим.
              /**
               * Координаты курсора менять не будем.
               */
              usleep(300*1000); // Засыпаем на 0.3 секунды.
              continue;
        }
        if(read(ctx->fdc, &ie, sizeof(struct input_event)) <= 0){
              continue;         // Данные курсора отсутствуют.
        }
       /**
        * Первый байт структуры - состояния кнопок.
        */
       unsigned char  button=ptr[0];
       ctx->bLeft   = button & 0x1;
       ctx->bMiddle = ( button & 0x4 ) > 0;
       ctx->bRight  = ( button & 0x2 ) > 0;

       /**
        * Относительные значения координат.
        */
       x=(char) ptr[1];
       y=(char) ptr[2];

       /**
        * Вычисление абсолютных координат мыши.
        */
       xt = ctx->xc + x;
       yt = ctx->yc - y;
         /**
          * Курсор должен быть полностью в пределах экрана.
          */
         if(xt < 0)  xt = 0;
         if(yt < 0)  yt = 0;
         if(xt >= ctx->w)  xt = ctx->w - 1;
         if(yt >= ctx->h)  yt = ctx->h - 1;
         ctx->xc = xt;
         ctx->yc = yt;
```

```
    }
      /**
       * Полный выход из нити.
       */
      pthread_exit(0);  // Завершаем работу нити.
}
```

После внесения указанных изменений в файл ***avk_mouse.c***, следует его откомпилировать в среде системы разработки и убедиться, что:

1) отсутствуют сообщения об ошибках в самом файле;
2) отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_mouse.c***;
3) можно, при желании, отредактировать файл ***avk_fb_compositor.c***, вызова и использования модифицированных функций, хотя — это в обязательном порядке будет сделано на последнем этапе преобразований.

Примечание — В файле ***avk_compositor.h***, замечания об ошибках могут оставаться до тех пор, пока не будут устранены все ошибки вызова функций во всех файлах проекта. **Внимание!** Системе разработки потребуется подключение библиотеки ***cairo***.

Этап 4. Компонента фреймбуфера и родительского окна.

Все концептуальные замечания, высказанные о ПО компонент виртуального терминала и мыши, являются справедливыми и к данной компоненте.

Главная особенность этой компоненты — отсутствие собственной нити, а также использование двух окон изображений.

В компоненте устройства фреймбуфера и родительского окна, все изменения ПО проводятся в рамках уже определённых функций:

1) функция ***avk_fb_paint()*** будет реализовывать алгоритм заполнения родительского окна и перенос его содержимого в окно композитора; в нашей задаче можно было бы обойтись и без этой функции, перенеся ее алгоритм в функцию ***avk_fb_open()***;
2) функция ***avk_fb_open()***, кроме обязанности открытия и инициализации устройства фреймбуфера, дополняется функционалом создания объектов библиотеки ***cairo*** для двух окон;
3) функция ***avk_fb_close()***, кроме обязанности закрытия устройства фреймбуфера, дополняется функционалом удаления объектов библиотеки ***cairo*** и освобождения выделенной ранее оперативной памяти компьютера.

Все указанные изменения в файле ***avk_fb.c*** показаны на листинге 2.15.

Листинг 2.15 — Новая компонента ПО устройства фреймбуфера

```
/*
 * avk_fb.c
 *
 *  Created on: 18 авг. 2017 г.
 *      Author: upk
 */

#include "avk_compositor.h"

/**
 * Удаление cairo устройства для framebuffer
 */
void
asufb_device_destroy(void * arg)
```

```
{
      avk_context_t *ctx = (avk_context_t *)arg;

      if ((ctx->screen != NULL)&&(ctx->stride > 0)){
      /**
       * Завершаю работу.
       */
        munmap(ctx->screen, ctx->stride * ctx->h);
      close(ctx->fdf);
      ctx->fdf = -1;
    }
}

/**
 * Закрытие устройства фреймбуфера.
 */
void
avk_fb_close(void *arg){
      avk_context_t *ctx =
                  (avk_context_t *)arg;
      /**
       * Закрываем созданные объекты библиотеки cairo.
       */
      if(ctx->cr != NULL)
            cairo_destroy (ctx->cr);
      ctx->cr = NULL;

      if(ctx->sur != NULL)
            cairo_surface_destroy (ctx->sur);
      ctx->sur = NULL;
      if(ctx->data != NULL)
            free(ctx->data);
      ctx->data = NULL;

      if(ctx->crf != NULL)
            cairo_destroy (ctx->crf);
      ctx->crf = NULL;

      if(ctx->surf != NULL)
            cairo_surface_destroy (ctx->surf);
      ctx->surf = NULL;
}

/**
 * Открытие устройства фреймбуфера.
 */
int
avk_fb_open(avk_context_t *ctx){
      char * title =
                  "avk_fb_open():";

      /**
       * Открываем устройство фреймбуфера для
       * чтения и записи.
       */
      ctx->fdf = open(FBFILE, O_CLOEXEC | O_RDWR);
      if(ctx->fdf < 0){
            printf("%s\tНе могу открыть файл: %s\n",
                        title, FBFILE);
            return -1;
```

```
}

/**
 * Определяем параметры монитора.
 */
struct fb_fix_screeninfo fix;
if(ioctl (ctx->fdf, FBIOGET_FSCREENINFO, &fix) == -1){
      printf("%s\tНе могу получить информацию о буфере...\n",
                  title);
      close(ctx->fdf);
      return -1;
}
/**
 * Длина строки экрана в пикселях:
 * по 4 байта на пиксель.
 */
ctx->stride = fix.line_length;
/**
 * Читаю формат пикселя в битах.
 */
struct fb_var_screeninfo var;
var.bits_per_pixel = 0;
ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var);
/**
 * Если формат не 32 бита на пиксель, то
 * переустанавливаю это значение.
 */
if(var.bits_per_pixel != 32){
      var.bits_per_pixel = 32;
      ioctl (ctx->fdf, FBIOPUT_VSCREENINFO, &var);
}
/**
 * Снова (для контроля) читаю структуру.
 */
if(ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var) == -1){
      printf("%s\tНе могу получить информацию о буфере...\n",
                  title);
      close(ctx->fdf);
      return -1;
}
/**
 * Проверка формата пикселя.
 */
if(var.bits_per_pixel != 32){
      printf("%s\tНа такой экран - не записываю!!!\n",
                  title);
      close(ctx->fdf);
      return -1;
}
/**
 * С фреймбуфером можно работать как с последовательным
 * символьным файлом, но лучше - как с матрицей целых чисел,
 * поэтому проводим mmap() на память ЭВМ.
 */
ctx->w   = var.xres;               // Ширина фреймбуфера в пикселях.
ctx->h   = var.yres;               // Высота фреймбуфера в пикселях.
int nbuf =
           ctx->h*fix.line_length; // Длина фреймбуфера в байтах.
ctx->screen = (unsigned char *)mmap (0, nbuf, PROT_READ |
           PROT_WRITE, MAP_SHARED,
```

```
                ctx->fdf, 0);
      if(ctx->screen <= 0){
            printf("%s\tНе могу сделать mmap() на фреймбуфер!!!",
                        title);
            close(ctx->fdf);
            return -1;
      }
    /**
     * Создание cairo surface, для рисования в framebuffer
     */
    ctx->surf = cairo_image_surface_create_for_data(ctx->screen,
            CAIRO_FORMAT_ARGB32, var.xres, var.yres, ctx->stride);
    /**
     * Инициализация данными cairo surface: для рисования в framebuffer
     */
    cairo_surface_set_user_data(ctx->surf, NULL, ctx,
                                &asufb_device_destroy);
    ctx->crf = cairo_create(ctx->surf);

    /**
     * Создание окна композитора.
     * Выделяем память для изображения окна композитора.
     */
      ctx->data = malloc(nbuf);
      if(ctx->data == NULL){
      printf("%s\t\tОшибка выделения памяти ...\n",
                  title);
            return -1;
      }
    /**
     * Создаём cairo_surface_t - поверхность композитора.
     */
    ctx->sur = cairo_image_surface_create_for_data(ctx->data,
            CAIRO_FORMAT_ARGB32, ctx->w, ctx->h, ctx->stride);
    /**
     * Создаём cairo_t - управление рисунком композитора.
     */
    ctx->cr = cairo_create(ctx->sur);

    /**
     * Неплохо бы и проверять созданное!!!
     */

    avk_fb_paint(ctx);

      return EXIT_SUCCESS;
}

/**
 * Заполнение окна фреймбуфера цветом.
 */
int
avk_fb_paint(avk_context_t *ctx){
      /**
       *  Заполняем фоновым цветом весь экран.
       *  Новое:
       *  Действия выполняются в буфере композитора, имитируя ситуацию,
       *  что вроде как композитор сам перенёс в буфер родительское окно.
       */
      cairo_set_source_rgba(ctx->cr, 0, 0, 1.0, 1.0); // Цвет фона синий.
```

```
      cairo_move_to(ctx->cr, 0, 0);                    // В начало координат.
      cairo_rectangle(ctx->cr, 0, 0, ctx->w, ctx->h);// Задаём область.
      cairo_fill(ctx->cr);                             // Заполняем цветом.

    /**
     * Выводим текст
     */
      char text[] =
                  "AVK-композитор";
    cairo_select_font_face(ctx->cr, "Sans", CAIRO_FONT_SLANT_NORMAL,
            CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size(ctx->cr, ctx->h/20);

    cairo_text_extents_t ext; // Вычисляем длину текста.
    cairo_text_extents (ctx->cr, text, &ext);
    int wtext = ext.width;

    /**
     * Создаём тень.
     */
    int shade = 5;
    cairo_set_source_rgb (ctx->cr, 0.3, 0.3, 0.0);   // Темно-желтый.
    cairo_move_to(ctx->cr, (ctx->w - wtext)/2 + shade, ctx->h/2);
    cairo_show_text(ctx->cr, text);

    /**
     * Нормальный цвет.
     */
    cairo_set_source_rgb (ctx->cr, 0.6, 0.6, 0.0);   // Жёлтый.
    cairo_move_to(ctx->cr, (ctx->w - wtext)/2, ctx->h/2);
    cairo_show_text(ctx->cr, text);

      return 0;
}
```

После внесения указанных изменений в файл ***avk_fb.c***, следует его откомпилировать в среде системы разработки и убедиться, что:

1) отсутствуют сообщения об ошибках в самом файле;
2) отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_fb.c***;
3) можно, при желании, отредактировать файл ***avk_fb_compositor.c***, вызова и использования модифицированных функций, хотя — это в обязательном порядке будет сделано на последнем этапе преобразований.

Этап 5. Нить динамического рисунка.

Хотя все концептуальные замечания, высказанные о ПО компонент виртуального терминала, мыши и фреймбуфера, являются справедливыми и к данной компоненте, ее реализация имеет ряд существенных особенностей.

Главная особенность этой компоненты — реализация (как и заявлено ранее) в виде одной функции ***avk_dynamic_set()***. Это, само собой, порождает дополнительные особенности ее реализации.

В компоненте функции динамического изображения, которая сама и является реализацией и реализацией этой нити, необходимо учесть следующие моменты:

1) *функция* ***avk_dynamic_set()***, кроме основного цикла нити, должна содержать части кода, которые создают объекты библиотеки cairo и уничтожают их, освобождая выделенную память компьютера;
2) *данная функция нити* должна быть запущена композитором, поскольку она не может запустить сама себя;
3) *нить функции*, имитируя работу некоторого приложения, функционирует независимо от переключений виртуальных терминалов и записывает изображение в свой собственный буфер;
4) *координаты прямоугольной области*, отображаемые нитью этой функции ***avk_dynamic_set()***, в глобальной структуре данных, должны быть относительными, указывать композитору на изменённую часть изображения;
5) *для синхронизации* с композитором, функция ***avk_dynamic_set()***, должна использовать мютекс, для блокировки записи в буфер композитора, и переменную ***k0***, выставляя ее в единицу, когда изображение изменено;
6) для контроля завершения работы нити, может использоваться только глобальная переменная ***if_exit***, поскольку нить сама должна удалять свои объекты.

Все указанные изменения в файле ***avk_dynamic.c*** показаны на листинге 2.16.

Листинг 2.16 — Новая компонента ПО динамического рисунка

```
/*
 * avk_dynamic.c
 *
 *  Created on: 18 авг. 2017 г.
 *      Author: upk
 */

#include "avk_compositor.h"

/**
 * Рисование динамической части
 * изображения в фреймбуфер.
 */
void *
avk_dynamic_set(void *arg){
      char * title =
                  "avk_dynamic_set():";
      avk_context_t *ctx = arg;

      /**
       * Параметры динамической части рисунка:
       * 5 позиций квадрата 40x40 пикселей каждый.
       *
      uint32_t bg = 0x007f7f7f;     // Цвет фона.
      uint32_t fg = 0x00007f00;     // Цвет заполнения.
      uint32_t w0 = 40;             // Ширина прямоугольника.
      uint32_t h0 = 40;             // Высота прямоугольника.
      uint32_t n0 = 5;              // Число  прямоугольников.
      uint32_t k0 = 0;              // Текущее состояние: 0 - изменений нет.
*/

      /**
       * Начальная часть.
       */
      __useconds_t dt = 300*1000;   // Период цикла - для экспериментов.
```

```
  /**
   * Композитор не должен обращаться к изображению.
   */
  ctx->k0     = 0;
  uint32_t kk = 0;        // Состояние отрисовки прямоугольников.
  int      w  = ctx->w0;  // Запоминаем общую ширину области.
  int      w0 = ctx->w0/ctx->n0;// Запоминаем ширину квадрата.
  int      h  = ctx->h0;  // Запоминаем общую высоту области.
  ctx->x0 = 0;
  ctx->y0 = 0;
  /**
 * Выделяем память для динамического изображения.
 */
  ctx->datad = malloc(w*4*h);
  if(ctx->datad == NULL){
  printf("%s\t\tОшибка выделения памяти ...\n",
              title);
  usleep(2000*1000);

  pthread_exit(0);  // Завершаем работу нити.
  }
/**
 * Создаём cairo_surface_t - поверхность изображения.
 */
ctx->surd = cairo_image_surface_create_for_data(ctx->datad,
        CAIRO_FORMAT_ARGB32, w, h, 4*w);
/**
 * Создаём cairo_t - управление рисунком изображения.
 */
ctx->crd = cairo_create(ctx->surd);

/**
 * Неплохо бы здесь и проверять созданное!!!
 */

  /**
   * Цикл нити.
   * Отрисовка динамической части изображения.
   */
  while(1){
        if(ctx->if_exit)  // На выход.
              break;
        usleep(dt);             // Экспериментальная часть задержки.
        /**
         * Начинаем отрисовку.
         */
        pthread_mutex_lock(&(ctx->mutex_comp));   // Захватываем мютех.
        cairo_set_source_rgb(ctx->crd, 0.5, 0.5, 0.5); // Цвет фона.

        if(kk == 0){
              cairo_move_to(ctx->crd, 0, 0);
              cairo_rectangle(ctx->crd, 0, 0, w, h);    // Задаём область.
              cairo_fill(ctx->crd);               // Заполняем цветом.
              ctx->x0 = 0;
              ctx->w0 = w;            // Полная ширина, а высота не меняется.
              ctx->k0 = 1;            // Композитор может брать изображение.
              pthread_mutex_unlock(&(ctx->mutex_comp));// Освобождаем мютех.

              kk = 1;           // Новое значение внутреннего состояния.
              continue;
```

```
		}
		if(kk > 1){
			cairo_move_to(ctx->crd, (kk-2)*w0, 0);
			// Задаём область.
			cairo_rectangle(ctx->crd, (kk-2)*w0, 0, w0, h);
			cairo_fill(ctx->crd);   // Заполняем цветом.
			// Если композитор исполнил предыдущие изменения.
			if(ctx->k0 == 0)
			ctx->x0 = (kk-2)*w0;
		}
		cairo_set_source_rgb(ctx->crd, 0, 0.5, 0);      // Цвет квадрата.
		cairo_move_to(ctx->crd, (kk-1)*w0, 0);
		cairo_rectangle(ctx->crd, (kk-1)*w0, 0, w0, h); // Задаём область.
		cairo_fill(ctx->crd);                           // Заполняем цветом.

		/**
		 * Проверки.
		 */
		if(ctx->x0 >= kk*w0)
			ctx->x0 = 0;
		ctx->w0 = kk*w0 - ctx->x0;

		kk++;
		if(kk > ctx->n0)
			kk = 0;

		ctx->k0 = 1;      // Композитор может брать изображение.

		pthread_mutex_unlock(&ctx->mutex_comp);   // Освобождаем мютех.
	}

	/**
	 * Завершающая часть.
	 *
	 * Удаляем объекты библиотеки cairo.
	 */
	if(ctx->crd != NULL)
		cairo_destroy(ctx->crd);
	ctx->crd = NULL;
	if(ctx->surd != NULL)
		cairo_surface_destroy(ctx->surd);
	ctx->surd = NULL;
	/**
	 * Освобождаем память изображения.
	 */
	if(ctx->datad != NULL)
		free(ctx->datad);
	ctx->datad = NULL;

	/**
	 * Полный выход из нити.
	 */
	pthread_exit(0);  // Завершаем работу нити.
}
```

После внесения указанных изменений в файл ***avk_dynamic.c***, следует его откомпилировать в среде системы разработки и убедиться, что:

1) отсутствуют сообщения об ошибках в самом файле;

2) отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_dynamic .c***.

2.3.4 Реализация проекта avk_fb_compositor

Выполнив организацию ПО отдельных компонент нашего проекта, переходим к реализации главной части ПО, проводящей синхронизацию (композицию) всех функциональных элементов системы.

Для этого, необходимо:

1) *перейти* к ранее созданному файлу проекта: ***avk_fb_compositor.c***;
2) *включить* в него заголовочный файл ***avk_compositor.h***;
3) *реализовать* в функции ***main()*** целевой алгоритм, обеспечивающий решение поставленной задачи.

Общее типовое решение, при реализации функции ***main()***, как и в предыдущем проекте, также состоит из трех частей:

1) первая часть, состоит в *инициализации* всех глобальных переменных и *открытии* используемых устройств;
2) вторая часть, в цикле, *собственно и реализует* алгоритм синхронизации (композитинга) решаемой задачи;
3) третья часть, обеспечивает *согласованное закрытие* всех устройств и *нормальное завершение* работы программы.

Выделим отличительные особенности реализации каждой из частей.

В первой части, изменения касаются прежде всего добавления новых переменных и изменения семантики ряда старых. Максимально полно эти изменения отображены в файле ***avk_compositor.h***, с чем и следует разобраться.

Другой аспект особенностей связан с изменением функциональной нагрузки на функции открытия устройств, с чем нужно разбираться, анализируя исходный текст этих функций.

Дополнительно, нужно учесть, что в этой части должна запускаться нить динамического рисунка.

Общий порядок инициализации программных компонент приложения следующий:

- *первым* запускается устройство терминала и одновременно нить проекта, обслуживающая клавиатуру;
- *затем*, - устройство фрейбуфера, с одновременным созданием буфера композитора и записью в него фонового изображения;
- *последними* запускаются устройства мыши и нить динамического изображения; порядок их запуска — произвольный.

В третьей части, главная задача — остановка ранее запущенных нитей, поскольку их функционирование может создать серьёзные проблемы, при закрытии устройств. Затем проводятся обычные действия по закрытию устройств, как и в предыдущем проекте.

Во второй части проекта, реализующей алгоритм композитора, все изменения связаны с *изменением уровня реализации* взаимодействующих компонент. Композитор освобождён от необходимости самостоятельно определять состояния переключения виртуальных терминалов, анализировать данные клавиатуры, проверяя требование на завершение работы, уточнять координаты мыши и запускать алгоритм динамического изображения.

Композитор обязан:

1) *отреагировать* на команду завершения работы, по значению переменной ***if_exit***;
2) *прекратить* вывод данных в фреймбуфер, по значению переменной ***no_fb***;
3) *обеспечить* полное восстановление изображения экрана, по значению переменной ***dirty***;
4) *синхронизировать* своё взаимодействие с нитью динамического изображения посредством мютекса и значению переменной **k0**;
5) *переписывать* в свой буфер изменённую часть динамического изображения, с последующим выводом в фреймбуфер;
6) *отображать* в фреймбуфере курсор в новой позиции экрана, по значениям переменных ***xc*** и ***yc***, восстанавливая изображение фона в старой его позиции.

Таким образом, в данном проекте, композитор стал более полно реализовывать свои прямые функции, по сравнению с реализацией предыдущего проекта.

Все указанные требования реализованы в тексте файла ***avk_compositor.c***, приведённого на листинге 2.17.

Листинг 2.17 — Реализация главной нити композитора

```
/*
 ============================================================================
 Name        : avk_fb_compositor.c
 Author      : Reznik V.G., 18.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

#include "avk_compositor.h"

int main(void) {
	char *title =
			"Проект avk_fb_compositor:";
	printf("%s\tТреугольный курсор мыши\n",
			title);
	printf("\nКомбинация клавишь Ctrl-X - завершение работы программы!\n");
	printf("Для продолжения - нажми Enter...\n");
	getchar();

	/** ***************************************
	 * Первая часть ПО.
	 * Инициализация данных и открытие устройств.
	 *****************************************/
	/**
	 * Инициализация общей контекстной структуры проекта.
	 */
	avk_context_t ctx;
	/**
	 * Данные виртуального терминала.
	 */
//	ctx.nvt = avk_get_current_vc(0);	// Номер виртуального  терминала.
	ctx.tty_status = avk_setup_tty(&ctx);// Структура состояния терминала.
	if(ctx.tty_status == NULL || ctx.tty_status->changed == 0){
		printf("%s\tНе могу инициализировать виртуальный терминал\n",
				title);
		usleep(2000*1000);
		return -1;
	}
```

```
	/**
	 * Данные фреймбуфера.
	 */
	ctx.x = 0;
	ctx.y = 0;
	ctx.color32 = 0x00ff;
	/**
	 * Запускаем устройство фреймбуфера.
	 */
	if(avk_fb_open(&ctx) < 0){
		printf("%s\tНе могу открыть фреймбуфер\n",
				title);
		usleep(2000*1000);
		avk_restore_tty(&ctx);
		return -1;
	}

	/**
	 * Данные курсора.
	 *
	 * Курсор большой и в середине экрана.
	 */
	ctx.xc = ctx.w/2;	// Курсор - в центр экрана.
	ctx.yc = ctx.h/2;
	ctx.wc = 100;		// Размеры курсора.
	ctx.hc = 200;
//	ctx.colorC = 0x007f7f00; // Цвет курсора.
	/**
	 * Сохраняем координаты мыши, в рабочих переменных.
	 */
	int xt = ctx.xc;
	int yt = ctx.yc;
	/**
	 * Запускаем устройство мыши.
	 */
	if(avk_mouse_open(&ctx) < 0){
		printf("%s\tНе могу открыть устройство мыши\n",
				title);
		usleep(2000*1000);
		avk_fb_close(&ctx);
		avk_restore_tty(&ctx);
		return -1;
	}

	/**
	 * Инициализируем Быстрый мьютекс.
	 */
	if (pthread_mutex_init(&ctx.mutex_comp, NULL) != 0) {
		printf("%s Mutex mutex_comp   initialization failed\n",
				title);
		usleep(2000*1000);
		avk_mouse_close(&ctx);
		avk_fb_close(&ctx);
		avk_restore_tty(&ctx);
		return -1;
	}

	/**
	 * Данные динамической части рисунка.
```

```
       *
       * Здесь семантика переменных сильно изменена!!!
       */
//     ctx.bg = 0x007f7f7f;    // Цвет фона = 0x007f7f7f.
//     ctx.fg = 0x00007f00;    // Цвет заполнения = 0x00007f00.
//     ctx.x0 = ...;           // Относительные координаты изменённой
//     ctx.y0 = ...;           // части динамического рисунка.
       // Число прямоугольников динамического изображения.
       ctx.n0 = 10;
       int x0 = (ctx.w - 40*ctx.n0)/2;// Абсолютные координаты начала окна.
       int y0 = 5*ctx.h/8;
       ctx.k0 = 0;             // Текущее состояние: 0 - данные динамического
                               // изображения отсутствуют; 1 - имеются.
       /**
        * Переменные,  изменяемые самим динамическим изображением.
        * При запуске, нить запомнит эти значения.
        */
       ctx.w0 = 40*ctx.n0; // Ширина прямоугольника динамического изображения.
       // Высота прямоугольника динамического изображения.
       ctx.h0 = 40;
       int x1, y1, w1, h1;     // Рабочие переменные.
       /**
        * Запускаем саму нить динамического изображения.
        *
        * Устанавливаем дефолтные значения атрибутов нити.
        */
       pthread_attr_init(&(ctx.attr_d));
       /**
        * Стартуем саму нить.
        */
       if (pthread_create(&(ctx.thr_d), &(ctx.attr_d),
                   avk_dynamic_set, (void*)&ctx) != 0) {
/*           printf("%s\t\tНе могу запустить нить ...\n",
                        title);
             Завершать работу программы не имеет смысла,
             поскольку это не критично для приложения в целом.
*/
       }

       /**
        * Рабочие переменные.
        */
       int          dirty = 1;       // Нужно полностью перерисовать экран.
       __useconds_t dt    = 20*1000; // Период цикла для экспериментов.

       /** ****************************************
        * Вторая часть ПО.
        * Цикл композитинга, реализующего
        * программный мониторинг состояний устройств.
        ******************************************/
       while(1){
             /**
              * Проверяю, не нужно ли завершить работу.
              */
             if(ctx.if_exit)
                   break;      // Завершаю работу.
             // Засыпаем на заданный период (для экспериментов).
             usleep(dt);
             /**
              * Проверяю, является ли терминал текущим.
```

```
 */
if(ctx.no_fb)
      dirty = 1;
/**
 * Проверяю, нужно ли перерисовывать весь экран.
 */
if(!ctx.no_fb && dirty){
      //avk_fb_paint(&ctx);   // Перерисовываем.
      cairo_set_source_surface (ctx.crf, ctx.sur, 0, 0);
      cairo_paint(ctx.crf);

      dirty  = 0; // Отмечаем выполнение.

      ctx.k0 = 0; // Динамический рисунок также уже отрисован.
}

/**
 * Работа с динамическим рисунком,
 * при условии, что есть изменения.
 */
//avk_dynamic_set(&ctx);
if(ctx.k0){       // Изменения есть.
      pthread_mutex_lock(&(ctx.mutex_comp));    // Захватываем мютех.

      x1 = x0 + ctx.x0;
      y1 = y0 + ctx.y0;
      w1 = ctx.w0;
      h1 = ctx.h0;
      cairo_set_source_surface (ctx.cr, ctx.surd,
                  x0, y0);
      /**
       * Задаём область рисования: рисуем в буфер композитора.
       */
      cairo_rectangle(ctx.cr, x1, y1, w1, h1);

      cairo_fill(ctx.cr);     // Заполняем содержимым.

      ctx.k0 = 0;

      pthread_mutex_unlock(&(ctx.mutex_comp)); // Освобождаем мютех.

      if(!ctx.no_fb){
            cairo_set_source_surface (ctx.crf, ctx.sur, 0,0);
            /**
             * Задаём область рисования: рисуем в фреймбуфер.
             */
            cairo_rectangle(ctx.crf, x1, y1, w1, h1);

            cairo_fill(ctx.crf);    // Заполняем содержимым.
      }
}

/**
 * Проверяю, нужно ли перерисовывать курсор.
 */
//avk_mouse_get(&ctx);// Получаем новые.
/**
 * Если позиция курсора изменилась, то:
 * - восстанавливаем фон экрана, в старой позиции;
 * - рисуем курсор, в новой позиции.
```

```
             */
            if(!ctx.no_fb && (xt != ctx.xc || yt != ctx.yc)){
//                iprintf("Рисую курсор\n");
                  cairo_set_source_surface (ctx.crf, ctx.sur, 0, 0);
                  /**
                   * Задаём область рисования: рисуем в фреймбуфер.
                   */
                  cairo_rectangle(ctx.crf, xt, yt, ctx.wc, ctx.hc);
                  cairo_fill(ctx.crf);    // Заполняем содержимым.

                  xt = ctx.xc;
                  yt = ctx.yc;
                  cairo_set_source_surface (ctx.crf, ctx.surc, xt, yt);
                  /**
                   * Задаём область рисования: рисуем в фреймбуфер.
                   */
                  cairo_rectangle(ctx.crf, xt, yt, ctx.wc, ctx.hc);
                  cairo_fill(ctx.crf);    // Заполняем содержимым.
            }
      }
      /** ****************************
       * Третья часть ПО.
       * Закрытие устройств и завершение
       * работы программы.
       ******************************/
//    printf("%s\tЗавершаю работу main()\n",
//                title);
      usleep(2000*1000);
      avk_mouse_close(&ctx);
//    printf("Вызываю avk_fb_close()\n");
      avk_fb_close(&ctx);
//    printf("Вызываю avk_restore_tty()\n");
      avk_restore_tty(&ctx);

      return EXIT_SUCCESS;
}
```

После внесения указанных изменений в файл ***avk_fb_compositor.c***, следует его откомпилировать в среде системы разработки и убедиться, что:

1) отсутствуют сообщения об ошибках в самом файле;
2) отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_fb_compositor.c***.

Завершив устранение ошибок, следует перейти к запуску и исследованию созданного проекта. В процессе исследования, необходимо обратить внимание на следующее:

- реакцию работы программы на изменение периода цикла изменения динамического изображения;
- реакцию программы на переключения виртуальных терминалов;
- поведение курсора, при указанных выше действиях.

Закончив исследования, следует отразить полученные результаты в личном отчёте.

Особо следует уделить внимание сравнительному анализу полученных результатов по всем работам данной темы.

На этом, лабораторные работы, по данной тематике, можно считать законченными.

3 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Третья тема изучаемой дисциплины рассматривает общие вопросы построения архитектур вычислительных комплексов. Обсуждаются вопросы распараллеливания вычислительных процессов в ЭВМ. Также рассматриваются отдельные современные архитектурные решения. Обязательным является изучение материала, изложенного в источнике [4]. Теоретический материал данной темы закрепляется выполнением двух лабораторных работ.

3.1 Лабораторная работа №7. Применение технологии OpenMP

Рассматривая архитектуры различных вычислительных комплексов, нетрудно прийти к выводу, что общая тенденция создания таких архитектур направлена на распараллеливание вычислений, при большом желании эффективного использования аппаратных средств ЭВМ.

Поскольку современный уровень вычислительной техники является микропроцессорным, то существенную роль в создании вычислительных комплексов начинает играть специальное системное программное обеспечение, которое призвано облегчить и ускорить создание новых архитектур высокопроизводительных систем.

В лабораторных работах предыдущей темы было показано, что применение потоков (*нитей*, *threads*) позволяет, в ряде случаев, как упростить, так и повысить качество решения системных задач. Следует ожидать, что расширение программного арсенала технологий, ориентированных на параллельные вычисления, должно привести к возможности построения более совершенных систем.

Среди такого арсенала технологий следует отметить программные проекты ***OpenMP***, ***MPI*** и ***PVM***, которые мы и рассмотрим, прежде чем преходить к конкретной формулировке тем лабораторных работ.

3.1.1 Технологии параллельных вычислений

Сразу следует оговориться, что все рассматриваемые три проекта относятся к технологиям параллельного программирования, поэтому могут оказаться наиболее эффективными и полезными при построении *вычислительных систем*.

С другой стороны, эти проекты построены, или включают в себя, технологии нитей (легковесных процессов), что:

1) *даёт основание* проводить параллели со стандартной технологией ***Ptheads***;
2) *позволяет* более точно учитывать многопроцессорность компьютера, наличие у него ***SMP***-архитектуры ОС, а также сетевые возможности данных проектов.

Технология OpenMP.

OpenMP (*Open Multi-Processing*) — открытый стандарт для распараллеливания программ на языках программирования ***C/C++*** и ***Fortran***.

Этот стандарт даёт описание совокупности директив компилятору языка, которые позволяют программировать *многопоточные приложения* на *многопроцессорных системах* с архитектурой *общей памяти*.

Стандарт OpenMP был разработан ***в 1997 году***, как ***API***, ориентирован на написание портируемых многопоточных приложений для языка ***Fortran***. В октябре 1998 года появились спецификации ***OpenMP*** для языка ***C***.

Синтаксис *OpenMP* использует модель «*ветвление-слияние*» (***fork-join***), что хорошо подходит для распараллеливания циклов, в которых, как правило, выполняются однотипные операции. Такая модель очень хорошо реализуется на аппаратной платформе ***SMP***-архитектур, а также позволяет воспользоваться преимуществами потоковой модели ***SIMD***.

Семантика *OpenMP* основана на понятии *нити* (***потока***, ***threads***), что позволяет сравнивать её со стандартной моделью ***Ptreads*** стандарта ***POSIX***:

1) *обе модели* используют понятие главной (***master***) нити, которая присутствует всегда, причём ***OpenMP*** нумерует нити целыми числами, начиная с нуля, что позволяет с минимальными затратами на кодирование обрабатывать массивы данных достаточно большого размера;
2) *модель Pthreads* использует для реализации программы классическую парадигму функционального подхода, что проще воспринимается разработчиками ПО на языке С, но требует непосредственного его участия на всех этапах распараллеливания и синхронизации алгоритма решения задачи;
3) *модель OpenMP* опирается на множество специальных директив, перекладывающих многие аспекты распараллеливания и синхронизации алгоритма задачи «на усмотрение» компилятора, что требует от программиста лишь указания последовательности параллельных и последовательных участков кода, а многие аспекты синхронизации выполняются по умолчанию;
4) *хотя обе модели* позволяют использовать произвольное количество нитей, ограниченное лишь лимитами ОС, непосредственное распределение процессоров ЭВМ на конкретные нити процессов - скрыто от разработчика ПО;
5) *хотя модель OpenMP* содержит большое число директив и, детализирующих их требования опций, эффективность применения этой модели для решения задач построения вычислительных комплексов вызывает серьёзные сомнения.

Для учебных целей, прагматика использования модели ***OpenMP*** демонстрируется решением одной из задач мониторинга, которая подробно описана далее в пункте 3.1.2.

В качестве базовых основ описания модели ***OpenMP*** можно воспользоваться:

1) методическим пособием М.В. Васильевой и других «Параллельное программирование на основе библиотек», электронный вариант которого расположен на сайте: http://edu.chpc.ru/parallel/main.html;
2) учебными материалами сайта: http://parallel.ru/tech/tech_dev/openmp.html;
3) справочным материалом по операторам и директивам OpenMP, размещённом в файле *~/Документы/OpenMP-4.5-1115-CPP-web.pdf* рабочей области пользователя ***upk***.

Технология MPI.

MPI (Message Passing Interface) - интерфейс передачи сообщений или программный интерфейс (***API***) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

MPI было предложено ***Уильямом Грууппом*** и ***Эвином Ласком***, в период 1993 - 1994 годов. В 1994 году вышла первая версия этого проекта. 12 июня 1995 года опубликована спецификация ***MPI 1.1***, а её первая реализация появилась в 2002 году, в которой поддерживаются следующие функции:

1) передача и получение сообщений между отдельными процессами;
2) коллективные взаимодействия процессов;
3) взаимодействия в группах процессов;
4) реализация топологий процессов.

18 июля 1997 года опубликована спецификация ***MPI 2.0***, дополнительно поддерживающая функции:

1) динамическое порождение процессов и управление процессами;
2) односторонние коммуникации (***Get/Put***);
3) параллельный ввод и вывод;
4) расширенные коллективные операции: процессы могут выполнять коллективные операции не только внутри одного коммуникатора, но и в рамках нескольких коммуникаторов.

Версия MPI 2.1 вышла в начале сентября 2008 года.
Версия MPI 2.2 вышла 4 сентября 2009 года.
Версия MPI 3.0 вышла 21 сентября 2012 года.

Главные отличия технологии ***MPI*** от технологий ***Ptreads*** и ***OpenMP***:

1) *распараллеливание* реализации программного обеспечения производится на ***уровне процессов***, а не на уровне нитей;
2) *коммуникации* между параллельными компонентами проводится на уровне ***передачи сообщений***, а не на уровне общей памяти.

Для учебных целей, прагматика использования этой модели демонстрируется на известной реализации ***Open MPI***. Сама демонстрация проводится посредством решения одной из задач мониторинга, подробно описанной в следующей лабораторной работе.

В качестве базовых основ описания модели ***MPI*** можно воспользоваться:

1) изучением главы 15 из книги: Jeffrey M. Squyres «Архитектура приложений с открытым исходным кодом», том 2, размещённой на сайте: http://rus-linux.net/MyLDP/BOOKS/Architecture-Open-Source-Applications/Vol-2/openmpi-1.html;
2) учебными материалами сайта: http://parallel.ru/tech/tech_dev/mpi.html;
3) электронным вариантом книги: Оленев Н.Н. «Основы параллельного программирования в системе MPI». - М.: ВЦ РАН. 2005, с.81, размещённом в файле *~/Документы/MPIbook1.pdf* рабочей области пользователя ***upk***.

Кластерное решение по технологии PVM.

Parallel Virtual Machine (***PVM***) - «*виртуальная параллельная машина*», которая представляет собой общедоступный системный программный пакет, позволяющий объединять разнородный набор компьютеров в общий вычислительный ресурс.

Виртуальная параллельная машина (***PVM***) и предоставляет возможность управлять процессами с помощью технологии передачи сообщений.

Существуют реализации ***PVM*** для самых различных платформ, диапазон которых распространяется от нотебуков до суперкомпьютеров ***Gray***. Система поддерживается посредством предоставления специальных библиотек программирования на языках ***Fortran***, ***C/C++***.

PVM является свободным ПО, распространяемым под двумя лицензиями: ***BSD License*** и ***GNU General Public License***. По сравнению с технологией ***MPI***, имеет более расширенные возможности, в плане контроля вычислений: присутствует специализированная консоль управления параллельной системой, а также ее графический эквивалент ***XPVM***, позволяющий наглядно продемонстрировать работу всей системы.

PVM является плодом сотрудничества университетов ***Эмори*** и штата ***Теннсси***. **Работа началась** в 1989 году и в этом же году была выпущена версия ***PVM 1.0***. Последняя версия ***PVM 3.4.6***, выпущена в феврале 2009 года.

Примечание - Проект ***PVM*** — достаточно громоздкий и его изучение не входит в программу нашего курса.

Для самостоятельного изучения, можно обратиться к источникам:

1) документация сайта OpenNET под общим названием «PVM-параллельная виртуальная машина», - http://www.opennet.ru/docs/RUS/linux_parallel/node209.html;
2) по адресу http://chaos.ssu.runnet.ru/dynamics/books/pvm/using_PVM.htm, - описание системы PVM под общим названием «Использование PVM. Введение в программирование»;
3) по адресу http://www.portablecomponentsforall.com/edu/pvm-ru/, - описание, ссылки на документацию и другое, под общим названием «PVM. Parallel Virtual Machine».

Краткие выводы.

Подводя итоги развития средств вычислительной техники и сопутствующих им средств программного обеспечения, нетрудно заметить существующую диалектику, присущую самому этому процессу:

1) стремление *упорядочить* и *согласовать* последовательность выполняемых вычислений, что естественным образом приводить к понятию алгоритма и развитию технологий последовательных действий основанных на логике;
2) стремление *ускорить* процесс вычислений, что естественным образом приводит к идеям и технологиям параллельных вычислений.

Другой аспект вычислительных технологий содержит диалектику связанную со стремлением:

1) обеспечить *надёжность вычислений*, что требует дублирования (распараллеливания) однотипных функциональных блоков, включая контроль их работы;
2) обеспечить *эффективность использования* имеющихся вычислительных ресурсов, включая энергетические ресурсы, что актуально для мобильных систем.

Общая современная направленность технологических решений, обеспечивающих некоторое приемлемое обеспечение сразу всех указанных противоречивых требований основана на выделении *однотипных* и *максимально независимых* объектов данных, на основе которых и строится архитектура вычислительных машин и комплексов.

Это можно показать следующими примерами:

1) эволюция *последовательных блочных устройств* от магнитных лент, перфолент и перфокарт к устройствам с *прямым доступом*, таким как магнитные диски, имеющим блоки (сектора) одинаковой длины;
2) замена *сегментной организации* оперативной памяти ЭВМ на *страничную*;
3) широкое распространение технологий RISC-архитектур процессоров, имеющих *одинаковый размер команд*, *конвейерную организацию* микропроцессорных вычислений;
4) *SMP-архитектура* компьютеров с общей памятью и другие.

Серьёзными препятствиями на этом пути являются:

- *последовательная природа* базовых вычислительных операций, например, операции суммирования, которые требуют переноса результата суммирования каждого бита слагаемых;
- *необходимость согласования кешей*, при построении SMP-архитектур ЭВМ;
- *неоднородная*, с точки зрения типизации объектов данных, память в сложных системах;
- *распределенная*, *динамичная* и *гетерогенная* природа большинства практически значимых приложений и задач.

Таким образом мы приходим к выводу, что построение единой универсальной архитектуры ЭВМ или вычислительного комплекса является неприемлемым решением, поскольку в своей основе она должна отражать вычислительную базу будущих вычислительных систем. В этой ситуации наиболее важными является изучение технологий ***OpenMP*** и ***Open MPI***, поскольку они отражают основную суть указанных выше диалектически разнонаправленных тенденций, на которых и должна строится целевая архитектура любого вычислительного комплекса.

3.1.2 Технология OpenMP

Приступая к практическому освоению технологии ***OpenMP***, мы должны помнить и сравнивать ее с технологией ***Pthreads***, которая к тому же подтверждена стандартами ***POSIX***:

- обе технологии ориентированы на ***SMP***-архитектуру ЭВМ или комплекса;
- обе технологии проводят распараллеливание процесса посредством нитей («***легковесных***» ***процессов***, ***threads***);
- обе технологии только заказывают необходимое количество нитей, а непосредственное распределение их по процессорам ***SMP***-архитектуры осуществляется средствами ОС.

Несмотря на общее сходство, идеологическая и библиотечная реализации этих технологий имеют существенные различия, которые обязательно следует учитывать во время практического их применения. Описание явных отличий этих технологий начнём с ***Ptreads***, а затем раскроем на конкретных примерах использования ***OpenMP***.

Технология *Pthreads*, практическое применение которой было продемонстрировано в лабораторной работе №6, имеет следующие особенности:

1) использует функции явного *создания* и *уничтожения* нитей;
2) каждая нить имеет свое *конкретное назначение* и *функциональную реализацию*;
3) *для синхронизации* работы нитей используются дополнительные средства ОС и, в частности, - ***мютексы***.

Таким образом, технология ***Pthreads*** предоставляет разработчику мощный инструмент для построения системных программных решений.

Технология *OpenMP*, являющаяся предметом изучения в данной лабораторной работы, основана на парадигме «***ветвлений-слияний***», что поддерживается соответствующими конструкциями самого языка программирования.

Общий вид таких конструкций (для языка С) имеет один из следующих форматов представленных выражениями (3.1) и (3.2).

#pragma omp <директива> [раздел [[,] раздел]...] **(3.1)**
#pragma omp <directive> [clause [[,] clause]...] **(3.2)**

Раздел (***опция***, ***clause***) — является необязательным синтаксическим элементом языковой конструкции и предназначен для конкретизации применения операции <***директива***>.

Директива — это указание компилятору сформировать специальные команды, которые воздействуют на следующий после директивы блок операторов базового языка, обычно ограничиваемый фигурными скобками «***{...}***».

OpenMP поддерживает следующие основные директивы ***parallel***, ***for***, ***parallelfor***, ***section***, ***sections***, ***single***, ***master***, ***critical***, ***flush***, ***ordered*** и ***atomic***.

Каждая из указанных директив, конечно по своему, воздействует на нити процесса, конкретизируя общие идеи:

- нити процесса *имеют целочисленную нумерацию*, начинающуюся с ***номера 0***;
- нить с ***номером 0*** ассоциируется с *главной* (***master***) нитью процесса и существует всегда, в течении его жизненного цикла;
- после функционального блока директивы, создающей множество нитей, происходит *неявная процедура синхронизации*, соответствующая ожиданию завершения работы всех нитей, кроме нити с ***номером 0***.

Такие идеи очень хорошо ложатся на реализацию различных вычислительных алгоритмов:

- *освобождая* программиста от явного кодирования функций создания и удаления нитей;
- *обеспечивая* простейший тип синхронизации «***barrier***», после завершения параллельного блока, что поддерживается также и специальной директивой ***barrier***;
- *реализуя* удобную нумерацию нитей, обеспечивая индексацию алгоритмов работы с массивами данных.

Дополнительно, по сравнению с ***ptreads***:

- поддерживается *механизмы защиты* общих переменных всего процесса, выделяя *приватные* (***private***) и общедоступные (***share***) данные;
- обеспечивается *вложенность директив*, также обеспеченная средствами синхронизации.

В целом может показаться, что технология **OpenMP** мало подходит в качестве технологии, обеспечивающей реализацию системных решений вычислительных комплексов, следует учитывать ряд особенностей присущих сложным системам:

- сложные системы ***практически не реализуются*** в рамках одного процесса, что позволяет использовать разные технологии в отдельных ее компонентах;
- системное ПО вычислительных комплексов ***может содержать компоненты***, запускаемые в рамках отдельных процессов, которые реализуют параллельные алгоритмы обработки данных.

Все это оправдывает изучение и применение технологии ***OpenMP***.

3.1.3 Учебный тестовый пример технологии OpenMP. Проект omp1

За неимением возможности изучать все директивы и варианты использования технологии ***OpenMP***, начнём нашу работу с тестового примера, который почти полностью формируется шаблоном проекта системы разработки ***EclipsePTP***.

По традиции этот пример выводит на консоль терминала приветствие, демонстрируя нам параллельную работу нитей, а заодно и готовность к использованию самой системы разработки.

> Примечание — Для применения технологии ***OpenMP*** нет острой необходимости использования специальных сред разработки. Все современные компиляторы способны реализовывать подобные решения.

Для компилятора ***gcc***, необходимо:

1) включить в исходный текст программы заголовочный файл <***omp.h***>;
2) указать компилятору опцию ***-fopenmp***;
3) обеспечить линковщику доступ к библиотеке ***libomp.so***.

На листинге 3.1 приведён исходный текст тестового примера, реализующего технологию ***OpenMP***.

Текст приведённого примера снабжён необходимыми комментариями, поэтому не требует дополнительных пояснений.

Листинг 3.1 — Тестовый пример применения технологии OpenMP

```
/*
 ============================================================================
 Name        : omp1.c
 Author      : Reznik V.G., 12.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */
#include <omp.h>

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

/**
 * Hello OpenMP World печатает число нитей и
 * текущий id нити.
 */
int main (int argc, char *argv[]) {

  int numThreads, tid, end, nprocs;

  /**
   * Определяем количество процессоров.
   */
  nprocs = omp_get_num_procs();
  printf("Компьютер имеет %i процессоров\n", nprocs);
  /**
   * Характеристики нитей.
   */
  printf("Устанавливаю 10 нитей...\n");
  omp_set_num_threads(10);
  printf("omp_get_max_threads(): %d\n", omp_get_max_threads());

  printf("omp_get_thread_limit(): %d\n", omp_get_thread_limit());

  /**
   * Выполняем команду создания нитей;
   * каждая нить имеет собственную копию переменных.
   */
#pragma omp parallel private(numThreads, tid)
 {
   tid = omp_get_thread_num();
   printf("Привет от thread (нити) номер %d\n", tid);

   /**
    * Эта часть запускается только master-нитью (tid=0)
    */
   if (tid == 0)
     {
       numThreads = omp_get_num_threads();
       printf("Число нитей: %d\n", numThreads);
     }
```

```
 }
 end = omp_get_thread_num();
 printf("После оператора #pragma: нить номер %d\n", end);

 return 0;
}
```

Один из возможных вариантов вывода результатов работы тестовой программы приведён на рисунке 3.1.

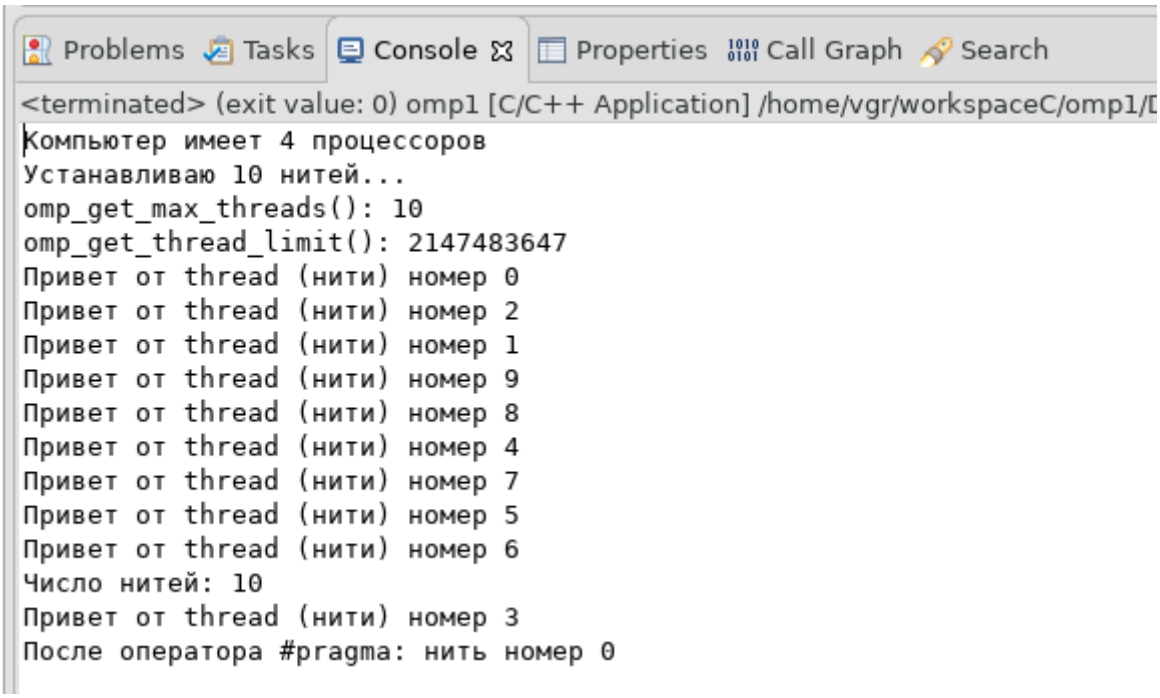

Рисунок 3.1 — Результат работы тестовой программы листинга 3.1

Задание на реализацию проекта тестового примера.

1) создать в среде системы разработки EclipseC проект ***omp1***;
2) скопировать туда текст листинга 3.1 и выполнить действия, указанные в замечании, а затем — запустить проект на выполнение.
3) обязательно запустить проект ***с заданием*** и ***без задания*** количества нитей;
4) отразить в отчёте полученные результаты запуска тестового примера.

3.1.4 Постановка учебной задачи. Проект avk_opemp

Применение технологии ***OpenMP*** продемонстрируем на примере *задачи мониторинга* некоторого заданного набора системных ресурсов вычислительного комплекса.

С целью обобщения и одновременного упрощения учебной задачи, контролируемые ресурсы промоделируем программным способом, подразумевая *некоторый абстрактный ресурс*, который задаётся вектором (структурой) своих параметров. В качестве конкретного вектора абстрактного ресурса выберем набор следующих параметров:

- ***n*** — номер ресурса: целое число от ***1*** до ***N***;
- ***x*** — координата ***x***: целое число со значением в пределах экрана монитора;
- ***y*** — координата ***y***: целое число со значением в пределах экрана монитора.

Главное требование мониторинга ресурсов — контролируемые параметры *должны быть согласованы*: получены «одновременно».

Учебная задача данной лабораторной работы — написание программы мониторинга набора ресурсов вычислительного комплекса с применением технологии ***OpenMP***, которая бы отображала эти ресурсы на экране компьютера, при выполнении следующих дополнительных условий:

1) *отслеживание* переключений виртуальных терминалов, с целью предотвращения вмешательства в их деятельность;
2) отображение каждого отдельного ресурса производится непосредственно в фреймбуфер виртуального терминала;
3) *отслеживание* нажатий на клавиши устройства клавиатуры, с целью обнаружения команды завершения работы программы, которая соответствует нажатию комбинации клавиш ***Ctrl-X***;
4) *вывод* на экран монитора конкретного ресурса производится в виде изображения круга некоторого цвета с указанием в центре номера ресурса;
5) *реализовать* задачу процесса в виде проекта ***avk_openmp***;
6) *использовать* в процессе реализации проекта *уровень прикладного программирования*, как это было определено в лабораторной работе №5;
7) *провести* исследование работы программы данного проекта:
8) *описать* в отчёте полученные результаты.

Общая технология программной реализации поставленной задачи:

- *выделение* отдельных частей программного обеспечения, соответствующего каждой асинхронно работающей компоненте проекта: компоненты *виртуального терминала*, компоненты *фреймбуфера*, компоненты *получения данных вектора* отдельного контролируемого ресурса;
- *объединение* (композитинг) отдельных частей ПО проекта с помощью главной функции: функции-монитора (функция ***main()***).

Для реализации заявленной технологии откроем в системе разработки ***EclipseC*** проект ***avk_openmp*** и приступим к поэтапной реализации этого проекта.

Этап 1. Формирование заголовочного файла задачи.

Заголовочный файл нашего проекта содержит в себе глобальное описание нужных нам структур данных и функций, что вполне оправдано предыдущими примерами.

Учитывая, что даже простейшие задачи, наподобие нашей, используют множество не всегда тривиальных функций, постараемся максимально использовать ПО, уже разработанное в других проектах.

Поскольку качественное рисование во фреймбуфере обеспечивается ПО библиотеки ***cairo***, то за основу будем брать ПО лабораторной работы №6, модифицируя его под требования нашей задачи.

Откроем в нашем проекте заголовочный файл ***avk_openmp.h*** и скопируем в него содержимое заголовочного файла ***avk_fb_compositor.h***.

Удаляем из заголовочного файла:

1) переменные структуры ***avk_context_t***, касающиеся нитей ***pthreads*** и мютексов, а также все, что касается курсора и динамического изображения;
2) описания функций, касающиеся курсора и динамического изображения.

Добавляем в заголовочный файл подключение файла <***omp.h***>.

Результирующий вариант заголовочного файла представлен на листинге 3.2.

Листинг 3.2 — Исходный текст заголовочного файла avk_openmp.h

```
/*
 * avk_openmp.h
 *
 *  Created on: 31 авг. 2017 г.
 *      Author: upk
 */

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

#include <sys/vt.h>
#include <sys/ioctl.h>
#include <termios.h>
#include <string.h>

#include <fcntl.h>
#include <linux/input.h>
#include <signal.h>
#include <errno.h>

#include <stdint-gcc.h>
#include <linux/fb.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

/**
 * Добавляем заголовочный файл для работы с
 * библиотекой cairo.
 */
//#include <pthread.h>
#include <cairo/cairo.h>
#include <syslog.h>

/**
 * Заголовочный файл для работы с технологией OpenMP.
 */
#include <omp.h>

#define iprintf(...)    syslog(LOG_INFO, __VA_ARGS__)

#ifndef AVK_OPENMP_H_
#define AVK_OPENMP_H_

#define FBFILE    "/dev/fb0"

/**
 * Объявление структуры состояния терминала.
 */
typedef struct _avk_tty {
      struct termios oldtty;  // Старое состояние терминала.
      struct termios newtty;  // Новое  состояние терминала.
      int            changed; // Статус состояния терминала.

} avk_tty_t;
```

```
/**
 * Объявление общей контекстной структуры проекта.
 */
typedef struct _avk_contect {
      /**
       * Данные виртуального терминала.
       * Терминал отслеживает необходимость завершения работы
       * и возможность вывода в фреймбуфер.
       */
      avk_tty_t         *tty_status;// Структура состояния терминала.
      int               if_exit;    // Новое: 0 - продолжаем работу; 1 - выход.
      int               no_fb;// Новое: 0 - можно, 1 - нельзя выводить в fb.

      /**
       * Данные фреймбуфера.
       * Добавляем указатели cairo
       * окна композитора и  фреймбуфера.
       * Композитор исполняется главной нитью!!!
       */
      int fdf;                      // Дескриптор фреймбуфера.
      int x, y, w, h;               // Размеры окна фреймбуфера.
      int stride;                   // Длина строки окна в байтах.
      uint32_t color32;             // Цвет заполнения всего экрана  = 0x00ff
      unsigned char * screen;       // Изменение: выровненный по байту указатель
                                    // на данные фреймбуфера.
      unsigned char * data;         // Новое: Указатель на  буфер  композитора.
      cairo_surface_t *surf;        // Новое: Указатель на  окно   фреймбуфера.
      cairo_t         *crf;         // Новое: Контекст  окна фреймбуфера.
      cairo_surface_t *sur;         // Новое: Указатель на  окно композитора.
      cairo_t         *cr;          // Новое: Контекст  окна  композитора.

} avk_context_t;


/** ********************************************************
 * Обявления функций.
 **********************************************************/
/**
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_context_t *ctx);

/**
 * Установка нового состояния теринала.
 * Новое: аргументом является глобальна яструктура.
 */
avk_tty_t *
avk_setup_tty(avk_context_t *ctx);

/**
 * Получить номер текушей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd);

/**
 * Чтение массива байт с терминала:
```

```
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */
size_t
avk_read_tty(void *buff, size_t length);

/**
 * Закрытие устройства фреймбуфера.
 */
void
avk_fb_close(void *arg);

/**
 * Открытие устройства фреймбуфера.
 */
int
avk_fb_open(avk_context_t *ctx);

/**
 * Заполнение окна фреймбуфера цветом.
 */
int
avk_fb_paint(avk_context_t *ctx);


#endif /* AVK_OPENMP_H_ */
```

Этап 2. Формирование компоненты виртуального терминала.

Открываем в проекте файл ***avk_tty.c*** и переносим в него содержимое аналогичного файла из проекта ***avk_compositor***.

В файле ***avk_tty.c*** проводим следующие действия:

1) заменяем подключение заголовочного файла на ***avk_openmp.h***;
2) удаляем функцию обработки нити `avk_tty_thread(void *arg);`
3) в функциях `avk_restore_tty(avk_context_t *ctx)` и `avk_setup_tty(avk_context_t *ctx)` удаляем операторы связанные с созданием и удалением нитей ***pthreads***.

В результате файл ***avk_tty.c***, содержащий функции управления виртуальным терминалом, примет вид показанный на листинге 3.3.

Листинг 3.3 — Исходный текст файла avk_tty.c

```
/*
 * avk_tty.c
 *
 *  Created on: 3 сент. 2017 г.
 *      Author: upk
 */

#include "avk_openmp.h"

/**
```

```
 * Изменено:
 * Функция восстановления состояния терминала,
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_context_t *ctx){

      avk_tty_t *p =
                  ctx->tty_status;
      /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Включаем аппаратный курсор
       */
     printf("\033[?25h\n");

    if(p->changed)
      /**
       * Возвращаем состояние терминала.
       */
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
    /**
     * Освобождаем память, выделенную структуре.
     */
    free(p);
}

/**
 * Изменено:
 * Установка нового состояния теринала.
 * Новое:
 * - аргументом является глобальна яструктура.
 * Результат:
 * - если инициализация не успешна, то выставляем if_exit = 1;
 * - если инициализация успешна, то выставляем if_exit = 0
 * и запускаем нить обслуживания.
 */
avk_tty_t *
avk_setup_tty(avk_context_t *ctx){
      /**
       * Выделяем память под структуру.
       */
      avk_tty_t *p = malloc(sizeof(avk_tty_t));
    if(p == NULL) return NULL;

    /**
       * Очищаем весь экран.
       * Курсор переводим в позицию (1,1).
       */
      printf("\033[2J\033[H\n");
      /**
       * Выключаем аппаратный курсор.
       */
     printf("\033[?25l\n");
    /**
     * Сохраняем состояние терминала.
     */
```

```
    tcgetattr(STDIN_FILENO, &p->oldtty);
    tcgetattr(STDIN_FILENO, &p->newtty);
    /**
     * Переводим терминал в новое состояние:
     * отключаем канонический режим и эхо.
     */
    p->newtty.c_lflag &= ~(ICANON | ECHO | ECHOE );
    if(tcsetattr(STDIN_FILENO, TCSANOW, &p->newtty) < 0){
        /**
         * Если - проблема!!!
         * Возвращаем старое состояние терминала.
         */
      printf("Не могу установить терминал...\n");
        tcsetattr(STDIN_FILENO, TCSANOW, &p->oldtty);
        p->changed   = 0;
        ctx->if_exit = 1;     // Всем на выход!!!
    }else{
      p->changed   = 1;
        ctx->if_exit = 0;
    }

    return p;
}

/**
 * Получить номер текушей виртуальной консоли.
 * Если ошибка, то возвращает: 0xffff
 */
unsigned short
avk_get_current_vc(int fd)
{

      struct vt_stat vs;

      if(ioctl(fd, VT_GETSTATE, &vs) < 0)
            return 0xffff;

      return(vs.v_active);
}

/**
 * Чтение массива байт с терминала:
 *
 * Аргументы функции:
 * @param buff (o) - Внешний буфер для чтения данных;
 * @param length   - Заявленная длина буфера;
 * @return - возвращает число прочитанных байт
 * с ожиданием не более 20 миллисекунд.
 */

size_t
avk_read_tty(void *buff, size_t length){
    ssize_t L = 0;
    fd_set rfds;
    struct timeval tv;
    int retval;
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /**
     * Ожидаем не более 20ms: 50 раз/сек
```

```
     */
    tv.tv_sec = 0; tv.tv_usec = 20*1000;
    /**
     * Проверяем наличие прочитанных данных.
     */
    retval = select(1, &rfds, NULL, NULL, &tv);
    if (!retval) return 0;
    /**
     * Читаем данные, если они доступны.
     */
    if(FD_ISSET(0, &rfds)){
        if((L = read(0, buff, length)) < 1) return 0;
    }
    return (size_t)L;
}
```

Этап 3. Формирование компоненты фреймбуфера.

Открываем в проекте файл ***avk_fb.c*** и переносим в него содержимое аналогичного файла из проекта ***avk_compositor***.

> Примечание — Поскольку компонента фреймбуфера не формирует нитей ***Pthreads***, то изменения здесь — минимальны.

В файле ***avk_fb.c*** проводим следующие действия:

1) заменяем подключение заголовочного файла на ***avk_openmp.h***;
2) в функции `avk_fb_paint(avk_context_t *ctx)` заменяем текст «`AVK-композитор`» на текст «`AVK-OpenMP`».

В результате файл ***avk_fb.c***, содержащий функции управления фреймбуфером и окном композитора, примет вид показанный на листинге 3.4.

Листинг 3.4 — Исходный текст файла avk_fb.c

```
/*
 * avk_fb.c
 *
 *  Created on: 3 сент. 2017 г.
 *      Author: upk
 */

#include "avk_openmp.h"

/**
 * Удаление cairo устройства для framebuffer
 */
void
asufb_device_destroy(void * arg)
{
      avk_context_t *ctx = (avk_context_t *)arg;

    if ((ctx->screen != NULL)&&(ctx->stride > 0)){
      /**
       * Завершаю работу.
       */
        munmap(ctx->screen, ctx->stride * ctx->h);
      close(ctx->fdf);
      ctx->fdf = -1;
    }
}
```

```
/**
 * Закрытие устройства фреймбуфера.
 */
void
avk_fb_close(void *arg){
	avk_context_t *ctx =
			(avk_context_t *)arg;
	/**
	 * Закрываем созданные объекты библиотеки cairo.
	 */
	if(ctx->cr != NULL)
		cairo_destroy (ctx->cr);
	ctx->cr = NULL;

	if(ctx->sur != NULL)
		cairo_surface_destroy (ctx->sur);
	ctx->sur = NULL;
	if(ctx->data != NULL)
		free(ctx->data);
	ctx->data = NULL;

	if(ctx->crf != NULL)
		cairo_destroy (ctx->crf);
	ctx->crf = NULL;

	if(ctx->surf != NULL)
		cairo_surface_destroy (ctx->surf);
	ctx->surf = NULL;
}

/**
 * Открытие устройства фреймбуфера.
 */
int
avk_fb_open(avk_context_t *ctx){
	char * title =
			"avk_fb_open():";

	/**
	 * Открываем устройство фреймбуфера для
	 * чтения и записи.
	 */
	ctx->fdf = open(FBFILE, O_CLOEXEC | O_RDWR);
	if(ctx->fdf < 0){
		printf("%s\tНе могу открыть файл: %s\n",
				title, FBFILE);
		return -1;
	}

	/**
	 * Определяем параметры монитора.
	 */
	struct fb_fix_screeninfo fix;
	if(ioctl (ctx->fdf, FBIOGET_FSCREENINFO, &fix) == -1){
		printf("%s\tНе могу получить информацию о буфере...\n",
				title);
		close(ctx->fdf);
		return -1;
	}
```

```
  /**
   * Длина строки экрана в пикселях:
   * по 4 байта на пиксель.
   */
  ctx->stride = fix.line_length;
  /**
   * Читаю формат пикселя в битах.
   */
  struct fb_var_screeninfo var;
  var.bits_per_pixel = 0;
  ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var);
  /**
   * Если формат не 32 бита на пиксель, то
   * переустанавливаю это значение.
   */
  if(var.bits_per_pixel != 32){
        var.bits_per_pixel = 32;
        ioctl (ctx->fdf, FBIOPUT_VSCREENINFO, &var);
  }
  /**
   * Снова (для контроля) читаю структуру.
   */
  if(ioctl (ctx->fdf, FBIOGET_VSCREENINFO, &var) == -1){
        printf("%s\tНе могу получить информацию о буфере...\n",
                    title);
        close(ctx->fdf);
        return -1;
  }
  /**
   * Проверка формата пикселя.
   */
  if(var.bits_per_pixel != 32){
        printf("%s\tНа такой экран - не записываю!!!\n",
                    title);
        close(ctx->fdf);
        return -1;
  }
  /**
   * С фреймбуфером можно работать как с последовательным
   * символьным файлом, но лучше - как с матрицей целых чисел,
   * поэтому проводим mmap() на память ЭВМ.
   */
  ctx->w   = var.xres;                // Ширина фреймбуфера в пикселях.
  ctx->h   = var.yres;                // Высота фреймбуфера в пикселях.
  int nbuf =
              ctx->h*fix.line_length; // Длина фреймбуфера в байтах.
  ctx->screen = (unsigned char *)mmap (0, nbuf, PROT_READ |
              PROT_WRITE, MAP_SHARED,
              ctx->fdf, 0);
  if(ctx->screen <= 0){
        printf("%s\tНе могу сделать mmap() на фреймбуфер!!!",
                    title);
        close(ctx->fdf);
        return -1;
  }
/**
 * Создание cairo surface, для рисования в framebuffer
 */
ctx->surf = cairo_image_surface_create_for_data(ctx->screen,
        CAIRO_FORMAT_ARGB32, var.xres, var.yres, ctx->stride);
```

```
    /**
     * Инициализация данными cairo surface: для рисования в framebuffer
     */
    cairo_surface_set_user_data(ctx->surf, NULL, ctx,
                                &asufb_device_destroy);
    ctx->crf = cairo_create(ctx->surf);

    /**
     * Создание окна композитора.
     * Выделяем память для изображения окна композитора.
     */
      ctx->data = malloc(nbuf);
      if(ctx->data == NULL){
      printf("%s\t\tОшибка выделения памяти ...\n",
                  title);
            return -1;
      }
    /**
     * Создаём cairo_surface_t - поверхность композитора.
     */
    ctx->sur = cairo_image_surface_create_for_data(ctx->data,
            CAIRO_FORMAT_ARGB32, ctx->w, ctx->h, ctx->stride);
    /**
     * Создаём cairo_t - управление рисунком композитора.
     */
    ctx->cr = cairo_create(ctx->sur);

    /**
     * Неплохо бы и проверять созданное!!!
     */

    avk_fb_paint(ctx);

      return EXIT_SUCCESS;
}

/**
 * Заполнение окна фреймбуфера цветом.
 */
int
avk_fb_paint(avk_context_t *ctx){
      /**
       *  Заполняем фоновым цветом весь экран.
       *  Новое:
       *  Действия выполняются в буфере композитора, имитируя ситуацию,
       *  что вроде как композитор сам перенес в буфер родительское окно.
       */
      cairo_set_source_rgba(ctx->cr, 0, 0, 1.0, 1.0); // Цвет фона синий.
      cairo_move_to(ctx->cr, 0, 0);                   // В начало координат.
      cairo_rectangle(ctx->cr, 0, 0, ctx->w, ctx->h);// Задаем область.
      cairo_fill(ctx->cr);                            // Заполняем цветом.

    /**
     * Выводим текст
     */
    char text[] =
                  "AVK-OpenMP";
    cairo_select_font_face(ctx->cr, "Sans", CAIRO_FONT_SLANT_NORMAL,
            CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size(ctx->cr, ctx->h/20);
```

```
    cairo_text_extents_t ext; // Вычисляем длину текста.
    cairo_text_extents (ctx->cr, text, &ext);
    int wtext = ext.width;

    /**
     * Создаём тень.
     */
    int shade = 5;
    cairo_set_source_rgb (ctx->cr, 0.3, 0.3, 0.0);   // Темно-желтый.
    cairo_move_to(ctx->cr, (ctx->w - wtext)/2 + shade, ctx->h/2);
    cairo_show_text(ctx->cr, text);

    /**
     * Нормальный цвет.
     */
    cairo_set_source_rgb (ctx->cr, 0.6, 0.6, 0.0);   // Желтый.
    cairo_move_to(ctx->cr, (ctx->w - wtext)/2, ctx->h/2);
    cairo_show_text(ctx->cr, text);

      return 0;
}
```

3.1.5 Реализация проекта avk_openmp

Реализацию проекта также проведём в три этапа:

- на этапе 1, создадим основу файла ***avk_openmp.c***, которая содержит уже имеющееся ПО виртуального терминала и фреймбуфера;
- на этапе 2, реализуем функционал ПО, который моделирует контролируемые ресурсы;
- на этапе 3, завершим создание проекта, объединив весь имеющийся функционал, в соответствии с требованиями поставленной задачи.

Этап 1. Реализация функционала виртуального терминала и фреймбуфера.

За основу функционала, реализуемого функцией ***main()*** нашего проекта, возьмём исходный текст файла ***avk_compositor.c***, где:

1) *заменим* файл подключаемого заголовка на ***avk_openmp.h***;
2) *удалим* из исходного текста вызовы функций и определения переменных, касающиеся устройства мыши и динамического рисунка;
3) *добавим* во вторую часть исходного теста функции ***main()***, реализуемого циклом, функционал, реагирующий на переключение виртуальных терминалов, восстановление содержимого фреймбуфера из содержимого буфера композитора, а также реакцию на завершение работы программы по нажатию комбинации клавиш ***Ctrl-X***.

Примечание — Указанные изменения студент может выполнить самостоятельно, учитывая опыт предыдущих лабораторных работ.

Далее необходимо посредством компилирования проекта в системе разработки, убедиться в отсутствии ошибок, а затем проверить реальную работу программы.

Работа программы проверяется следующим образом:

1) *переходим* на виртуальный терминал ***tty3***;
2) *выполняем* ***login*** в терминале от имени пользователя ***upk***;
3) *запускаем* приложение командой ***avk_openmp***.

Примечание - Убедитесь, что имеется *мягкий link* из директории ***~/bin*** на запускаемую программу. Если «линка» нет, то, перейдя в директорию ***~/bin***, создайте его командой выражения (3.3).

В результате запуска проекта на этом этапе, экран монитора должен окрашиваться в синий цвет, а посередине экрана должна быть соответствующая надпись. При нажатии комбинации клавиш ***Ctrl-X***, программа должна завершить работу.

Этап 2. Реализация функционала моделируемых ресурсов.

В соответствии с постановкой задачи, мы моделируем N — ресурсов, каждый из которых задается структурой типа:

```
typedef struct _resurs {// Структура отдельного ресурса.
      int   n;          // Номер ресурса.
      int   x;          // Координата x ресурса.
      int   y;          // Координата y ресурса.
} resurs_t;
```

Полный перечень исходных данных представлен на листинге 3.5.

Листинг 3.5 — Исходные данные модели для файла avk_openmp.c

```
/**
 * Общий контекст проекта.
 */
avk_context_t ctx;

/**
 * Модель ресурсов.
 */
typedef struct _resurs {      // Структура отдельного ресурса.
      int   n;          // Номер ресурса.
      int   x;          // Координата x ресурса.
      int   y;          // Координата y ресурса.
} resurs_t;

int N = 20;             // Число моделируемых ресурсов.

resurs_t N_VECTOR[20];  // Вектор новых  ресурсов.
resurs_t O_VECTOR[20];  // Вектор старых ресурсов.

time_t      nt = 3;                 // Время в сек между замерами ресурсов.
time_t      tt = 0;                 // Текущее время, в секундах.
time_t      tp = 0;                 // Время следующего замера, в секундах.
int n;                              // Рабочая переменная, номер ресурса.

/**
 * Функция, генерирующая отдельный ресурс.
 */
void
set_new_resurs(avk_context_t *pctx, resurs_t *pnew, int n){
      pnew->n = n+1;
      float f = rand();                   // Получаем случайное число.
      pnew->x = (f/RAND_MAX)*pctx->w;     // Устанавливаем координату x.
      f = rand();                         // Получаем случайное число.
      pnew->y = (f/RAND_MAX)*pctx->h;     // Устанавливаем координату y.
//printf("tt=%li tp=%li n=%i x=%i y=%i\n", tt, tp, pnew->n, pnew->x, pnew->y);
}
/**
 * Задаем параметры формы отображаемого ресурса.
```

```
 */
int ro = 20;      // Размер круга в пикселях.
double reds[3]   = {0.5, 0.7, 0.9}; // Цвета для ресурсов.
double greens[3] = {0.9, 0.7, 0.5};
int k;            // Рабочая переменная для выбора цвета окружности.
char snum[10];    // Рабочая переменная для текста номера ресурса.
```

Этап 3. Полная реализация проекта.

Реализацию проекта завершаем модификацией головной функции ***main()***. Для этого:

1) до основного цикла, - генерируем начальные значения координат ресурсов;
2) внутри основного цикла, измеряем текущее время и проверяем необходимость обновления координат ресурсов; когда время обновления — наступило, мы генерируем новые значения, а затем, в цикле, восстанавливаем изображение фреймбуфера из буфера композитора и рисуем окружности, в точках новых координат ресурсов.

Результирующий вариант файла ***avk_openmp.c*** приведён на листинге 3.6.

Листинг 3.6 — Результирующий вариант файла avk_openmp.c

```
/*
 ============================================================================
 Name        : avk_openmp.c
 Author      : Reznik V.G., 31.08.2017
 Version     :
 Copyright   : Your copyright notice
 Description : AVK in C, Ansi-style
 ============================================================================
 */

#include "avk_openmp.h"
#include <time.h>
#include <math.h>

/**
 * Общий контекст проекта.
 */
avk_context_t ctx;

/**
 * Модель ресурсов.
 */
typedef struct _resurs {// Структура отдельного ресурса.
      int   n;          // Номер ресурса.
      int   x;          // Координата x ресурса.
      int   y;          // Координата y ресурса.
} resurs_t;

int N = 20;             // Число моделируемых ресурсов.

resurs_t N_VECTOR[20];  // Вектор новых  ресурсов.
resurs_t O_VECTOR[20];  // Вектор старых ресурсов.

time_t      nt = 3;                 // Время в сек между замерами ресурсов.
time_t      tt = 0;                 // Текущее время, в секундах.
time_t      tp = 0;                 // Время следующего замера, в секундах.
int n;                              // Рабочая переменная, номер ресурса.

/**
```

```
 * Функция, генерирующая отдельный ресурс.
 */
void
set_new_resurs(avk_context_t *pctx, resurs_t *pnew, int n){
      pnew->n = n+1;
      float f = rand();                    // Получаем случайное число.
      pnew->x = (f/RAND_MAX)*pctx->w;      // Устанавливаем координату x.
      f = rand();                          // Получаем случайное число.
      pnew->y = (f/RAND_MAX)*pctx->h;      // Устанавливаем координату y.
//printf("tt=%li tp=%li n=%i x=%i y=%i\n", tt, tp, pnew->n, pnew->x, pnew->y);
}

/**
 * Задаем параметры формы отображаемого ресурса.
 */
int ro = 20;      // Размер круга в пикселях.
double reds[3]   = {0.5, 0.7, 0.9}; // Цвета для ресурсов.
double greens[3] = {0.9, 0.7, 0.5};
int k;            // Рабочая переменная для выбора цвета окружности.
char snum[10];    // Рабочая переменная для текста номера ресурса.

/**
 * Головная функция.
 */
int
main(void) {
      char *title =
                  "Проект avk_openmp:";
      printf("\nКомбинация клавишь Ctrl-X - завершение работы программы!\n");
      printf("Для продолжения - нажми Enter...\n");
      getchar();

      /** ***************************************
       * Первая часть ПО.
       * Инициализация данных и открытие устройств.
       ****************************************/
      /**
       * Инициализация общей контекстной структуры проекта.
       */
      /**
       * Данные виртуального терминала.
       */
      ctx.tty_status = avk_setup_tty(&ctx);// Структура состояния терминала.
      if(ctx.tty_status == NULL || ctx.tty_status->changed == 0){
            printf("%s\tНе могу инициализировать виртуальный терминал\n",
                        title);
            usleep(2000*1000);
            return -1;
      }
      unsigned   short   nvt   =   avk_get_current_vc(0);//   Номер   виртуального
терминала.

      /**
       * Данные фреймбуфера.
       */
      ctx.x = 0;
      ctx.y = 0;
      ctx.color32 = 0x00ff;
      /**
       * Запускаем устройство фреймбуфера.
```

```
	 */
	if(avk_fb_open(&ctx) < 0){
		printf("%s\tНе могу открыть фреймбуфер\n",
				title);
		usleep(2000*1000);
		avk_restore_tty(&ctx);
		return -1;
	}

	/** ************************************
	 * Генерируем начальные значения ресурсов.
	 ************************************/
	omp_set_num_threads(N); // Задаю число нитей.
	resurs_t *pnew;         // Указатель на вектор ресурсов.
	tt = time(NULL);
	tp = tt;
#pragma omp parallel shared(ctx,N_VECTOR) private(n,pnew)
{
		n = omp_get_thread_num();           // Определяю номер нити.
		pnew = &N_VECTOR[n];
		set_new_resurs(&ctx, pnew, n);      // Генерирую данные ресурса.
		// Устанавливаю барьер синхронизации.
#pragma omp barrier
}

	/**
	 * Рабочие переменные.
	 */
	char buffer[20];                    // Буфер чтения данных с клавиатуры.
	int          dirty  = 1;            // Нужно полностью перерисовать экран.
	__useconds_t dt         = 20*1000;  // Период цикла для экспериментов.

	/** ****************************************
	 * Вторая часть ПО.
	 * Цикл композитинга, реализующего
	 * программный мониторинг состояний устройств.
	 ****************************************/
	while(1){
		/**
		 * Проверяю, не нужно ли завершить работу.
		 */
		if(avk_read_tty(buffer, 20) > 0){
			if((int)buffer[0] == 24)
				ctx.if_exit = 1;
		}
		if(ctx.if_exit)
			break;      // Завершаю работу.
		// Засыпаем на заданный период (для экспериментов).
		usleep(dt);
		/**
		 * Проверяю, является ли терминал текущим.
		 */
		if(avk_get_current_vc(0) != nvt)
			ctx.no_fb = 1;
		else
			ctx.no_fb = 0;

		if(ctx.no_fb)
			dirty = 1;
		/**
```

```
 * Проверяю, нужно ли перерисовывать весь экран.
 */
if(!ctx.no_fb && dirty){
      cairo_set_source_surface (ctx.crf, ctx.sur, 0, 0);
      cairo_paint(ctx.crf);

      dirty  = 0; // Отмечаем выполнение.

}

/** ************************************
 * Генерируем начальные значения ресурсов.
 ***************************************/
tt = time(NULL);// Определяем текущее время.
if(tt >= tp){   // Генерируем и отображаем новый ресурс.
      omp_set_num_threads(N); // Задаю число нитей.

#pragma omp parallel shared(ctx,N_VECTOR,O_VECTOR) private(n,pnew)
{
            n = omp_get_thread_num();// Определяю номер нити.
            O_VECTOR[n] = N_VECTOR[n];// Сохраняю старое значение ресурса.
            pnew = &N_VECTOR[n];
            // Генерирую данные ресурса.
            set_new_resurs(&ctx, pnew, n);
            // Устанавливаю барьер синхронизации.
#pragma omp barrier
}
      tp = tt + nt;// Новое значение порога времени обновления значений ресурсов.
      /** ********************************
       * Отображаю новое состояние ресурсов.
       *
       * Привязываю исходный рисунок композитора к окну фреймбуфера.
       */
      cairo_set_source_surface (ctx.crf, ctx.sur, 0, 0);
      /**
       * В цикле:
       * Восстанавливаем области, занятые окружностями ресурсов,
       * изображениями из окна композитора.
       */
      for(int i=0; i< N; i++){
        cairo_rectangle(ctx.crf, O_VECTOR[i].x-ro, O_VECTOR[i].y-ro,
                        2*ro, 2*ro);
        cairo_fill(ctx.crf);
      }
      /**
       * В цикле:
       * Рисуем новое положение ресурсов.
       */
    cairo_select_font_face(ctx.crf, "Sans", CAIRO_FONT_SLANT_NORMAL,
            CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size(ctx.crf, ro);
      for(int i=0; i< N; i++){
            cairo_move_to(ctx.crf, N_VECTOR[i].x, N_VECTOR[i].y);
            /**
             * Устанавливаем цвет круга и рисуем его.
             */
            k = i%3;
            // Цвет круга.
            cairo_set_source_rgb(ctx.crf, reds[k], greens[k], 0);
            cairo_arc(ctx.crf, N_VECTOR[i].x, N_VECTOR[i].y,
```

```
                                ro, 0, 2*M_PI);
                        cairo_fill(ctx.crf);
                        /**
                         * Рисуем номер ресурса.
                         */
                        sprintf(snum, "%i", N_VECTOR[i].n);
                        cairo_move_to(ctx.crf, N_VECTOR[i].x-ro/2,
                              N_VECTOR[i].y+ro/2);
                        // Цвет черный.
                        cairo_set_source_rgb (ctx.crf, 0.0, 0.0, 0.0);
                        cairo_show_text(ctx.crf, snum);
                }
        }

    }
    /** ****************************
     * Третья часть ПО.
     * Закрытие устройств и завершение
     * работы программы.
     ******************************/
//  printf("%s\tЗавершаю работу main()\n",
//              title);
    usleep(2000*1000);
//  printf("Вызываю avk_fb_close()\n");
    avk_fb_close(&ctx);
//  printf("Вызываю avk_restore_tty()\n");
    avk_restore_tty(&ctx);


    return EXIT_SUCCESS;
}
```

Завершив отладку программы, студенту следует провести:

1) исследование работы программы;
2) описание в личном отчёте результатов исследования.

Примечание — Запуск программы производится на отдельном терминале, например ***/dev/tty3***, после соответствующей процедуры ***login*** от имени пользователя ***upk***.

3.2 Лабораторная работа №8. Применение технология MPI

В данной лабораторной работе рассматривается технология ***OpenMPI***.

Open MPI — объединение четырёх исследовательских и академических реализаций MPI с открытым исходным кодом: LAM/MPI, LA/MPI (Лос-Аламосский вариант MPI) и FT-MPI (отказоустойчивый вариант MPI). Вскоре, после создания группы Open MPI, к ней присоединилась команда проекта PACX-MPI.

Первый вариант этой реализации был помещён в Subversion, 22 ноября 2003 года.

3.2.1 Архитектура OpenMPI

Архитектура ***OpenMPI***, состоящая из трех основных слоёв абстракции, представлена на рисунке 3.2.

Рисунок 3.2 — Архитектура OpenMPI

Назначение уровней (слоёв) абстракции, следующее:

1. *Open, Portable Access Layer (**OPAL**)*: Слой OPAL является нижним слоем абстракции проекта Open MPI, обеспечивающим выполнение отдельных процессов посредством утилит и связующего кода, например, связные списки общего назначения, обработка строк, управление отладкой и другие необходимые функции. В этом слое также реализуется *ядро переносимости* Open MPI между различными операционными системами, например, доступ к интерфейсам ***IP***, совместное использование памяти на одном и том же сервере, согласование процессора и памяти, высокоточные таймеры и другое.
2. *Open MPI Run-Time Environment (**ORTE**) («**ор-тей**»)*: обеспечивает предоставление как *интерфейса API*, необходимый для передачи сообщений, так и сопутствующую *систему времени выполнения* для запуска, отслеживания и уничтожения параллельно выполняемых заданий. Для ***Open MPI*** параллельно выполняемое задание состоит из одного или нескольких процессов, которые могут исполняться на нескольких экземплярах операционной системы и могут быть взаимосвязаны друг с другом так,

чтобы они действовали как нечто единое. В простых средах со слабой поддержкой или без поддержки распределенных вычислений *слой ORTE* использует команды **`rsh`** или **`ssh`** для запуска отдельных процессов в параллельно выполняемых заданиях. В более продвинутых HPC-средах (*Hight Performance Computing*, см. рисунок 3.3) обычно есть планировщики и менеджеры ресурсов, которые применяются для «справедливого» распределения вычислительных ресурсов между многими пользователями. В таких средах обычно предоставляются специализированные интерфейсы, предназначенные для запуска и регулирования процессов на вычислительных серверах. В слое *ORTE* поддерживается широкий спектр таких управляемых сред, например: ***Torque/PBS Pro***, ***SLURM***, ***Oracle Grid Engine*** и ***LSF***.

3. *Open MPI (**OMPI**)* — самый высокий уровень абстракции, который является одним единственным слоем видимым приложениям и в котором реализован интерфейс ***API*** для ***MPI***, содержащий семантику передачи сообщений определяемую *стандартом MPI*.

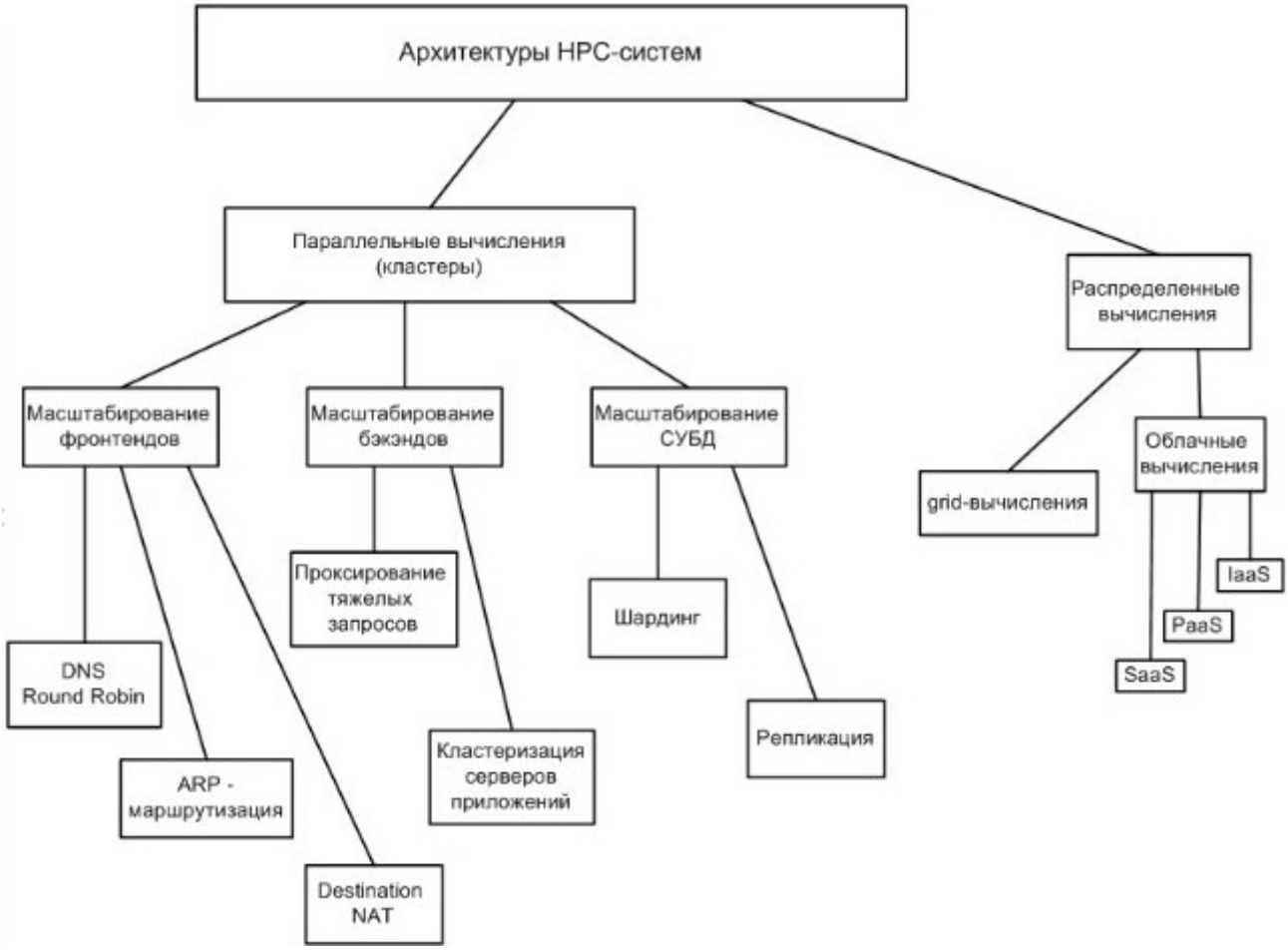

Рисунок 3.3 — Архитектуры HPC-систем

Каждый слой собран в виде отдельной библиотеки:

- библиотека ORTE зависит от библиотеки OPAL;
- библиотеки OMPI зависит от библиотеки ORTE.

Хотя каждая абстракция представляет собой слой, расположенный поверх нижележащего слоя, как показано на рисунке 3.3, слои ORTE и OMPI могут обходить нижележащие слои абстракции и непосредственно взаимодействовать с операционной системой и/или аппаратным обеспечением.

Технология MPI.

Технология MPI основана на ряде определений, составляющих её понятийную основу.

MPI - *message passing interface* - библиотека функций, предназначенная для поддержки работы параллельных процессов в терминах передачи сообщений.

Номер процесса - целое неотрицательное число, являющееся уникальным атрибутом каждого процесса.

Атрибуты сообщения - номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура *MPI_Status*, содержащая три поля:

- *MPI_Source* — номер процесса отправителя;
- *MPI_Tag* — идентификатор сообщения;
- *MPI_Error* — код ошибки: могут быть и добавочные поля.

Идентификатор сообщения ***(msgtag)*** — атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от ***0*** до ***32767***:

1. Процессы объединяются в ***группы***, которые могут быть вложенными группами.
2. Внутри группы все процессы перенумерованы.
3. С каждой группой ассоциирован свой ***коммуникатор***, поэтому при осуществлении пересылки необходимо указать идентификатор группы, внутри которой производится эта пересылка.
4. Все процессы содержатся в группе с заранее предопределенным идентификатором ***MPI_COMM_WORLD***.
5. Технология MPI, по разным оценкам, содержит ***более 300 функций***.

Рассмотрим наиболее значимые функции, отражающие суть самой технологии.

Общие процедуры MPI.

MPI_Init — инициализация параллельной части приложения, синтаксический вид которой задаётся выражением (3.4).

int MPI_Init(int* argc, char* argv)** **(3.4)**

Реальная инициализация для каждого приложения выполняется не более одного раза, а если MPI уже был инициализирован, то никакие действия не выполняются и происходит немедленный возврат из подпрограммы.

Все оставшиеся MPI-процедуры могут быть вызваны только после вызова ***MPI_Init***.

В случае успешного выполнения возвращается значение ***MPI_SUCCESS***, иначе — код ошибки.

MPI_Finalize — завершение параллельной части приложения. Имеет синтаксический вид выражения (3.5).

int MPI_Finalize(void) **(3.5)**

Все последующие обращения к любым MPI-процедурам, в том числе к ***MPI_Init***, — запрещены. К моменту вызова ***MPI_Finalize*** некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Примечание — Представленные две функции MPI_Init() и MPI_Finalize() начинают и завершают использование технологии OpenMPI, поэтому необходимо внимательно следить за их использованием.

Необходимо также помнить, что сложный тип аргументов функции ***MPI_Init()*** предусмотрен для того, чтобы передавать всем процессам технологии OpenMPI аргументы функции ***main()***, как это показано ниже шаблоном выражения (3.6).

```
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
      ...
    MPI_Finalize();
}
```
(3.6)

MPI_Comm_size — определение общего числа параллельных процессов в группе ***comm*** (см. выражение (3.7)), где:

- *comm* - идентификатор группы;
- *size* - размер группы.

```
int MPI_Comm_size( MPI_Comm comm, int* size)
```
(3.7)

MPI_Comm_srank — определение номера процесса в группе ***comm*** (см. выражение (3.8)), где:

- *comm* — идентификатор группы;
- *rank* — номер вызывающего процесса в группе ***comm*** (значение, возвращаемое по адресу ***&rank***, лежит в диапазоне от **0** до ***size_of_group-1***).

```
int MPI_Comm_rank( MPI_Comm comm, int* rank)
```
(3.8)

MPI_Wtime — функция, возвращающая астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Определяется выражением (3.9) и гарантирует, что этот момент не будет изменён за время существования процесса.

```
double MPI_Wtime(void)
```
(3.9)

3.2.2 Приём и передача сообщений между отдельными процессами

Приём и передача сообщений между отдельными процессами выполняется с помощью двух функций: *MPI_Send()* и *MPI_Recv()*.

Примечание — Функции приёма и передачи сообщений технологии OpenMPI выполняются с блокировкой и используют специальные буферы посылки и приёма сообщений.

MPI_Send — функция передачи сообщений с блокировкой, определённая общим выражением (3.10).

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm), где: (3.10)

- *buf* - адрес начала буфера посылки сообщения;
- *count* - число передаваемых элементов в сообщении;
- *datatype* - тип передаваемых элементов;
- *dest* - номер процесса-получателя;
- *msgtag* - идентификатор сообщения;
- *comm* - идентификатор группы.

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*. Все элементы сообщения расположены

подряд в буфере *buf*. Значение *count* может быть нулём. Тип передаваемых элементов *datatype* должен указываться с помощью предопределённых констант типа. Разрешается также передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы.

Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу ***dest***, остаётся за MPI.

Примечание — Следует специально отметить, что возврат из подпрограммы ***MPI_Send()*** не означает ни того, что сообщение уже передано процессу ***dest***, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший ***MPI_Send()***.

MPI_Recv — функция приёма сообщений с блокировкой, определённая общим выражением (3.11).

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Status *status), где: **(3.11)**

- *buf* - адрес начала буфера приема сообщения;
- *count* - максимальное число элементов в принимаемом сообщении;
- *datatype* - тип элементов принимаемого сообщения;
- *source* - номер процесса-отправителя;
- *msgtag* - идентификатор принимаемого сообщения;
- *comm* - идентификатор группы;
- *status* - параметры принятого сообщения.

Приём сообщения с идентификатором *msgtag* от процесса *source* с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в отдельном сообщении, то можно воспользоваться подпрограммой ***MPI_Probe()***.

Блокировка гарантирует, что после возврата из функции все элементы сообщения приняты и расположены в буфере *buf*.

В качестве номера процесса-отправителя можно указать предопределённую константу *MPI_ANY_SOURCE* - признак того, что подходит сообщение от любого процесса.

В качестве идентификатора принимаемого сообщения можно указать константу *MPI_ANY_TAG* - признак того, что подходит сообщение с любым идентификатором.

Примечание — Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову ***MPI_Recv()***, то первым будет принято то сообщение, которое было отправлено раньше.

3.2.3 Учебный тестовый пример. Проект ompi1

Существенной особенностью технологии MPI является понятие коммуникатора.

Коммуникатор — распределённая среда, включающая в себя как локальные процессора компьютера, так и процессора удалённых компьютеров.

Коммуникатор имеет *имя* и предварительные *настройки* на множество компьютеров, которые видны прикладной программе только через *имя коммуникатора*.

Базовое имя коммуникатора — ***MPI_COMM_WORLD***, что освобождает программиста от явного программирования средств доставки сообщений и позволяет ему сосредоточиться на самом алгоритме решения задачи.

Таким образом программист пишет программу, подразумевая, что:

1) выполняется главная нить некоторого процесса;
2) различение процессов производится по целочисленному номеру, что учитывается при написании программы.

На листинге 3.7 приведён исходный код тестового примера, предоставляемого системой разработки, который реализован в пределах проекта ***ompi1***.

Листинг 3.7 — Тестовый пример проекта ompi1

```
/*
 ============================================================================
 Name        : ompi1.c
 Author      : Reznik V.G.
 Version     :
 Copyright   : Your copyright notice
 Description : Hello MPI World in C
 ============================================================================
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[]){
      int  my_rank; /* rank of process */
      int  p;       /* number of processes */
      int source;   /* rank of sender */
      int dest;     /* rank of receiver */
      int tag=0;    /* tag for messages */
      char message[100];    /* storage for message */
      MPI_Status status ;   /* return status for receive */

      /* start up MPI */

      MPI_Init(&argc, &argv);

      /* find out process rank */
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      /* find out number of processes */
      MPI_Comm_size(MPI_COMM_WORLD, &p);

      if (my_rank !=0){
            /* create message */
            sprintf(message, "Hello MPI World from process %d!", my_rank);
            dest = 0;
            /* use strlen+1 so that '\0' get transmitted */
            MPI_Send(message, strlen(message)+1, MPI_CHAR,
               dest, tag, MPI_COMM_WORLD);
      }
      else{
            printf("Hello MPI World From process 0: Num processes: %d\n",p);
            for (source = 1; source < p; source++) {
                  MPI_Recv(message, 100, MPI_CHAR, source, tag,
                        MPI_COMM_WORLD, &status);
```

```
                printf("%s\n",message);
            }
        }
        /* shut down MPI */
        MPI_Finalize();


        return 0;
}
```

Замысел алгоритма программы — достаточно прост:

1) процессы, с номерами отличными от ***0***, посылают текстовое сообщение типа «***Hello MPI World...***» процессу с номером ***0***;
2) процесс с номером ***0*** читает эти сообщения и распечатывает их на терминале.

На рисунке 3.4 представлены два варианта запуска результирующей программы:

1) *вариант 1* — просто запуск;
2) *вариант 2* — запуск с использованием утилиты ***mpirun***.

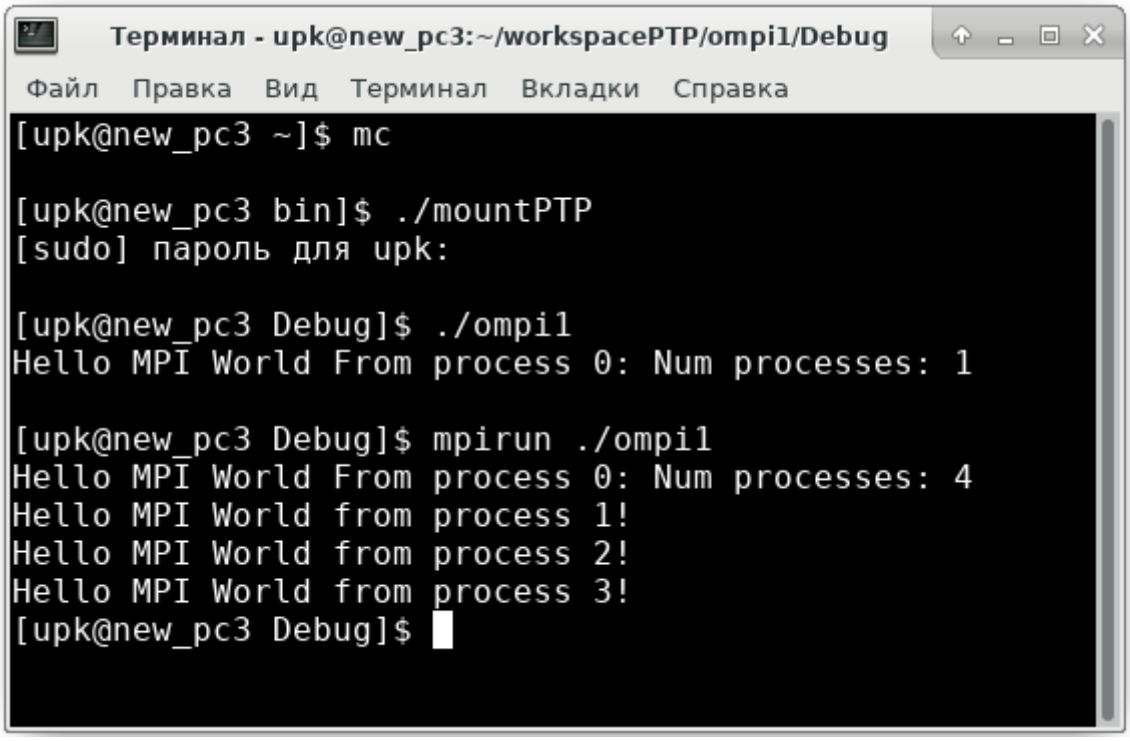

Рисунок 3.4 — Два варианта запуска программы ompi1

Из рисунка 3.4 хорошо видно, что сама программа реализует только один процесс, поэтому для её запуска используется специальная утилита ***mpirun*** или её эквиваленты: ***mpiexec*** и ***orterun***.

Основная задача этих утилит — параллельный запуск нужного количества процессов и распределение их по вычислительным ресурсам вычислительного комплекса. Для этих целей используется общий синтаксис запуска определённый выражением ().

mpirun -np <#> <имя_программы> <аргументы> , где: **(3.12)**

где **<#>** — число параллельно запущенных процессов.

В общем случае утилита ***mpirun*** имеет достаточно много аргументов, которые обеспечивают различные варианты запуска приложений. Для изучения всех опций запуска этой утилиты следует использовать руководство ***man***.

Примечание - При создании приложений по технологии ***OpenMPI*** необходим специальный препроцессор mpicc, поэтому в данной лабораторной работе используется специальная среда разработки ***EclipsePTP***.

Для запуска приведённого учебного примера следует:

1) предварительно запустить сценарий ***~/bin/mountPTP***, который подмонтирует нужную среду разработки ***eclipse*** (см. содержимое сценария);
2) запустить среду разработки ***EclipsePTP***, воспользовавшись соответствующим значком, расположенным на рабочем столе;
3) создать проект ***ompi1***, выбрав в системе разработки соответствующий нужный шаблон проекта.

Примечание — Имеются также утилиты *oshrun* и *shmemrun*, которые запускают параллельные процессы, используя разделяемую память компьютера: *SHMEM*. В общем случае, выбор средств запуска программы ложится на плечи разработчика.

Задание на реализацию и исследование проекта ompi1.

1) реализовать тестовый пример;
2) исследовать запуск приложения по технологии OpenMPI;
3) описать результаты в личном отчёте.

3.2.4 Использование OpenMPI в архитектуре ВК

В данном подразделе, сокращение ***ВК*** будет использоваться как аббревиатура понятия «*вычислительный комплекс*».

Все реализации MPI обычно используются в средах «*высокопроизводительных вычислений*» (HPC).

HPC — *High-Performance Computing* — высокопроизводительные вычисления.

Интерфейс MPI — соединения типа ***IPC*** для программ моделирования, вычислительных алгоритмов и других приложений, требующих большого объёма вычислений.

IPC — ***I**nter-**P**rocess **C**ommunication* — обмен данными между потоками (нитями) одного или разных процессов. Он реализуется посредством механизмов, предоставляемых ядром ОС и реализующим новые возможности межпроцессного взаимодействия.

Взаимодействие между процессами может осуществляться как на одном компьютере, так и между несколькими компьютерами в сети или объединёнными с помощью специальных адаптеров, например, InfiniBand.

Основные механизмы, предоставляемые ОС и используемые средвами ***IPC***:

1) механизмы *обмена сообщениями*;
2) механизмы *синхронизации*;
3) механизмы *разделения памяти*;
4) механизмы *удаленных вызовов* (***RPC*** — Remote Procedur Call).

Infiniband (***IB***) — высокоскоростная коммутируемая компьютерная сеть, используемая в высокопроизводительных вычислениях, которая имеет:

- очень большую пропускную способность;
- низкую задержку, при передаче данных.

Архитектура плагинов OpenMPI.

Ранее, на рисунке 3.1, уже была представлена многослойная архитектура ***OpenMPI***, которая реализована в виде отдельных библиотек. Отдельные слои технологии *OMPI*, *ORTE* и *OLAP* имеют стандартизированный стабильный интерфейс, обеспечивающий надёжную реализацию всей системы.

Для того, чтобы библиотеки системы было удобно использовать, их стали реализовывать в виде компонент, загружаемых во время исполнения программы. Такие компоненты представляют собой динамически разделяемые объекты, часто называемые «DSO» или «плагины». Чтобы компоненты или «плагины» было удобно использовать, была проведена их типизация. Каждый тип компонента представлен в виде *фреймворка*.

Фреймворк (***framework*** **—** каркас, структура) — программная платформа, определяющая структуру программной системы, облегчающая разработку и объединение разных компонентов большого программного проекта.

Каждый компонент технологии ***OpenMPI*** принадлежит ровно одному фреймворку, а фреймворк поддерживает ровно один вид компонента.

Набор слоёв проекта Open MPI — его фреймворки и компоненты образуют модульную архитектуру компонентов - *Modular Component Architecture* (***MCA***).

На рисунке 3.5 приведена общая компоновка архитектуры проекта ***Open MPI***, на которой показаны несколько фреймворков ***Open MPI*** и некоторые из имеющихся компонентов. Остальные фреймворки и компоненты ***Open MPI*** подключены к проекту аналогичным образом.

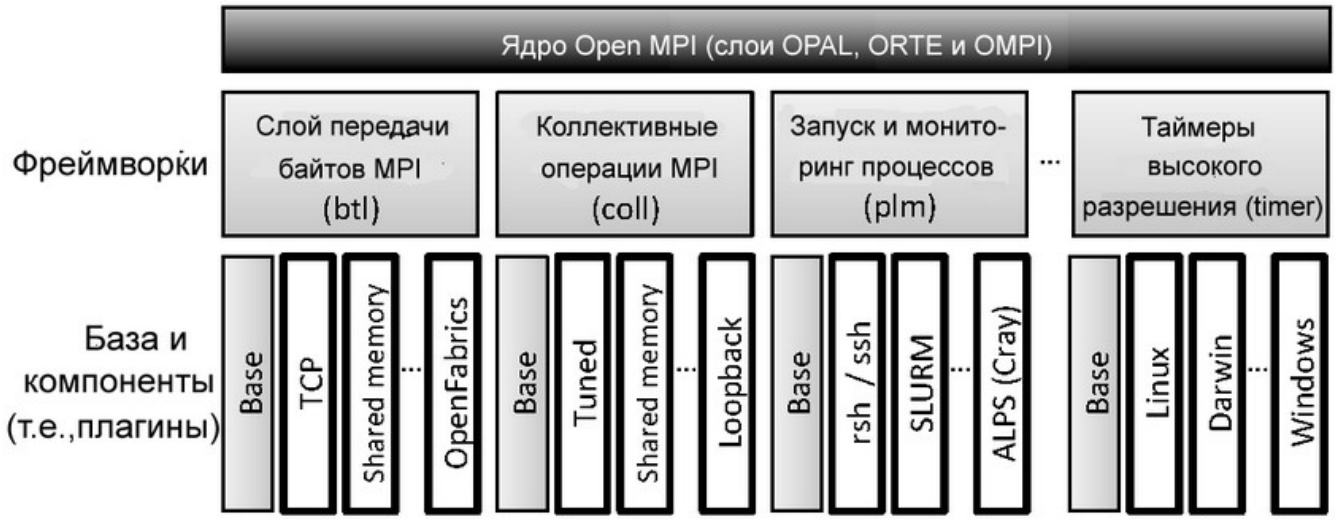

Рисунок 3.5 — Набор компонент технологии OpenMPI

Примечание — Каждый фреймворк, показанных на рисунке 3.5, содержит базовый код ***base*** и один или несколько компонентов. Эта структура *реплицируется* (повторяется) в каждом из слоёв.

На данном рисунке показаны примеры фреймворков всех трёх слоёв:

1) слой *OMPI* — фреймворки ***btl*** и ***coll***;
2) слой *ORTE* — фреймворк ***plm***;
3) слой *OPAL* — фреймворк ***timer***.

Наличие в системе множества фреймворков предоставляет пользователям (администраторам) возможность по-разному соединять различные плагины различных типов и созда-

вать программный стек, который наиболее эффективен в конструируемой архитектуре вычислительного комплекса.

Архитектура технических решений.

Наиболее позитивной частью технологии ***Open MPI*** является возможность разделения *прикладной части* ПО разрабатываемой системы и ПО её *системной части*.

Прикладная часть ПО — отдельная программа или набор различных программ, реализующих некоторый вычислительный алгоритм. Эти программы могут использовать инструментальные средства MPI или нет:

- *программы, использующие* инструментальные средства MPI, могут определить свой номер процесса (***rank***), а также передавать сообщения друг другу;
- *программы, не использующие* инструментальные средства MPI, не имеют возможности взаимодействовать друг с другом.

В любом случае, разработанные программы переносятся на те ЭВМ, на которых они будут в дальнейшем запускаться.

Системная часть ПО — и нструментальные средства самого пакета ***Open MPI***. В неё также входит набор библиотек и утилит.

Главная утилита системной части ПО — утилита ***mpirun***. Именно она создаёт отдельные *локальные и удалённые процессы*, в которых запускает прикладную часть ПО, обеспечивая его номерами процессов и средствами коммуникации.

Таким образом, технология MPI освобождает прикладного программиста от проблем, связанных с программированием низкоуровневого сетевого интерфейса и распределения нагрузки между системными средами ЭВМ, перекладывая эти заботы на архитекторов вычислительного комплекса.

Поскольку ЭВМ, входящие в вычислительный комплекс, могут иметь различные средства коммуникации, различную вычислительную нагрузку и даже различные ОС, все эти различия и дополнительные требования должны быть учтены при запуске утилиты ***mpirun***.

Для примера приведём три из множества возможных вариантов использования утилиты ***mpirun***:

- запуск трёх процессов на отдельной ЭВМ согласно выражения (3.13).

mpirun -np 3 программа **(3.13)**

Здесь три экземпляра одной программы запускаются на локальной ЭВМ и, для взаимодействия между ними, используется разделяемый «механизм» памяти;

- запуск трёх процессов на трёх разных ЭВМ согласно выражения (3.14).

mpirun -np 3 —host a,b,c программа **(3.14)**

Здесь три экземпляра одной программы запускаются на трёх разных ЭВМ ***a***, ***b*** и ***c***, а для взаимодействия между ними, обычно, используется сеть ***TCP/IP***;

- запуск трёх разных процессов на трёх разных ЭВМ согласно выражения (3.15).

mpirun -np 1 —host a прог1 : -np 1 —host b прог2 : -np 1 —host C прог3 **(3.15)**

Здесь три экземпляра разных программ (***прог1***, ***прог2*** и ***прог3***) запускаются на трёх разных ЭВМ ***a***, ***b*** и ***c***, а для взаимодействия между ними обычно используется сеть TCP/IP.

Утилита ***mpirun*** имеет множество ключей и вариантов запуска. Наиболее часто, используются следующие:

- **--hostfile файл** — задаёт файл, в котором перечисляются имена хостов, требуемое количество используемых процессов, а также максимальное количество возможных процессоров;
- **--host список** — задаёт имя одного или список имён хостов, перечисленных через запятую, на которых должны запускаться процессы;
- **--np** (или **-np**) **число** — определяет старт требуемого количества процессов;
- **--mca** (или **-mca**) **список_MCA_parameters** — список параметров, уточняющих средства коммуникации запущенных процессов;
- **--wdir <directory>** - устанавливает рабочую директорию, в которой стартуют процессы. Если параметр не задан, то подразумевается текущая директория (или используется значение системной переменной ***$HOME***);
- **-x <env-variable-name>** - Установка и экспорт имени и значения системной переменной, которая будет использоваться при запуске параллельного приложения. Ключ (опция) **-x** может быть задана множество раз.

Примечание — Поскольку, имеется множество способов запуска утилиты ***mpirun***, а также множество настроек самого пакета ***Open MPI***, то следует воспользоваться перечнем ответов на вопросы, которые подробно, но в англоязычном варианте, размещены на сайте: https://www.open-mpi.org/faq/

Учебный кластер ЭВМ.

Из материала предыдущих разделов видно, что технология MPI, и в частности, технология Open MPI, - достаточно сложны для изучения.

С другой стороны, очевидно, что технология имеет достаточно большой потенциал, который можно использовать в различных прикладных проектах, требующих повышенного вычислительного ресурса.

Для целей учебного процесса, используется задание, которое имеет название «Учебный кластер ЭВМ».

Чтобы максимально упростить задание, кластер ЭВМ будет состоять из двух ЭВМ, объединённых сетью в пределах учебного класса кафедры АСУ. Для выполнения лабораторной работы, студенты разделяются на проектные группы по два человека и занимают два отдельных компьютера, расположенного в одном учебном классе.

В процессе выполнения данной лабораторной работы, каждая учебная проектная группа последовательно решает следующие задачи:

1) настройка сети на машинах кластера;
2) настройка и тестирование работы пакета OpenSSH на отдельном компьютере;
3) создание и распространение ключей SSH;
4) тестирование ПО кластера.

Примечание — Поскольку компьютеры учебных классов кафедры АСУ используются для разных целей обучения, следует очень внимательно подойти к настройкам сети, чтобы в дальнейшем избежать проблем с выполнением данной работы.

Далее, по тексту данного методического пособия, излагается учебный материал по выполнению каждого пункта проекта.

Настройка сети.

Этап 1. Проверка и настройка подключения к сети. Запустить виртуальный терминал и выполнить в нем команду согласно выражения (3.16), которая обеспечит запуск файлового менеджера «***Midnight Commander***» от имени пользователя ***root***.

sudo mc **(3.15)**

Затем, командой (3/16) убедиться в наличии или отсутствии настроек компьютера на работу в сети.

ip address **(3.16)**

Если ***IP-адрес*** отсутствует, то перейти в директорию ***/etc/netctl*** и убедиться в наличии файла ***ethernet-dhcp***. Если файл ***ethernet-dhcp*** — отсутствует, то скопировать его шаблон из директории ***/etc/netctl/exemples***, после чего откорректировать в нем параметр *Interface*, который должен быть равен имени интерфейса сетевого устройства, полученного при выводе предыдущей команды.

Далее, используя типовую последовательностью команд (3.17), перезапустить сетевой интерфейс и убедиться в подключении компьютера к сети.

```
netctl stop    ethernet-dhcp
netctl disable ethernet-dhcp
netctl enable  ethernet-dhcp
netctl start   ethernet-dhcp
```
(3.17)

Этап 2. Установка уникального имени компьютера. Выполнить в терминале команду (3.18), читающую сетевое имя компьютера.

hostname **(3.18)**

Убедиться, что это имя является уникальным и больше не повторяется в именах компьютеров сети.

Примечание - Общее правило, обеспечивающее уникальность имени компьютера в сети, — использование имени, под которым студент проводит личный логин на кафедре АСУ.

Чтобы установить уникальное имя компьютера, следует:

- *подключить* к ЭВМ личный flashUSB студента;
- *отредактировать* файл ***/boot/grub/grub.cfg***, установив в первом пункте меню значение переменной ***UPK_HOST=имя-компьютера***;
- *завершить* (корректно) работу с ЭВМ;
- *включить* ЭВМ и *снова войти* в учебную систему.

Примечание — Нужно проверить настройки компьютера по пунктам этапов 1 и 2.

Этап 3. Настройка файла /*etc/hosts*. Завершив первые два этапа, следует:

1) *передать напарнику* по лабораторной работе свои данные об ***IP***-адресе и имени хоста (компьютера);
2) *получить от напарника* его данные об ***IP***-адресе и имени хоста (сетевые данные удалённого хоста).

Отредактировать файл ***/etc/hosts***, добавив в него две строки:

1) ваш_IP_адрес имя_вашего_хоста
2) его_IP_адрес имя_его_хоста

После указанных действий, проверить взаимосвязь между компьютерами кластера командой (3.19).

ping имя_его_хоста **(3.19)**

В случае успешной проверки, настройка сети — закончена.

Настройка и тестирование OpenSSH.

Технология ***Open MPI*** широко использует технологию ***OpenSSH*** для связи и запуска процессов на компьютерах кластера, которая обеспечивается запуском и настройкой серверного процесса ***sshd***.

ОС УПК АСУ содержит все необходимое программное обеспечение для работы с пакетом ***OpenSSH***, но первоначальная ее настройка не обеспечивает запуск ***sshd***, поэтому такую настройку следует осуществить.

Примечание — ***Неправильная настройка*** и запуск сервера ***sshd*** приводит к нарушению безопасности компьютера.

После запуска ЭВМ, от имени пользователя ***root***, следует выполнить следующие две команды (3.20) и (3.21), которые отключат работу сервера.

systemctl disable sshd.service **(3.20)**
systemctl stop sshd.service **(3.21)**

Далее работу с сервером ***sshd*** следует осуществлять в ограниченном объёме, как это используется в данной лабораторной работе.

Этап 4. Настройка серверной части технологии OpenSSH.

Запустив терминал и отключив работу сервера ***sshd***, следует перейти в директорию */**etc/ssh***, где следует удалить все файлы, кроме: *moduli*, *ssh_config* и *sshd_config*.

Необходимо также отредактировать файл ***ssh_config***, установив значения параметров, как это показано на листинге 3.8, закомментировав все остальные параметры.

Листинг 3.8 — Правильные параметры файла ssh_config

```
Host *
PasswordAuthentication no
ChallengeResponseAuthentication no
PubkeyAuthentication yes
IdentityFile ~/.ssh/id_rsa
Port 22
Protocol 2
```

Отредактировать файл ***sshd_config***, установив значения параметров, как это показано на листинге 3.9, закомментировав все остальные параметры.

Листинг 3.9 — Правильные параметры файла sshd_config

```
AllowUsers asu upk
Port 22
HostKey /etc/ssh/ssh_host_rsa_key
PermitRootLogin yes
PubkeyAuthentication yes
AuthorizedKeyFile .ssh/authorized_keys
PasswordAuthentication no
```

```
PermitEmptyPassords yes
ChallengeResponseAuthentication no
Subsystem sftp /usr/lib/ssh/sftp-server
```

Запустить сервер ***sshd*** командой (3.22).

systemctl start sshd.service **(3.22)**

Убедиться командой (3.23), что сервер стартовал.

ps -ax | grep sshd **(3.23)**

Примечание — Перед стартом сервера ***sshd***, в директории ***/etc/ssh***, будут созданы все необходимые закрытые и открытые ключи сервера. Предполагается, что удалённое подключение к ***sshd*** будет проводиться без пароля с использованием сертификатов ключей.

Этап 5. Локальная проверка ssh-соединений.

Прежде чем проверять работу пакета ***OpenSSH*** в сети, это следует сделать на локальном компьютере. Проверка будет осуществляться посредством подключения пользователя ***upk*** к пользователю ***asu***.

Для этого пользователем ***upk*** следует открыть терминал, в котором выполнить команду (3.24).

ssh-keygen -t rsa **(3.24)**

Примечание — При генерации ключей, утилита ***ssh-keygen*** запросит ввод парольных фраз. На такие запросы необходимо просто нажимать клавишу <***Enter***>.

В результате указанного действия появится сообщение, как это показано на рисунке 3.6, а в директории ***~/.ssh*** пользователя ***upk***, появятся два файла:

- **id_rsa** — *закрытый ключ* по сертификату *rsa*;
- **id_rsa.pub** — *открытый ключ* по сертификату *rsa*.

Далее открытый ключ, под именем ***aa.pub***, следует скопировать в директорию ***/tmp***. Переключиться на виртуальный терминал ***/dev/tty3*** и произвести ***login*** пользователем ***asu***, после чего запустить ***mc***.

Пользователем ***asu*** скопировать файл ***/tmp/aa.pub*** в директорию ***~/.ssh*** и в этой директории выполнить команду (3.25), после чего удалить файл ***aa.pub***.

cat ./aa.pub >> authorized_keys **(3.25)**

В результате указанных действий, пользователь ***asu*** будет готов к подключению по ***ssh***.

Необходимо вернуться на ***/dev/tty1*** и удалить файл ***/tmp/aa.pub***.

Теперь можно провести проверку локального соединения.

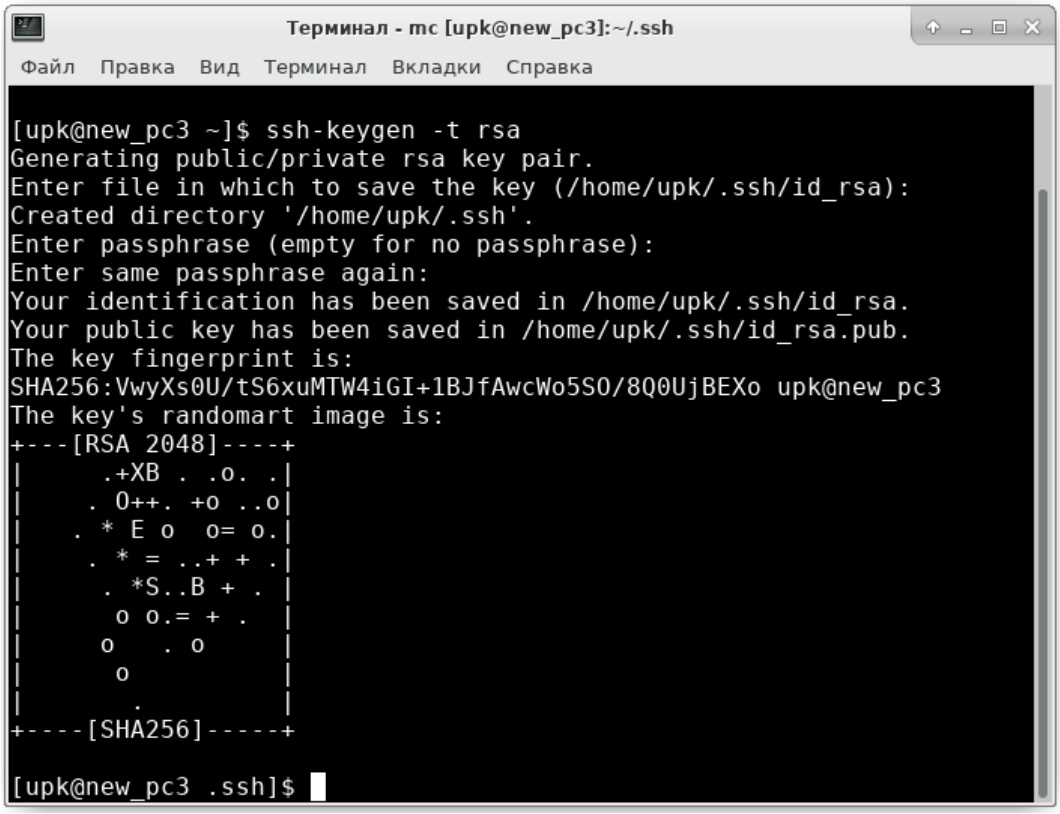

Рисунок 3.6 — Результат генерации ключей утилитой ssh-keygen

В виртуальном терминале, от имени пользователя ***upk***, выполнить команду (3.26).

ssh asu@имя_локального_компьютера **(3.26)**

Первоначально сервер спросит: *доверяем ли мы его сертификату?*

Необходимо ответить: ***yes***

В результате:

1) пользователь ***upk*** должен зайти на компьютер пользователем ***asu***;
2) в файл ***~/.ssh/known_hosts*** пользователя ***upk*** будет записан сертификат доверия указанному серверу, что в дальнейшем обеспечит полный беспарольный доступ к серверу для подключения пользователем ***asu***.

Чтобы убедиться в этом, следует выйти из логин пользователя ***asu*** и повторить соединение.

Этап 6. Распространение публичных ключей.

Убедившись в осуществимости локального защищённого соединения по протоколу ***ssh***, следует:

1) *передать* напарнику свой файл ***id_rsa.pub***;
2) *получить* от напарника его файл ***id_rsa.pub***, который необходимо скопировать в директорию ***/home/upk/.ssh*** под любыть именем, например, ***bb.pub***;
3) *выполнить* (от имени пользователя ***upk***) в директории команду (3.27);
4) *удалить* файл ***bb.pub***.

cat ./bb.pub >> authorized_keys **(3.27)**

Этап 7. Проверка защищённого удалённого соединения.

После успешного проведения предыдущих действий, следует провести соединение с компьютером напарника, выполнив команду (3.28).

ssh upk@имя_удаленного_компьютера **(3.28)**

Примечание - При первом соединении, обязательно согласие ***yes*** на запрос о доверии сертификату сервера.

Тестирование ПО кластера.

Тестирование ПО кластера проводится только после успешного выполнения всех предыдущих этапов. Обычно тестирование ПО кластера осуществляется в два этапа:

1) *упрощенное тестирование*, предполагающее запуск на удалённом компьютере общедоступной утилиты, не использующей технологии ***Open MPI***;
2) *полное тестирование*, предполагающее запуск на удалённом компьютере программ, разработанных по технологии ***Open MPI*** и включающих, при необходимости, дополнительную настройку системной среды ОС ЭВМ.

Мы проведём только упрощённое тестирование.

Этап 8. Упрощённое тестирование ПО кластера.

Для упрощённого тестирования следует:

1) запустить на локальной и удалённой ЭВМ утилиту ***hostname***, распечатывающую на терминале имя компьютера, на котором она запущена;
2) выполнить команду (3.29).

mpirun -np 1 —host имя_local hostname : -np 1 —host имя_remote hostname **(3.29)**

После успешного тестирования, следует занести результаты исследования в личный отчёт и завершить выполнение лабораторной работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Резник, В.Г. Архитектура вычислительных комплексов. Методические указания по самостоятельной и индивидуальной работе студентов. Учебно-методическое пособие [Электронный ресурс] / В.Г. Резник. – Томск, ТУСУР, 2017. – 13 с. — Режим доступа: https://asu.tusur.ru/learning/090401p/d03/090401p-d03-work.pdf (дата обращения: 28.09.2023).
2. Резник, В.Г. Архитектура вычислительных комплексов. Тема 1. Состояние и тенденции развития АВК. Учебно-методическое пособие [Электронный ресурс] / В.Г. Резник. – Томск, ТУСУР, 2017. – 49 с. — Режим доступа: https://asu.tusur.ru/learning/090401p/d03/090401p-d03-theme1.pdf (дата обращения: 28.09.2023).
3. Резник, В.Г. Архитектура вычислительных комплексов. Тема 2. Архитектура процессоров. Учебно-методическое пособие [Электронный ресурс] / В.Г. Резник. – Томск, ТУСУР, 2017. – 108 с. — Режим доступа: https://asu.tusur.ru/learning/090401p/d03/090401p-d03-theme2.pdf (дата обращения: 28.09.2023).
4. Резник, В.Г. Архитектура вычислительных комплексов. Тема 3. Архитектуры вычислительных комплексов. Учебно-методическое пособие [Электронный ресурс] / В.Г. Резник. – Томск, ТУСУР, 2017. – 64 с. — Режим доступа: https://asu.tusur.ru/learning/090401p/d03/090401p-d03-theme3.pdf (дата обращения: 28.09.2023).
5. Резник, В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux: Учебно-методическое пособие для студентов направления 09.03.01, Направление подготовки "Программное обеспечение средств вычислительной техники и автоматизированных систем" [Электронный ресурс] / В.Г. Резник. — Томск: ТУСУР, 2016. — 33 с. — Режим доступа: https://edu.tusur.ru/publications/6238 (дата обращения: 28.09.2023).