Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Курсовой проект

по дисциплине: «Проектирование операционных систем»

Тема: «Разработка мобильного оконного менеджера»

Работу выполнил студент

13541/4 *Абдуллин А. М.*

Преподаватель

_____ *Душутина Е.В.*

Санкт-Петербург
2017

Оглавление

1 Цель работы

Целью данной работы является разработка мобильного оконного менеджера. Оконный менеджер должен иметь строку состояния и рабочий стол, с которого можно запускать остальные приложения.

2 Описание задачи

Данная курсовая работа выполнена в раках проекта по разработке мобильного устройства на платформе Raspberry Pi Zero [1]. Данный проект включает в себя несколько задач:

- разработка аппаратной платформы мобильного устройства на основе Raspberry Pi Zero — подбор необходимых компонентов мобильного устройства (GSM модуль, динамик, микрофон, аккумулятор и т.д.) и их размещение на плате устройства
- установка и конфигурирование ОС для Raspberry Pi
- разработка стека драйверов для комплектующих
- разработка сервисного слоя (в виде демонов UNIX), который будет предоставлять необходимую информацию клиентским приложениям
- **разработка мобильного оконного менеджера**, который позволит запускать и отображать на экране графические пользовательские приложения
- разработка клиентских приложений (для осуществления звонков, настроек и т.д.)

Архитектура разрабатываемого проекта приведена на рисунке 1.

Таким образом, конечной целью разработки оконного менеджера (ОМ) является его запуск на Raspberry Pi Zero. Необходимо учесть, что Raspberry обладает малой вычислительной мощностью, поэтому ОМ должен быть реализован как можно более оптимальным образом.

Мобильный ОМ должен так же иметь два встроенных системных приложения:

- строка состояния, которая отображает информацию об устройстве (текущее время, уровень заряда, уровень сигнала и т.д.). Строка состояния всегда отображается в верхней части экрана
- рабочий стол, на котором расположены иконки запуска приложений, установленных в системе. Рабочий стол отображается на всю часть экрана, не занятую строкой состояния

Данные системные приложения не являются непосредственной частью ОМ. ОМ автоматически запускает их при своем запуске, и запоминает их идентификаторы для соответствующего отображения. ОМ должен иметь возможности указания путей к приложениям строки состояния и рабочего стола (т.е. должен иметь возможность конфигурирования).

При запуске несистемных приложений ОМ скрывает рабочий стол, выводит запущенное приложение на передний план и делает его активным. ОМ должен поддерживать три типа окон:

- обычные окна приложений, которые отображаются во весь экран

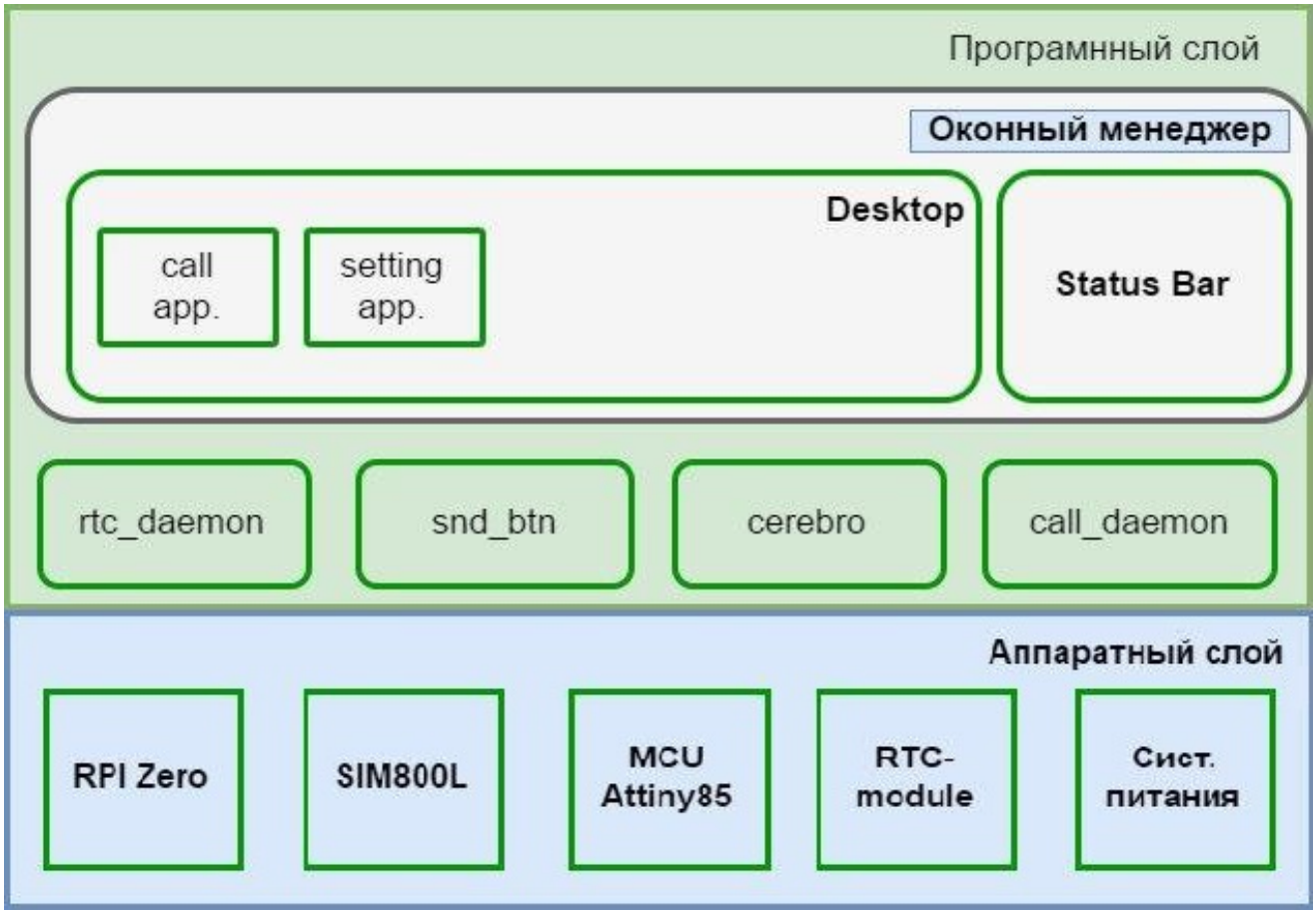

Рис. 1: Архитектура проекта

- всплывающие уведомления (например, контекстное меню при нажатии правой кнопки мыши), которые отображаются в соответствии со своими размерами в указанной точке. Например, при нажатии на правую кнопку мыши открывается контекстное меню в точке нажатия с необходимыми размерами
- окна меню (например стандартные меню типа ”Файл”и т.д. в верхней части приложений). Должны запускаться в соответствии со своими размерами и отрисовываться начиная с нажатой кнопки.

Созданный оконный менеджер так же должен иметь возможность обработки управляющих комбинаций клавиш для:

- закрытия приложения
- завершения работы ОМ
- перелистывания окон
- запуска терминала

Так же ОМ должен иметь возможности перемещения и изменения размеров окон.

3 Теоретические сведения

3.1 Протокол X и X Window System

X Window System — оконная система, обеспечивающая стандартные инструменты и протоколы для построения графического интерфейса пользователя, используется в UNIX-подобных ОС. X Window System обеспечивает базовые функции графической среды: отрисовку и перемещение окон на экране, взаимодействие с устройствами ввода, такими как, например, мышь и клавиатура. X Window System не определяет деталей интерфейса пользователя — этим занимаются оконные менеджеры. В X Window System предусмотрена сетевая прозрачность: графические приложения могут выполняться на другой машине в сети, а их интерфейс при этом будет передаваться по сети и отображаться на локальной машине пользователя. Архитектура протокола X приведена на рисунке 2.

X использует модель клиент-сервер: сервер X взаимодействует с различными клиентскими программами. Сервер принимает запросы на графический вывод (окна) и отправляет обратно пользовательский ввод (с клавиатуры, мыши или сенсорного экрана). Сервер может работать как:

- Приложение, отображающее окно другой системы отображения
- Системная программа, управляющая видеовыходом ПК
- Выделенный набор аппаратных средств

X рассматривает перспективу приложения, а не конечного пользователя: X предоставляет приложениям дисплей для отображения и пользовательский ввод-вывод приложениям, поэтому он является сервером; приложения используют эти службы, поэтому они являются клиентами.

Протокол связи между сервером и клиентом работает прозрачно: клиент и сервер могут работать на одном компьютере или на разных устройствах, возможно, с разными архитектурами и операционными системами. Клиент и сервер могут даже безопасно обмениваться данными через Интернет путем туннелирования соединения по зашифрованному сетевому сеансу. Сам клиент X может эмулировать X-сервер, предоставляя услуги отображения другим клиентам. Это называется ”X-вложенность”. Клиенты с открытым исходным кодом, такие как Xnest и Xephyr, поддерживают такую вложенность X.

Протокол X в первую очередь определяет примитивы протокола и графики — он преднамеренно не содержит спецификаций для дизайна пользовательского интерфейса приложения. Вместо этого прикладные программы — такие как оконные менеджеры, инструментальные средства GUI-виджета и среды рабочего стола или графические пользовательские интерфейсы приложений — определяют и предоставляют такие детали. В результате нет типичного интерфейса X, и несколько различных сред настольных компьютеров стали популярными среди пользователей. Оконный менеджер контролирует размещение и внешний вид окон приложений. Оконные менеджеры различаются по сложности и объемноси от самых простых (например, twm, основной оконный менеджер, поставляемый с X, или evilwm, чрезвычайно легкий оконный менеджер) в более комплексные среды рабочего стола, такие как Enlightenment. Основной идеей оконного менеджера в протоколе X является то, что OM не является непосредственной частью сервера, это клиентское приложение с некоторыми дополнительными возможностями. Для стандартизации протокола общения между OM и остальными клиентами, был разработан протокол ICCCM (Inter-Client Communication Conventions Manual) [2], X Window System, обеспечивающий интероперабельность X-клиентов в пределах одного и того же X-сервера.

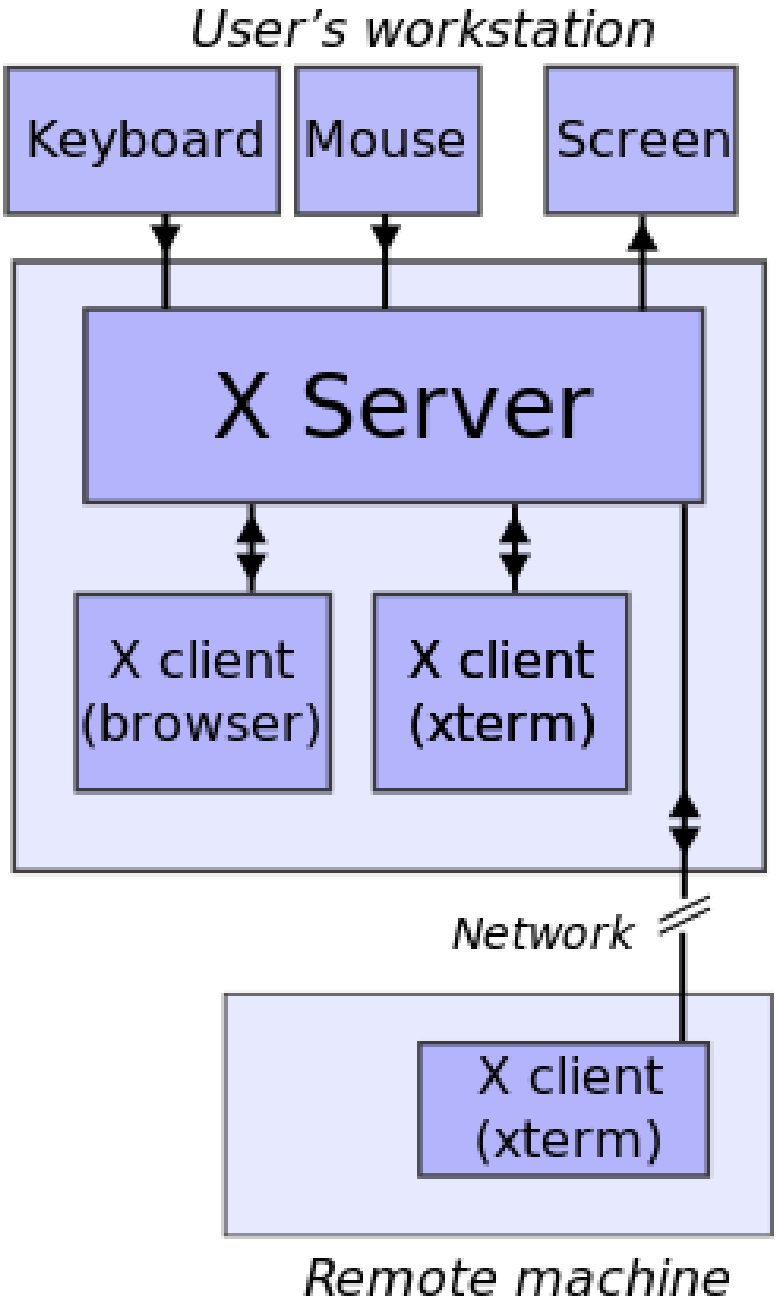

Рис. 2: Архитектура протокола X

Когда оконный менеджер запущен, некоторые виды взаимодействия между X-сервером и его клиентами перенаправляются через оконный менеджер. В частности, всякий раз, когда делается попытка показать новое окно, этот запрос перенаправляется в ОМ, который определяет начальную позицию окна. Кроме того, большинство современных оконных менеджеров являются репарентирующими, что обычно приводит к размещению баннера в верхней части окна и оформлению декоративной рамки вокруг окна. Эти два элемента управляются оконным менеджером, а не программой. Поэтому, когда пользователь нажимает или перетаскивает эти элементы, именно оконный менеджер выполняет соответствующие действия (например, перемещение или изменение размера окна).

Менеджеры окон также отвечают за конки приложений. Когда пользователь запрашивает окно для его изменения, ОМ удаляет его (делает его невидимым) и предпринимает соответствующие действия, чтобы показать иконку на своем месте.

Хотя главной задачей ОМ является управление окнами, многие оконные менеджеры имеют дополнительные функции, такие как обработка щелчков мыши в корневом окне, представление панелей и других визуальных элементов, обработка некоторых нажатий клавиш (например, Alt-F4 может закрыть окно).

X-сервер состоит из набора расширений, каждое из которых реализует определённые функции: от прорисовки геометрических примитивов до ускорения обработки и вывода на экран трёхмерной графики с использованием возможностей видеоаппаратуры. Почти каждый из этих модулей можно отключить или настроить в конфигурационном файле.

3.2 Протокол Wayland

Wayland — это протокол организации графического сервера в UNIX-подобных ОС и его библиотечная реализация на языке C [3]. Так же протокол имеет свою референсную реализацию Weston [4]. Архитектура протокола Wayland приведена на рисунке 3.

Можно увидеть, что Wayland является клиент-серверным протоколом, в котором клиентами являются графические приложения, а сервером является композитор. Композитор — это дисплейный сервер, который взаимодействует с пользовательскими устройствами ввода-вывода, с аппаратурой компьютера и управляет потоками данных клиентских приложений. В конечном счете, графические приложения запрашивают у сервера отображения своих графических буферов на экране, а композитор контролирует отображение этих буферов на экране.

Реализация Wayland была разработана как двухслойный протокол (рис. 4):

- Низкоуровневый протокол, который управляет межпроцессным взаимодействием между процессами клиента и композитора. Данный слой реализован с использованием средств ярда UNIX.

- Поверх него построен высокоуровневый протокол, который обрабатывает информацию, которой должны обмениваться клиент и композитор для реализации базовый возможностей графической оболочки.

Реализация протокола Wayland разделена на две библиотеки: libwayland-client для клиентских приложений и libwayland-server для реализации композиторов.

Протокол Wayland описывается как ”асинхронный объектно-ориентированный протокол”. Объектно - ориентированность означает, что сервисы, предлагаемые композитором, представлены в виде набора объектов. Каждый объект реализует интерфейс, который имеет имя, ряд методов (называемых запросами), а также несколько связанных событий. Каждый запрос и событие имеет ноль или более аргументов, каждый из которых имеет имя и тип данных. Протокол является асинхронным в том смысле, что запросы не должны ждать синхронизированных ответов, избегая времени задержки на двустороннее сканирование. Таким образом достигается улучшенная производительность.

Клиенты Wayland могут сделать запрос (вызов метода) для некоторого объекта, если интерфейс объекта поддерживает этот запрос. Клиент также должен предоставить необходимые данные для аргументов такого запроса. Именно так клиенты запрашивают сервисы у композитора. Композитор, в свою очередь, отправляет информацию клиенту, вызывая генерацию необходимых событий объекта. Эти события могут быть вызваны композитором как ответ на определенный

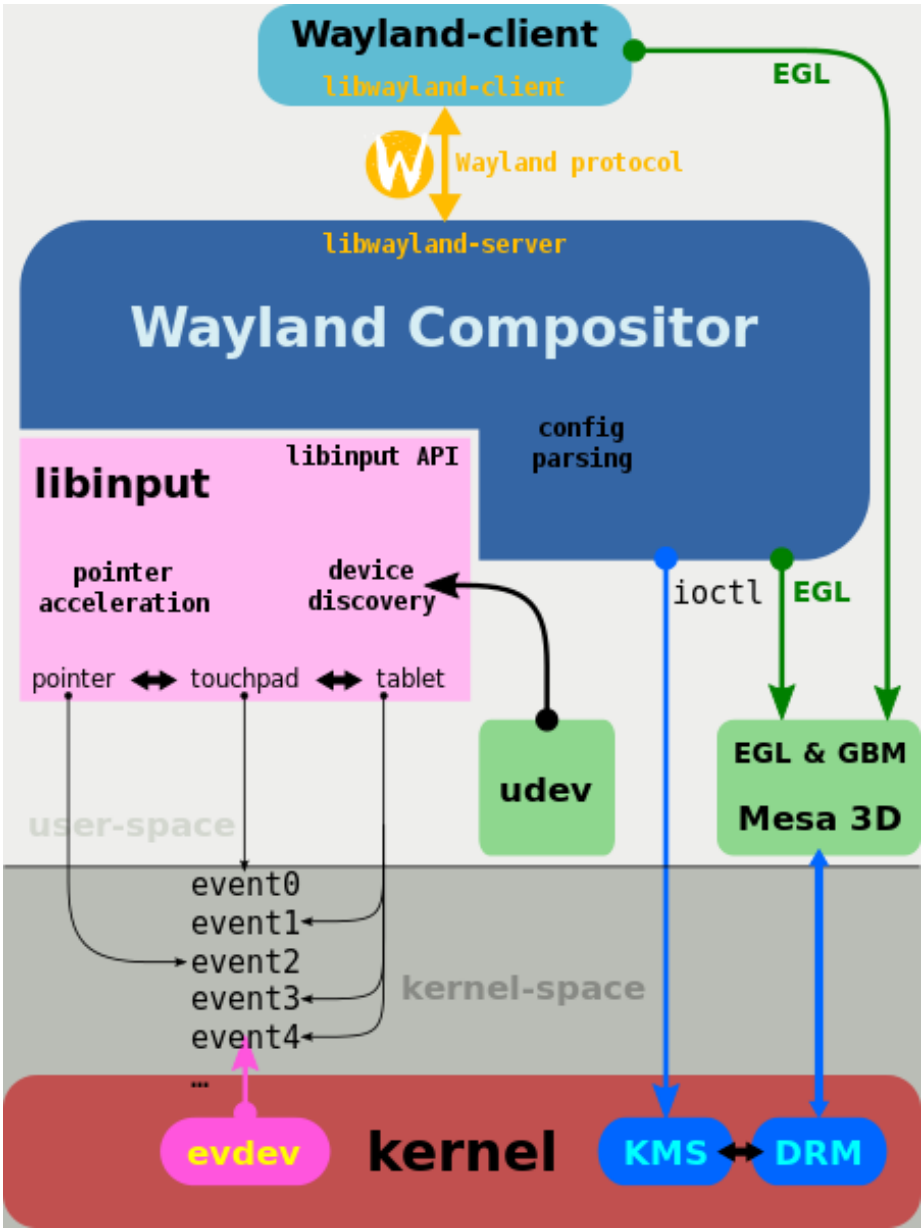

Рис. 3: Архитектура протокола Wayland

запрос или асинхронно, в ответ на возникновение каких-либо внутренних событий (например, активность одного из устройства ввода).

Для того, чтобы клиент мог сделать запрос к объекту, ему сначала нужно указать серверу идентификационный номер, который он будет использовать для идентификации этого объекта. В наборе данных есть два типа объектов: глобальные и локальные. Глобальные объекты указываются композитором клиентам, когда они созданы (а также когда они уничтожены), в то время как локальные объекты обычно создаются другими объектами, которые уже существуют как часть их функциональности. Интерфейсы, их запросы и события являются основными элементами, определяющими протокол Wayland. Каждая версия протокола включает в себя набор интерфейсов, а также их запросы и события, которые ожидаются в любом наборе Wayland. Опционально, Wayland композитор может определять и реализовывать свои собственные интерфейсы, которые поддерживают новые запросы и события, тем самым расширяя функциональность за пределами

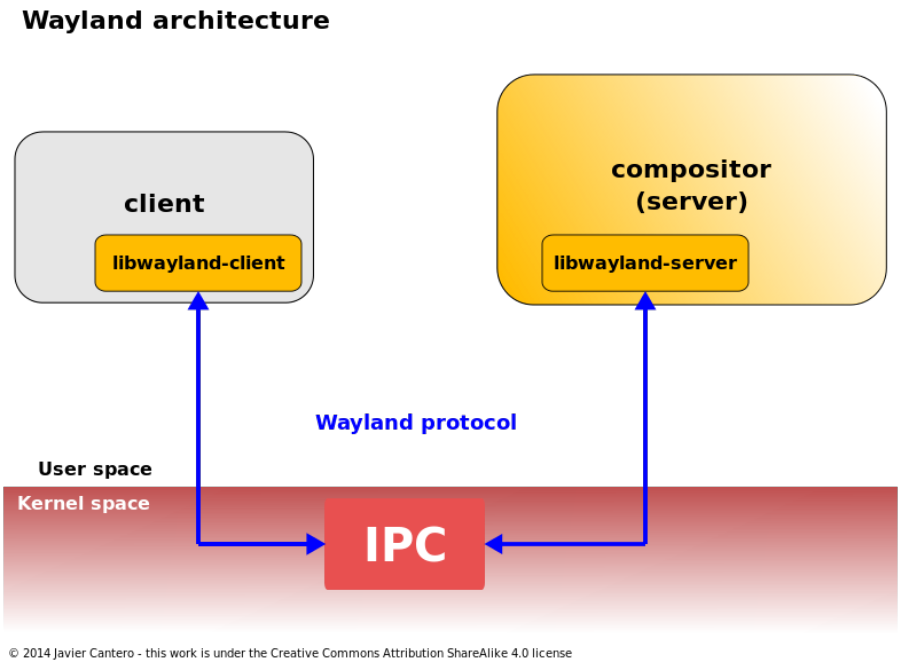

Рис. 4: Двухуровневая организация Wayland

основного протокола.

Интерфейсы текущей версии протокола Wayland определены в файле протокола protocol/wayland.xml исходного кода Wayland. Это XML файл, в котором перечислены существующие интерфейсы текущей версии, а также их запросы, события и другие атрибуты. Этот набор интерфейсов является минимальным, необходимым для любого композитора Wayland. Некоторые из основных интерфейсов протокола Wayland:

- wl_display — основной глобальный объект, специальный объект для инкапсуляции самого протокола Wayland
- wl_registry — глобальный объект реестра, в котором композитор регистрирует все глобальные объекты, которые он хочет предоставить клиентам
- wl_compositor — объект, который представляет собой композитор, и отвечает за объединение различных буферов в один вывод
- wl_surface — объект, представляющий прямоугольную область на экране, определяемую местоположением, размером и буфером пикселей
- wl_buffer — объект, который при подключении к объекту wl_surface предоставляет его отображаемое содержимое
- wl_output — объект, представляющий отображаемую область экрана
- wl_pointer, wl_keyboard, wl_touch — объекты, представляющие различные устройства ввода

Типичный сеанс клиента Wayland начинается с открытия соединения с композитором с использованием объекта wl_display. Это специальный локальный объект, который представляет

соединение и не живет на сервере. Используя его интерфейс, клиент может запросить глобальный объект wl_registry из композитора, где живут все глобальные имена объектов, и связывать те, что интересует клиента. Обычно клиент связывает по крайней мере объект wl_compositor, откуда он будет запрашивать один или несколько объектов wl_surface для отображения вывода приложения на дисплее.

Композитор Wayland может определять и экспортировать свои собственные дополнительные интерфейсы. Эта функция используется для расширения протокола за пределами базовых функций, предоставляемых основными интерфейсами, и стала стандартным способом реализации расширений протокола Wayland.

Для Wyaland так же существует расширение XWayland, которое позволяет запускать X-приложения в Wayland.

3.3 Сравнение X и Wayland

Существует несколько отличий между Wayland и X в отношении производительности, поддержки кода и безопасности:

- Архитектура: композитор — это отдельная дополнительная функция в X, а Wayland объединяет дисплейный сервер и композитор как единую функцию. Кроме того, он включает в себя некоторые из задач оконного менеджера, который в X является отдельным процессом на стороне клиента.

- Рендеринг: X-сервер по-умолчанию сам выполняет рендеринг окон. Существуют так же расширения позволяющие ему окно отрендеренное на стороне клиента. Напротив, Wayland по-умолчанию не предоставляет API для визуализации, а делегирует клиентам такие задачи (включая рендеринг шрифтов, виджетов и т. д.). Декорирование окон может выполняться на клиентской стороне (например, с помощью набора графических средств) или на стороне сервера (в функциональности композитора).

- Безопасность: Wayland изолирует входные и выходные данные каждого окна, обеспечивая конфиденциальность, целостность и доступность в обоих случаях; X не имеет этих важных функций безопасности. Кроме того, с подавляющим большинством кода, работающего на клиенте, меньше кода нужно запускать с правами root, что повышает безопасность.

- Межпроцессное взаимодействие: X-сервер предоставляет базовый метод обмена между X-клиентами, позднее расширенный протоколом ICCM. Это взаимодействие X клиент — X клиент используется менеджерами окон, для реализации X-сессий, функций drag-and-drop, а многих других функций. Основной протокол Wayland не поддерживает связь между wayland-клиентами вообще, и соответствующая функциональность (если необходимо) должна быть реализована в окружении рабочего стола (например, KDE или GNOME) или третьей стороной (например, используя IPC базовая операционной системы).

- Сетевое взаимодействие: X — это архитектура, изначально разработанная для работы по сети. Wayland не предлагает сетевой прозрачности сам по себе, однако, композитор может реализовать любой протокол удаленного рабочего стола для достижения удаленного отображения. Кроме того, существуют исследования потоковой передачи и сжатия изображений Wayland, которые обеспечивали бы доступ к буферу буфера удаленного доступа, подобный VNC.

На основе всех этих фактов можно сказать, что Wayland является более современной графической средой, в которой учтены и исправлены многие ошибки X. Благодаря этому, Wayland так же выигрывает у X в производительности. Wayland так же поддерживает X-приложения (через XWayland). Таким образом, можно сделать вывод, что наиболее оптимальным выбором для реализации своего мобильного оконного менеджера будет выбор Wayland.

4 Анализ существующих композиторов Wayland

4.1 Weston

Weston — это референсная реализация композитора Wayland. Weston поставляется с несколькими примерами клиентов, от простых, которые демонстрируют некоторые аспекты протокола для, до полных клиентов и упрощенного инструментария. Существует также полноценный эмулятор терминала (weston-terminal) и простая оболочка рабочего стола. Наконец, Weston также обеспечивает интеграцию с сервером Xorg и может разместить X-клиенты на рабочем столе Wayland и представляться им как OM X. Пример запуска Weston привежен на рисунке 5.

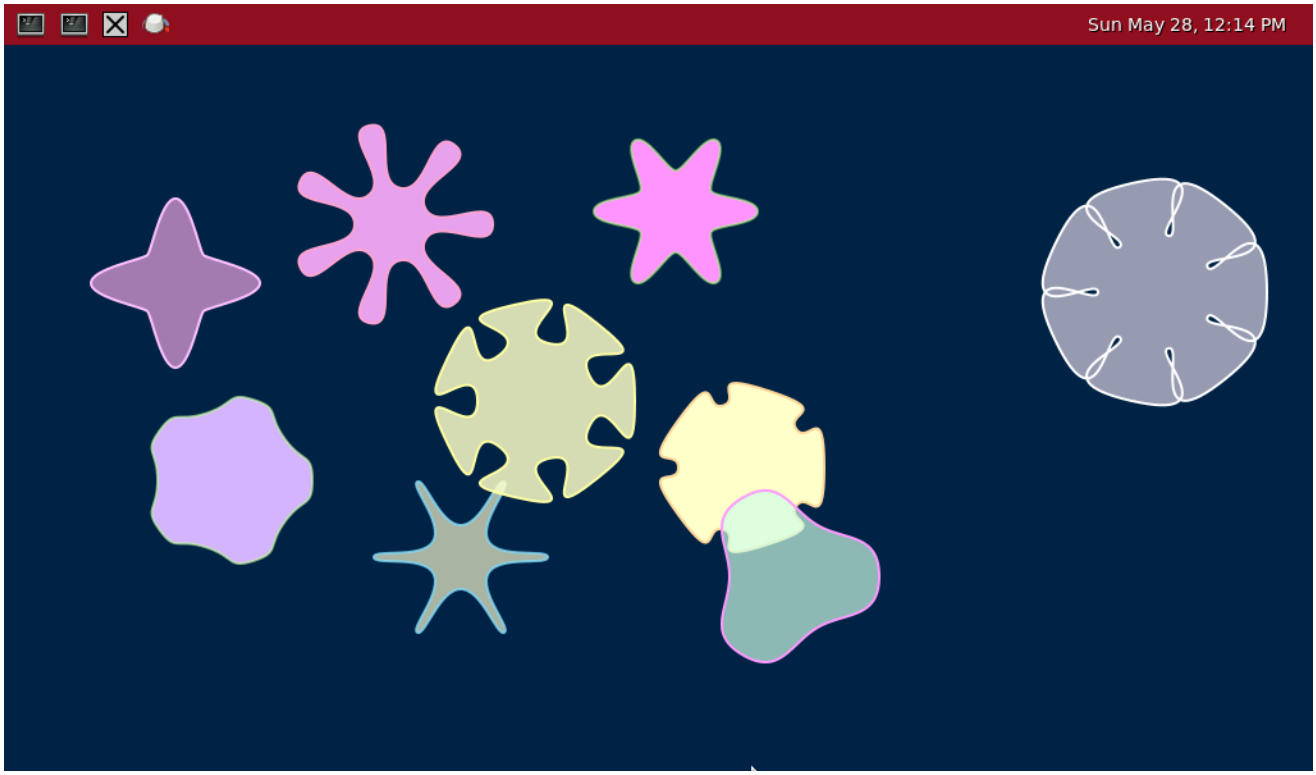

Рис. 5: Рабочий стол Weston

Weston по умолчанию поддерживает несколько бекендов, что позволяет ему без проблем запускаться на большом количестве платформ. До версии 1.9 Weston так же поддерживал бекенд для проприетарной графической системы Raspberry Pi, однако разработчики отказались от этой части из-за сложности поддержки кода.

Пакет Weston по умолчанию включает в себя библиотеку libweston. Libweston — это попытка отделить переиспользуемый исхдный код Weston в отдельную библиотеку. Эта библиотека

включает в себя корректную реализацию всех базовых протоколов Wayland и взаимодействие с подсистемами ввода вывода. Libweston предлагается использовать для облегчения разработки собственных ОМ.

Libweston впервые появился в версии Weston 1.12. В настоящее время библиотека находится в состоянии активной разработки. Разработчики утверждают, что API библиотеки не стабилен может сильно изменяться от версии к версии, поэтому в данный момент использование этой библиотеки не самое лучшее решение.

4.2 WLC

WLC — популярная библиотека-композитор для Wayland [5]. На основе этой библиотеки реализовано несколько оконных менеджеров и других библиотек. Их примеры:

- Тайлинговый оконный менеджер Sway (рис. 6)
- Модульный композитор orbment
- Модульный ОМ fireplace

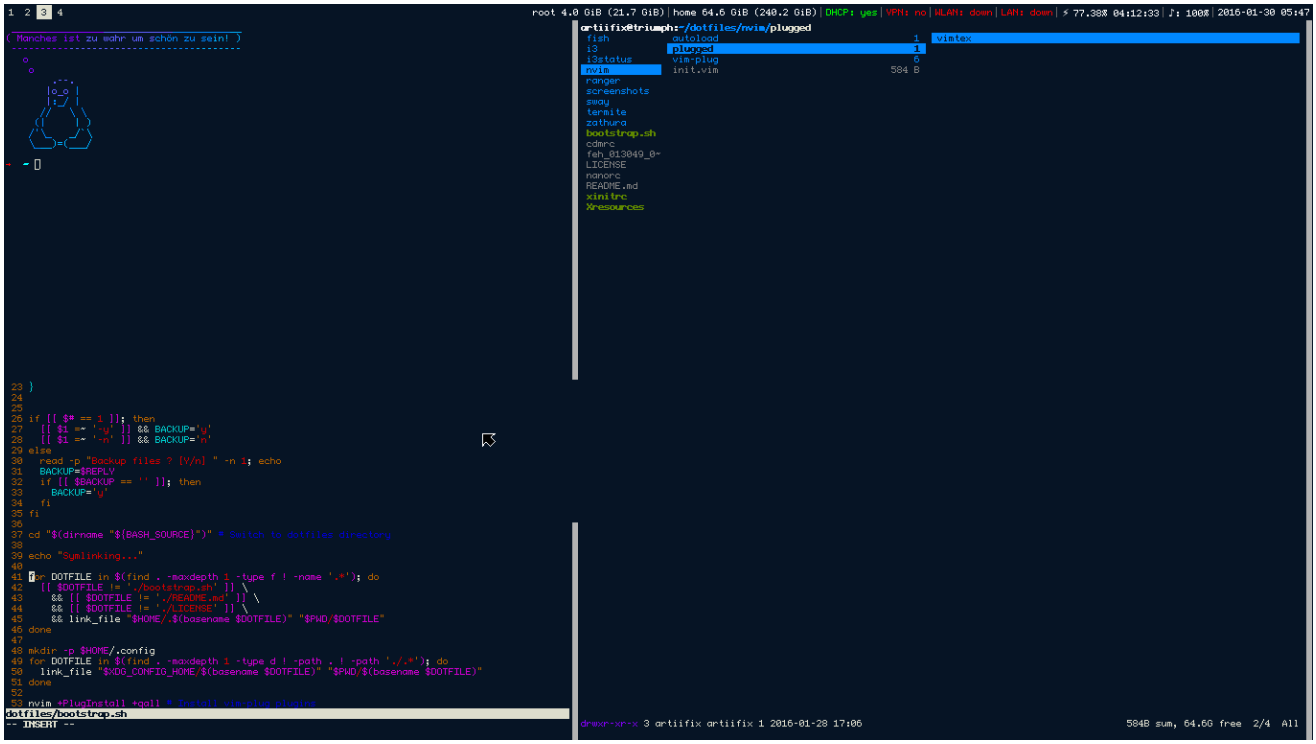

Рис. 6: Пример Sway

Библиотека обладает большой функциональностью. Она поддерживает различные бекенды, может запускаться как под X11, так и под Wayland. Библиотека очень удобна для разработки на ее основе собственного ОМ, так как она содержит примеры написания простейших ОМ.

4.3 SWC

SWC — это небольшой композитор Wayland, реализованный в виде библиотеки. Разработчик SWC утверждает, что swc был написан с целью предоставить минимальный набор функций для возможности отображения окон на экране.

На основе SWC реализован простой оконный менеджер Velox, предназначение которого — продемонстрировать возможности библиотеки.

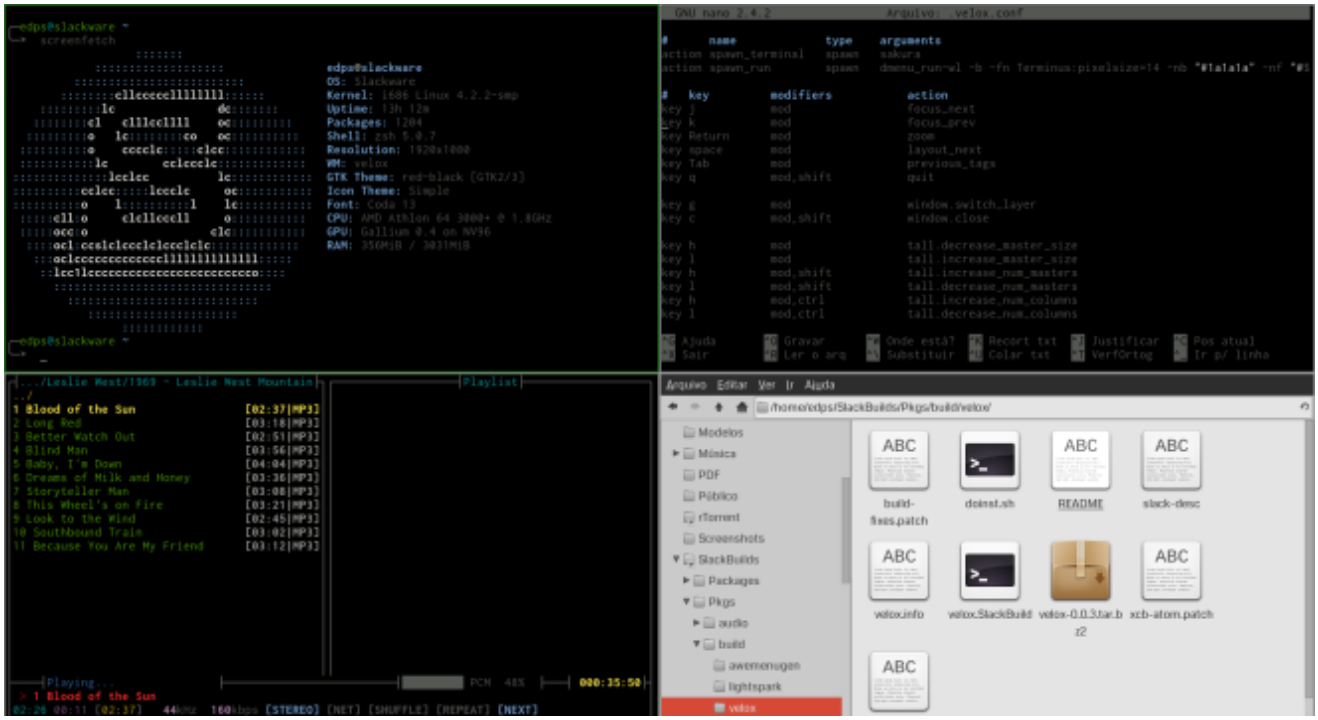

Рис. 7: OM Velox

5 Выполнение работы

5.1 Описание тестового стенда

Для выполнения работы использовалось два тестовых стенда:

- Платформа Raspberry Pi 1 с ОС ArchLinux
- Виртуальная машина с Arch Linux

Благодаря тому, что платформы Raspberry Pi 1 и Raspberry Pi Zero аппаратно полностью совместимы, ОМ разработанный на платформе RPi 1 будет совместим с RPi Zero. Однако, проблема в том, что RPi имеет проприетарную графику, которая не поддерживается в Wayland. Поэтому, задача конфигурирования ОС на платформе RPi включала в себя установку драйвера VC4 [6], который позволяет представить видеокарту RPi как стандартное DRI устройство.

Виртуальная машина с ArchLinux была установлена в гипервизоре VirtualBox. Однако, из-за того что в VirtualBox не реализована поддержка протокла Wayland, мобильный оконный менеджер запускался в качестве X-клиента в оконном менеджере xfce4.

5.2 Выбор библиотеки-композитора

Как было сказано ранее, в архитектуре Wayland оконный менеджер обязательно должен включать в себя композитор (в отличие от X). Так как реализация собственного Wayland-композитора с нуля — задача слишком объемная и трудоемкая, был решено использовать какую-либо библиотеку-композитор. В таком случае задачей разработанного ОМ будет управление окнами.

Из проанализированных библиотек было решено выбрать библиотеку wlc по нескольким причинам:

- API библиотеки устойчив и проверен
- на ее основе уже реализовано некоторое количество программ
- для библиотеки есть некоторое количество примеров

5.3 Разработка оконного менеджера

Исходный код основного файла оконного менеджера приведен в листинге 3. В основной функции приложения main производятся следующие действия:

- анализируются аргументы командной строки (строки 358-365)
- считывается конфигурация (строка 367)
- устанавливаются функции-обработчики действий для wlc (строки 370-392)
- инициализируется композитор (строки 395-397)
- запускаются системные приложения (строка состояния и рабочий стол) (строки 399-425)
- запускается оконный менеджер (строка 426)

Созданное приложение принимает один аргумент — файл конфигурации. Если аргумент не указан, ищется и открывается файл по-умолчанию – /.config/xxwm. Пример конфигурационного файла приведен в листинге 1. В нем указываются пути к исполняемым файлам строки состояния и рабочего стола.

Листинг 1: Формат конфигурационного файла ОМ

```
1 [statusbar]
2 exe=/home/kivi/workspace/Phone/src/status_bar/status
3
4 [desktop]
5 exe=/home/kivi/workspace/Phone/src/desktop/desktop
```

Считывание конфигурационного файла производится с помощью библиотеки inih. Исходные коды функций, которые производят считывание конфигурационного файла приведены в листингах 4 – 5.

Далее в функции main устанавливаются функции-обработчики для библиотеки-композитора. Данные функции обрабатывают события, получаемые от композитора и, соответственно, работают с типами данных композитора. Композитор определяет несколько абстракций:

- output — вся область отображения на экран. В терминах оконных менеджеров это соответствует ”рабочему столу”
- view — окно приложения

Рассмотрим все эти функции более подробно.

- wlc_log_set_handler устанавливает функцию, которая будет осуществлять логирование. В нашем случае устанавливается функция, которая просто выводит все сообщения в терминал с использованием функции printf (строчки 352-355).
- wlc_set_output_resolution_cb устанавливает функцию, которая обрабатывает изменение разрешения экрана. Устанавливаемая функция output_resolution просто вызывает функцию перерисовки окон relayout (строчки 174-177).
- wlc_set_view_created_cb устанавливает функцию, которая будет вызываться при создании нового окна. Устанавливаемая функция view_created устанавливает окну необходимые флаги, выносит окно на первый план, переключает фокус на это окно и вызывает функцию перерисовки (строки 180-192).
- wlc_set_view_destroyed_cb устанавливает функцию, которая будет вызваться при уничтожении окна. Устанавливаемая функция view_destroyed устанавливает фокус на самое верхнее окно и вызывает функцию перерисовки (строки 195-200).
- wlc_set_view_focus_cb устанавливает функцию, которая отвечает за установку фокуса на окно. Устанавливаемая функция view_focus устанавливает окну флаг WLC_BIT_ACTIATED (строчки 203-206).
- wlc_set_view_request_move_cb устанавливает функцию, которая отвечает за перемещение какого-либо окна по экрану. Устанавливаемая функция view_request_move вызывает функцию start_interactive_move (строчка 210), которая в свою очередь начинает интерактивное действие вызвав функцию start_interactive_action (строчка 42). Функция start_interactive_a сохраняет параметры окна, на котором начато интерактивное действие, в глобальную переменную и выводит это окно на первый план (строчки 26-38).
- wlc_set_view_request_resize_cb устанавливает функцию, которая отвечает за изменение размеров окна. Устанавливаемая функция view_request_resize начинает интерактивное действие изменения окна вызывая функцию start_interactive_resize (строчка 215). Данная функция начинает интерактивное действие вызвав функцию start_interactive_action, а затем определяет то, какую грань окна необходимо перемещать (строчки 46-66). Так же данная функция устанавливает окну флаг WLC_BIT_RESIZING, который указывает на то, что окно в текущий момент меняет свой размер.
- wlc_set_view_request_geometry_cb устанавливает функцию, которая отвечает за установку указанному окну определенных размеров. Устанавливаемая функция view_request_geometry не делает ничего, так как OM не предполагает возможности изменять размер окна извне.
- wlc_set_keyboard_key_cb устанавливает функцию-обработчик нажатий клавиатуры. Устанавливаемая функция keyboard_key считывает код нажатой клавиши, флаги модификаторов (CTRL, ALT и т.д.) и обрабатывает следующие комбинации (строчки 225-266):

  - CTRL+q — закрытие активного окна (если это не системное приложение)
  - CTRL+стрелка вниз — переключиться на следующее окно (аналог ALT+Tab в Windows)
  - CTRL+Escape — завершить работу оконного менеджера
  - CTRL+Enter — запустить терминал
- wlc_set_pointer_button_cb устанавливает функцию-обработчик нажатий кнопок мыши. Устанавливаемая функция pointer_button обрабатывает следующие комбинации (строчки 268-291):
  - CTRL+ЛКМ — переместить окно
  - CTRL+ПКМ — изменить размеры окна
- wlc_set_pointer_motion_cb устанавливает функцию-обработчик передвижения мыши. Устанавливаемая функция pointer_motion проверяет, если в данный момент выполняется интерактивное действие, она соответствующим образом изменяет отображение активного окна (передвигает или изменяет размеры, в зависимости от выполняемого действия) (строчки 294-350).

Далее в функции main выполняется запуск системных приложений (строки состояния и рабочего стола) и запуск самого композитора. При этом, ОМ запоминает PID системных приложений для возможности их идентификации. Например, на основе этих PID ОМ решает можно ли закрывать соответствующее окно.

Одной из самых главных функций ОМ является функция перерисовки окно relayout (строки 90-171). Данная функция действует по следующему алгоритму:

1. берет самое верхнее (переднее) окно
2. проверяем, является ли окно окном строки состояния
3. если да, то запоминаем идентификатор окна
4. если нет, то рисуем данное окно на весь экран, кроме верхней строчки высотой в 30 пикселей
5. обновляем окно строки состояния перерисовывая ее в верхних 30 пикселях экрана

Данный алгоритм позволяет каждый раз перерисовывать максимум два окно: активное окно приложения и окно строки состояния. Строку состояние необходимо перерисовывать, потому что в какой-то момент времени могло изменится разрешение экрана. Данный алгоритм позволяет снизить вычислительную нагрузку ОМ на систему.

Так же при перерисовке окна учитываются флаги типа окна. Библиотека WLC определяет пять флагов окна. Экспериментальным путем было выяснено, что при работе ОМ появляются и должны по-особому отображаться только два типа окон:

- WLC_BIT_UNMANAGED — окна меню
- WLC_BIT_POPUP — уведомления и контекстные меню (вызываемые при нажатии ПКМ)

Данные типы окон отображаются по-особому. Окна меню отображаются в соответствии с изначально заданными им параметрами, ничего не изменяется. Окна контекстных меню смещаются относительно координат их окна-родителя и координат нажатия мыши. Остальные окна отображаются по-умолчанию на всю область экрана, незанятую строкой состояния.

Примеры работы оконного менеджера приведены на рисунках 8 – 11.

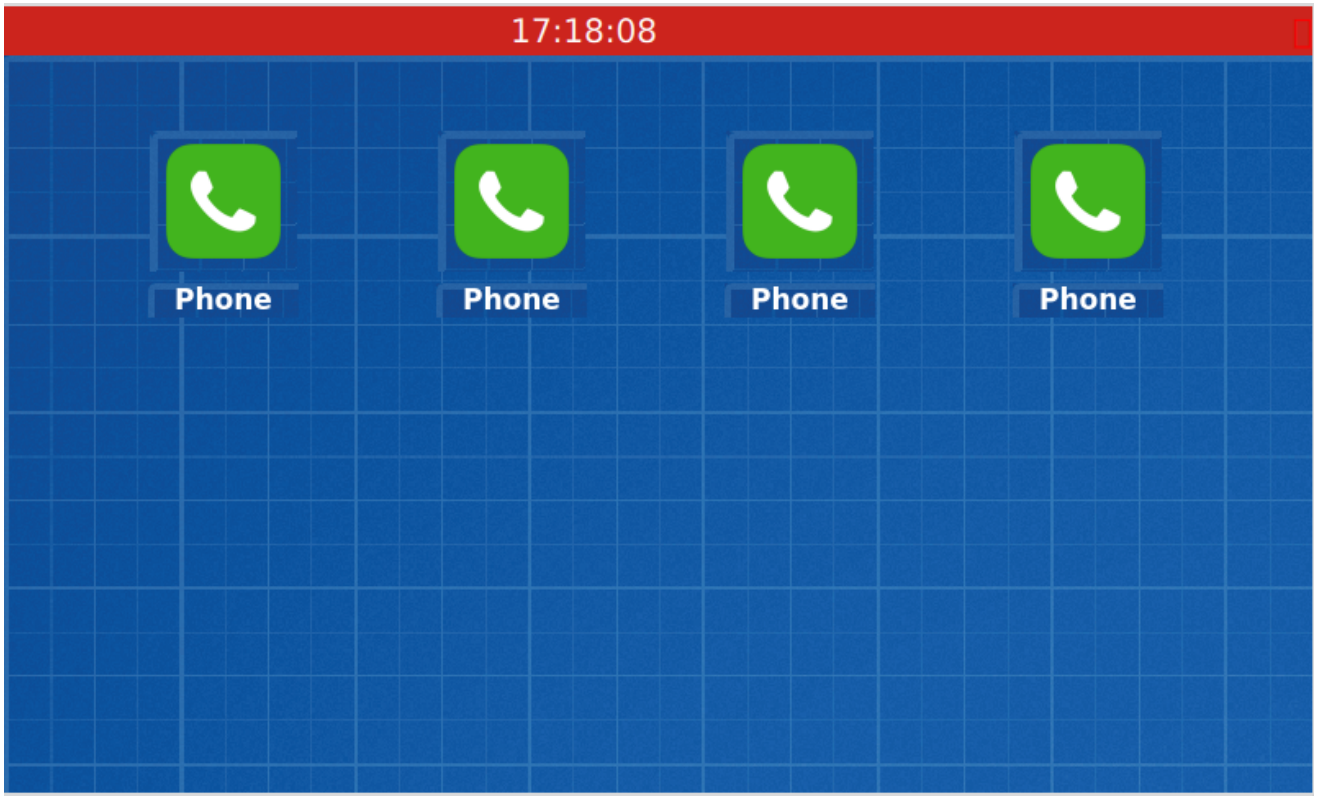

Рис. 8: Оконный менеджер

5.4 Добавление ОМ в экранный менеджер

Экранный менеджер или менеджер входа — графический экран, который отображается в конце процесса загрузки вместо стандартного приглашения командной строки. Экранный менеджер представляет собой экран ввода имени пользователя и пароля для входа в систему. Существует большое количество экранных менеджеров, однако все они детектируют установленные в систему оконные менеджеры по конфигурационным файлам формата .desktop. Данные файлы являются неким подобием ярлыков в Windows. .desktop — это сандартный для Linux конфигурационный файл. Подробное описание формата файлов .desktop приведено в [7]. Для созданного ОМ был создан минимальный файл .desktop (листинг 2).

Листинг 2: Файл .desktop для ОМ

```
1 [Desktop Entry]
2 Name=XXwm
3 Comment=Mobile Wayland window manager
4 Exec=/home/kivi/workspace/XXwm/xxonwm
5 Type=Application
```

Для того, чтобы экранный менеджер смог обнаружить ОМ необходимо поместить .desktop в каталог /usr/share/wayland-sessions/. Для проверки данного файла был установлен экранный менеджер sddm. Пример выбора ОС в sddm приведен на рисунке 12.

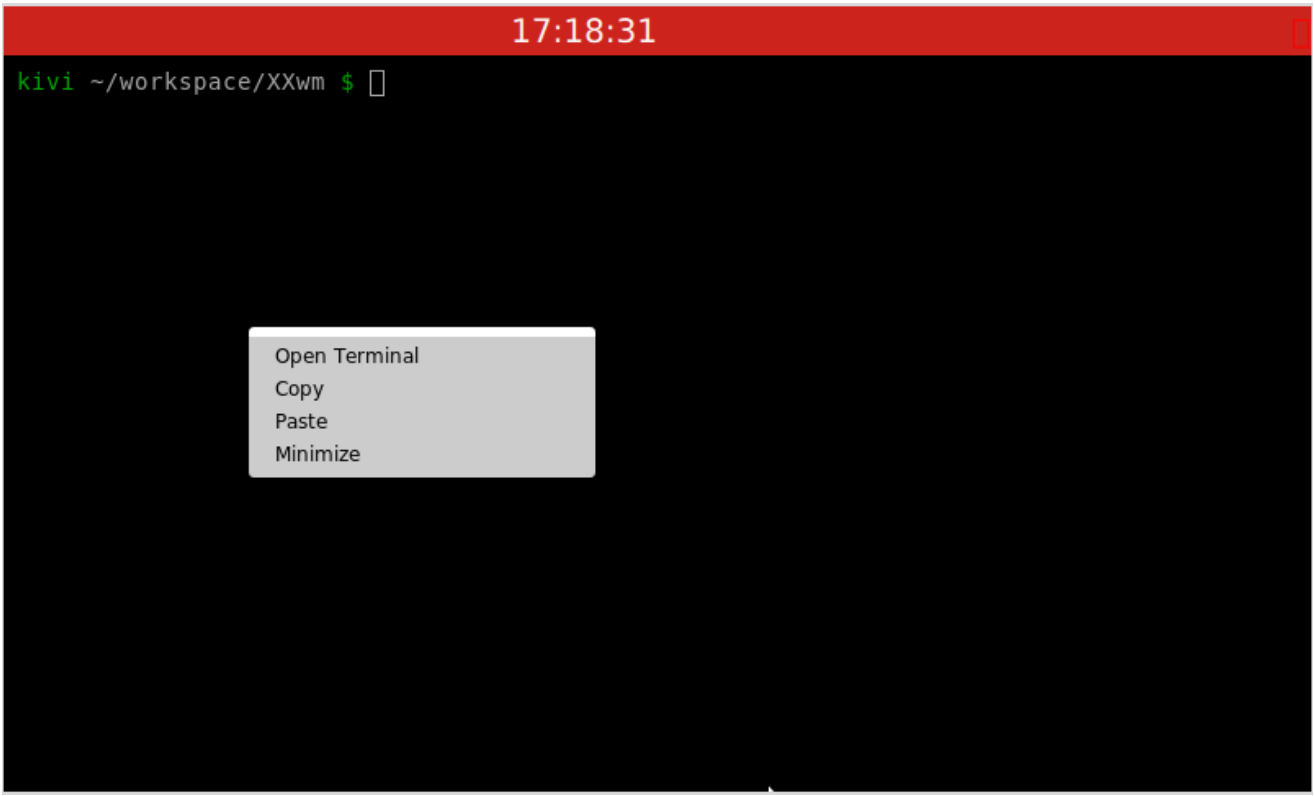

Рис. 9: Отображение терминала и контекстного меню

6 Выводы

В ходе работы были проанализированы и сравнены протоколы организации графических серверов в UNIX-подобных системах. В результате анализа было решено, что Wayland является более современной и оптимальной системой для разработки мобильного оконного менеджера. В данной работе был реализован мобильный оконный менеджер для протокола Wayland. Данный оконный менеджер разрабатывался в рамках проекта по разработке мобильного телефона. Разработанный оконный менеджер позволяет запускать системные приложения (строка состояния и рабочий стол) и обычные пользовательские приложения. Оконный менеджер так же обрабатывает несколько комбинаций клавиш для управления окнами, а так же позволяет перемещать окна и изменять из размеры с помощью мыши.

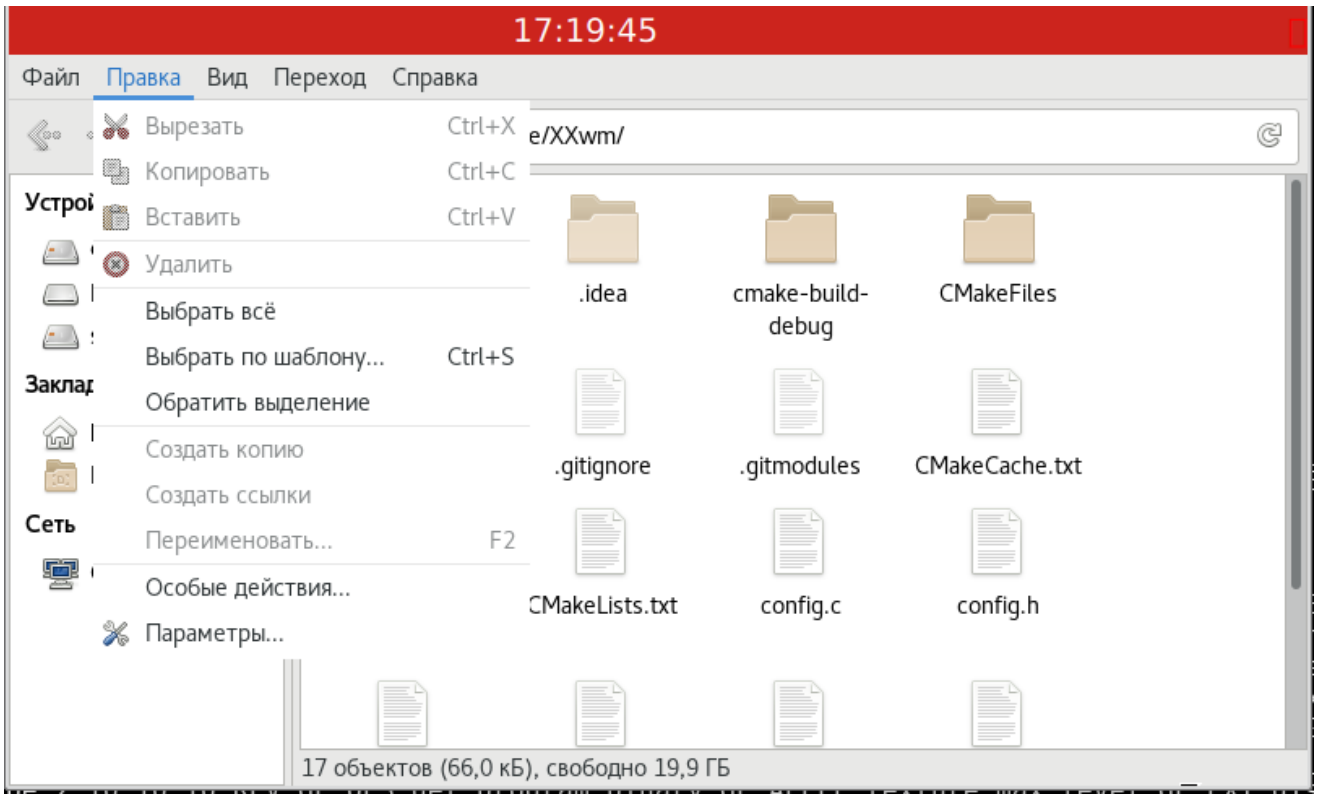

Рис. 10: Отображение файлового менеджера и меню

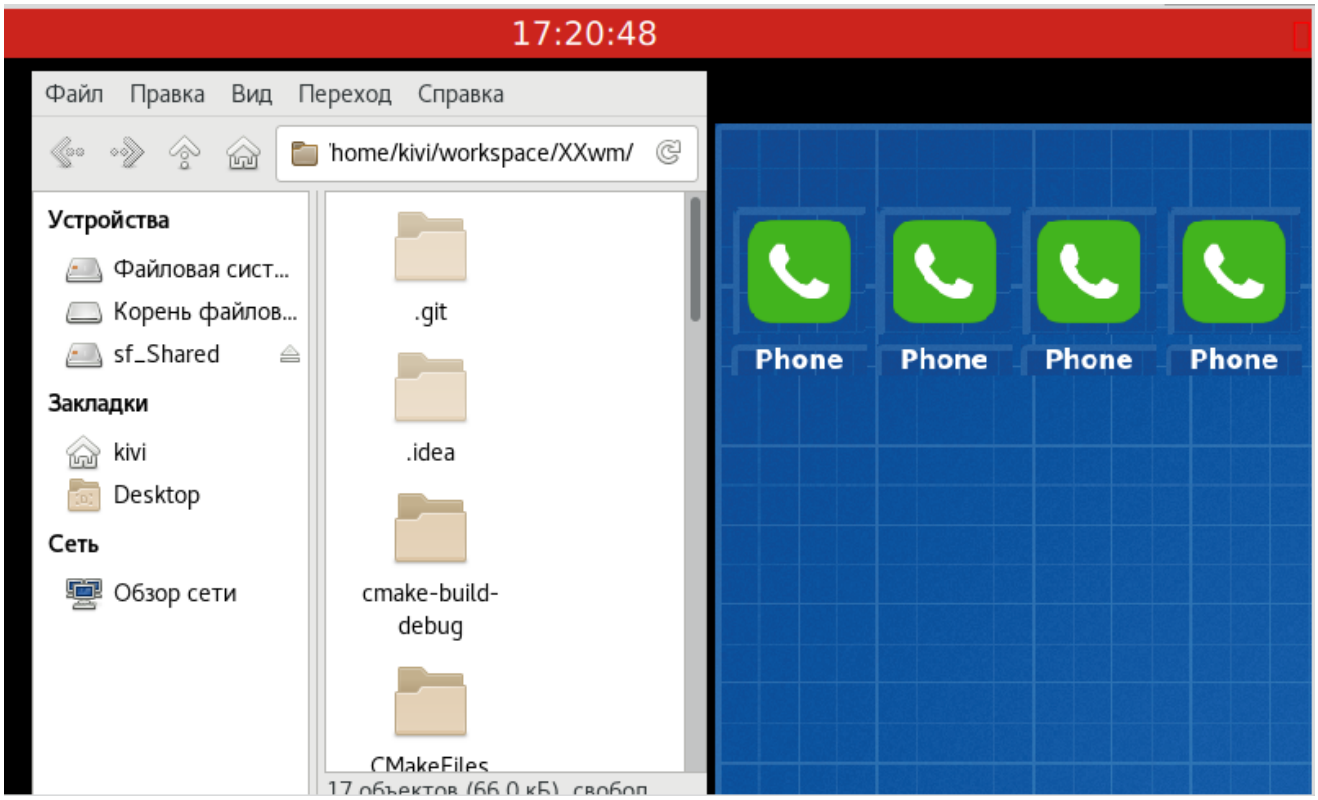

Рис. 11: Перемещение и изменение размеров окон

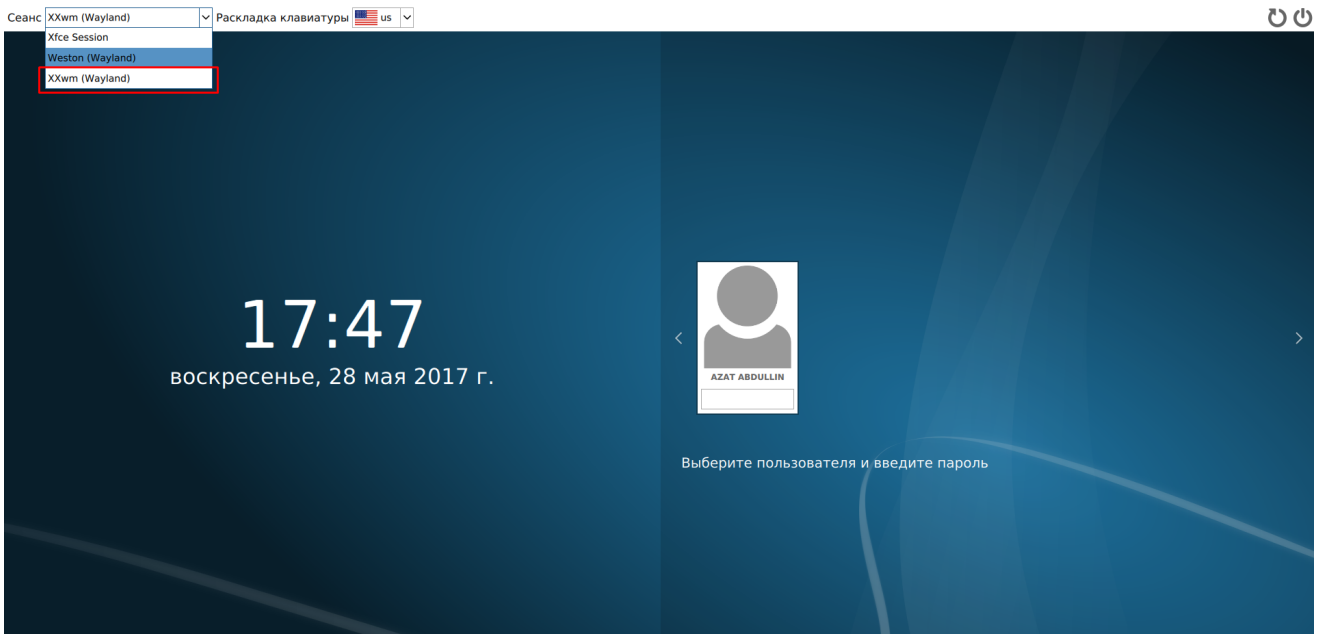

Рис. 12: Пример загрузки ОМ через экранный менеджер

Литература

[1] Официальный сайт Raspberry Pi. — https://www.raspberrypi.org/products/pi-zero/. — 2017. — Accessed: 2017-05-28.

[2] Rosenthal David, Marks Stuart W. Inter-Client Communication Conventions Manual // SGFR92. — 1991.

[3] Официальный сайт Wayland. — https://wayland.freedesktop.org/. — 2017. — Accessed: 2017-05-28.

[4] Инструкция к установке Weston. — https://wayland.freedesktop.org/building.html. — 2017. — Accessed: 2017-05-28.

[5] Исходные коды WLC. — https://github.com/Cloudef/wlc. — 2017. — Accessed: 2017-05-28.

[6] Описание драйвера VC4. — https://github.com/anholt/mesa/wiki/VC4. — 2017. — Accessed: 2017-05-28.

[7] Сандарт формата desktop. — https://specifications.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html. — 2017. — Accessed: 2017-05-28.

7 Прилагаемые материалы

Все прилагаемые материалы находятся в папке man. Список прилагаемых материалов следующий:

- Техническое задание.docx — техническое задание на проект;
- programmingGuilde.pdf — руководство системного программиста;
- programText.pdf — текст программы;
- spec.pdf — описание программы;
- testGuide.pdf — программа и методика испытаний;
- userGuide.pdf — руководство пользователя.

Так же прилагается пояснительная записка (report/report.pdf).

Листинги

Листинг 3: Файл main.c

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <wlc/wlc.h>
#include <linux/input.h>

#include "config.h"

#define DEFAULT_CONFIG "~/.config/xxwm"

pid_t statusbar_pid;          // pid of statusbar process
wlc_handle statusbar = 0;     // statusbar view
pid_t desktop_pid;            // pid of desktop app
wlc_handle desktop = 0;       // desktop view

// current view and details of active interactive action
static struct {
    struct {
        wlc_handle view;
        struct wlc_point grab;
        uint32_t edges;
    } action;
} compositor;

// функция, которая запускает интерактивное действие
static bool start_interactive_action(wlc_handle view, const struct
   wlc_point *origin) {
    // если нет активного окна, выходим
    if (compositor.action.view)
        return false;

    // запоминаем активное окно
    compositor.action.view = view;
    // изначальную точку расположения окна
    compositor.action.grab = *origin;
    // выводит активное окно на первый план
    wlc_view_bring_to_front(view);
    return true;
}

// начинает интерактивное передвижение окна
static void start_interactive_move(wlc_handle view, const struct
   wlc_point *origin) {
```

```
42      start_interactive_action(view, origin);
43  }
44
45  // начинает интерактивное изменение размеров окна
46  static void start_interactive_resize(wlc_handle view, uint32_t edges
        , const struct wlc_point *origin) {
47      const struct wlc_geometry *g;
48      // получаем параметры текущего окна
49      // и начинаем интерактивное действие
50      if (!(g = wlc_view_get_geometry(view)) || !
        start_interactive_action(view, origin))
51          return;
52
53      // определяем, какую грань окна передвигать
54      const int32_t halfw = g->origin.x + g->size.w / 2; // координата
        середины окна по горизонтали
55      const int32_t halfh = g->origin.y + g->size.h / 2; // координата
        середины окна по вертикали
56      if (!(compositor.action.edges = edges)) {
57          // если начальная точка окна левее середины, перемещаем левую грань
58          // если начальная точка окна правее середины, перемещаем правую грань
59          // точно так же с верхней и нижней границей
60          compositor.action.edges = (origin->x < halfw ?
        WLC_RESIZE_EDGE_LEFT : (origin->x > halfw ? WLC_RESIZE_EDGE_RIGHT
          : 0)) |
61                                    (origin->y < halfh ?
        WLC_RESIZE_EDGE_TOP : (origin->y > halfh ? WLC_RESIZE_EDGE_BOTTOM
          : 0));
62      }
63
64      // устанавливаем флаг, что окно в текущий момент меняет свой размер
65      wlc_view_set_state(view, WLC_BIT_RESIZING, true);
66  }
67
68  // завершение интерактивного действия
69  static void stop_interactive_action(void) {
70      // если нет активного окна, ничего не делаем
71      if (!compositor.action.view)
72          return;
73
74      // снимаем флаг изменения размеров окна
75      wlc_view_set_state(compositor.action.view, WLC_BIT_RESIZING,
        false);
76      // обнуляем все флаги действий активного окна
77      memset(&compositor.action, 0, sizeof(compositor.action));
78  }
79
```

```
80  // получить самое верхнее окно по смещению
81  static wlc_handle get_topmost(wlc_handle output, size_t offset) {
82      size_t memb;
83      // получаем все окна
84      const wlc_handle *views = wlc_output_get_views(output, &memb);
85      // возвращаем окно по смещению, или 0 если окон нет
86      return (memb > 0 ? views[(memb - 1 + offset) % memb] : 0);
87  }
88
89  // перерисовка всех окон
90  static void relayout(wlc_handle output) {
91      const struct wlc_size *r;
92      // получаем разрешение OM
93      if (!(r = wlc_output_get_virtual_resolution(output)))
94          return;
95
96      // получаем самое верхнее окно
97      wlc_handle topview = get_topmost(output, 0);
98      uint32_t shift = 30;
99      // получаем имя верхнего окна
100     const char* app = wlc_view_get_title(topview);
101     // создаем структуру с параметрами отрисовки окна статусбара
102     const struct wlc_geometry gstatus = {
103             .origin = {
104                     .x = 0,
105                     .y = 0
106             },
107             .size = {
108                     .w = r->w,
109                     .h = shift
110             }
111     };
112     // получаем тип окна и определяем как его рисовать
113     uint32_t viewtype = wlc_view_get_type(topview);
114     bool isPopup = (viewtype & WLC_BIT_POPUP) == WLC_BIT_POPUP;
115     bool isUnmanaged = (viewtype & WLC_BIT_UNMANAGED) ==
    WLC_BIT_UNMANAGED;
116     // если статусбар до сих пор не запущен, значит первое окно это статусбар
117     if (statusbar == 0 && wlc_view_get_pid(topview) == statusbar_pid
    ) {
118         statusbar = topview;
119     // если статусбар запущен, значит перерисовывается другое окно которое( на
    данный момент активно)
120     } else {
121         if (wlc_view_get_pid(topview) == desktop_pid) {
122             desktop = topview;
123         }
```

```
124         if (isUnmanaged) {
125             // если установлен флаг unmanaged, значит нам не надо менять
    параметры данного окна,
126             // тк.. это меню приложения с уже верно заданными параметрами
127
128         } else if (isPopup) {
129             // если установлен флаг popup, значит создается всплывающее
    уведомление
130             const struct wlc_geometry* anchor_rect =
    wlc_view_positioner_get_anchor_rect(topview);
131             // получаем размеры окна
132             struct wlc_size size_req = *wlc_view_positioner_get_size
    (topview);
133             if ((size_req.w <= 0) || (size_req.h <= 0)) {
134                 const struct wlc_geometry* current =
    wlc_view_get_geometry(topview);
135                 size_req = current->size;
136             }
137             // задаем размеры окна такие же, как и были при создании
138             struct wlc_geometry gpopup = {
139                     .origin = anchor_rect->origin,
140                     .size = size_req
141             };
142             // если есть родительское окно, смещаем popup относительно его
    начала
143             wlc_handle parent = wlc_view_get_parent(topview);
144             if (parent) {
145                 const struct wlc_geometry* parent_geometry =
    wlc_view_get_geometry(parent);
146                 gpopup.origin.x += parent_geometry->origin.x;
147                 gpopup.origin.y += parent_geometry->origin.y;
148             }
149             // перерисовываем окно
150             wlc_view_set_geometry(topview, 0, &gpopup);
151
152         } else {
153             // иначе создается обычное окно
154             // задаем параметры отрисовки активного окна
155             const struct wlc_geometry gview = {
156                     .origin = {
157                             .x = 0,
158                             .y = shift
159                     },
160                     .size = {
161                             .w = r->w,
162                             .h = r->h - shift
163                     }
```

```
164                 };
165                 // перерисовываем активное окно
166                 wlc_view_set_geometry(topview, 0, &gview);
167             }
168         }
169         // на всякий случай перерисовываем статусбар тк(.. могло поменяться
        разрешение экрана)
170         if (statusbar != 0) wlc_view_set_geometry(statusbar, 0, &gstatus
        );
171 }
172
173 // смена разрешения окна OM
174 static void output_resolution(wlc_handle output, const struct
        wlc_size *from, const struct wlc_size *to) {
175         (void)from, (void)to;
176         relayout(output);
177 }
178
179 // обработка нового созданного приложения
180 static bool view_created(wlc_handle view) {
181         // устанавливаем флаги отображения окна поумолчанию(-- 1, те.. рисовать)
182         wlc_view_set_mask(view, wlc_output_get_mask(wlc_view_get_output(
        view)));
183         // переносим новое окно на первый план
184         wlc_view_bring_to_front(view);
185         // устанавливаем новое окно активным
186         wlc_view_focus(view);
187         // указываем, чтобы новое окно рисовалось во весь экран
188         wlc_view_set_state(view, WLC_BIT_FULLSCREEN, true);
189         // перерисовываем все окна
190         relayout(wlc_view_get_output(view));
191         return true;
192 }
193
194 // обработка завершившегося приложения
195 static void view_destroyed(wlc_handle view) {
196         // получаем самое верхнее окно и выносим его на первый план
197         wlc_view_focus(get_topmost(wlc_view_get_output(view), 0));
198         // перерисовываем все окна
199         relayout(wlc_view_get_output(view));
200 }
201
202 // установить окно активным
203 static void view_focus(wlc_handle view, bool focus) {
204         // устанавливаем окну флаг активнсти
205         wlc_view_set_state(view, WLC_BIT_ACTIVATED, focus);
206 }
```

```
207
208 // запрос на перемещение окна
209 static void view_request_move(wlc_handle view, const struct
        wlc_point *origin) {
210     start_interactive_move(view, origin);
211 }
212
213 // запрос на изменение размеров окна
214 static void view_request_resize(wlc_handle view, uint32_t edges,
        const struct wlc_point *origin) {
215     start_interactive_resize(view, edges, origin);
216 }
217
218 // запрос на изменение отображения окна
219 static void view_request_geometry(wlc_handle view, const struct
        wlc_geometry *g) {
220     (void)view, (void)g;
221     // заглушка, не позволяющая изменить окно
222 }
223
224 // обработка событий клавиатуры
225 static bool keyboard_key(wlc_handle view, uint32_t time,
226                          const struct wlc_modifiers *modifiers,
        uint32_t key, enum wlc_key_state state) {
227     (void)time, (void)key;
228     // получаем считанный с клавиатуры символ
229     const uint32_t sym = wlc_keyboard_get_keysym_for_key(key, NULL);
230
231     // если есть активное окно
232     if (view) {
233         // CTRL+q -- закрытие окна
234         if (modifiers->mods & WLC_BIT_MOD_CTRL && sym == XKB_KEY_q)
        {
235             // close the window, if it's not system view
236             if (state == WLC_KEY_STATE_PRESSED && view != statusbar
        && view != desktop) {
237                 wlc_view_close(view);
238             }
239             return true;
240         // CTRLстрелка+ вниз -- перелистывание окон
241         } else if (modifiers->mods & WLC_BIT_MOD_CTRL && sym ==
        XKB_KEY_Down) {
242             if (state == WLC_KEY_STATE_PRESSED) {
243                 wlc_view_send_to_back(view); // отправляем текущее
        окно на задний план
244                 wlc_view_focus(get_topmost(wlc_view_get_output(view)
        , 0)); // устанавливаем новое верхнее окно активным
```

```
245             }
246             return true;
247         }
248     }
249
250     // CTRL+Escape -- завершение работы
251     if (modifiers->mods & WLC_BIT_MOD_CTRL && sym == XKB_KEY_Escape)
        {
252         if (state == WLC_KEY_STATE_PRESSED) {
253             wlc_terminate();
254         }
255         return true;
256     // CTRL+Enter -- запускаем терминал
257     } else if (modifiers->mods & WLC_BIT_MOD_CTRL && sym ==
    XKB_KEY_Return) {
258         if (state == WLC_KEY_STATE_PRESSED) {
259             char *terminal = (getenv("TERMINAL") ? getenv("TERMINAL"
    ) : "weston-terminal");
260             wlc_exec(terminal, (char *const[]){ terminal, NULL });
261         }
262         return true;
263     }
264
265     return false;
266 }
267
268 // обработка нажатий кнопок мыши
269 static bool pointer_button(wlc_handle view, uint32_t time, const
    struct wlc_modifiers *modifiers,
270                            uint32_t button, enum wlc_button_state
    state, const struct wlc_point *position) {
271     (void)button, (void)time, (void)modifiers;
272
273     // если кнопка нажата, то начинаем интерактивное действие
274     if (state == WLC_BUTTON_STATE_PRESSED) {
275         wlc_view_focus(view);
276         if (view) {
277             // CTRLлевая+ кнопка -- передвигаем окно
278             if (modifiers->mods & WLC_BIT_MOD_CTRL && button ==
    BTN_LEFT)
279                 start_interactive_move(view, position);
280             // CTRLправая+ кнопка -- изменяем размеры окна
281             if (modifiers->mods & WLC_BIT_MOD_CTRL && button ==
    BTN_RIGHT)
282                 start_interactive_resize(view, 0, position);
283         }
284     // иначе завершаем интерактивное действие
```

```
285     } else {
286         stop_interactive_action();
287     }
288 
289     return (compositor.action.view ? true : false);
290     //return false;
291 }
292 
293 // обарботка движения мыши
294 static bool pointer_motion(wlc_handle handle, uint32_t time, const
        struct wlc_point *position) {
295     (void)handle, (void)time;
296 
297     // если есть активное окно
298     if (compositor.action.view) {
299         // определяем координаты мышки относительно окна
300         const int32_t dx = position->x - compositor.action.grab.x;
301         const int32_t dy = position->y - compositor.action.grab.y;
302         struct wlc_geometry g = *wlc_view_get_geometry(compositor.
        action.view);
303 
304         // если есть запросы на перерисовку границ окна
305         if (compositor.action.edges) {
306             const struct wlc_size min = { 80, 40 }; // минимально
        допустиымые размеры окна
307 
308             struct wlc_geometry n = g;
309             if (compositor.action.edges & WLC_RESIZE_EDGE_LEFT) {
310                 n.size.w -= dx;
311                 n.origin.x += dx;
312             } else if (compositor.action.edges &
        WLC_RESIZE_EDGE_RIGHT) {
313                 n.size.w += dx;
314             }
315 
316             if (compositor.action.edges & WLC_RESIZE_EDGE_TOP) {
317                 n.size.h -= dy;
318                 n.origin.y += dy;
319             } else if (compositor.action.edges &
        WLC_RESIZE_EDGE_BOTTOM) {
320                 n.size.h += dy;
321             }
322 
323             if (n.size.w >= min.w) {
324                 g.origin.x = n.origin.x;
325                 g.size.w = n.size.w;
326             }
```

```
327
328             if (n.size.h >= min.h) {
329                 g.origin.y = n.origin.y;
330                 g.size.h = n.size.h;
331             }
332
333             // устанавливаем новые размеры окна
334             wlc_view_set_geometry(compositor.action.view, compositor
    .action.edges, &g);
335         // если нет запросов на изменение размеров окна, значит мы его
    перемещаем
336         } else {
337             g.origin.x += dx;
338             g.origin.y += dy;
339             wlc_view_set_geometry(compositor.action.view, 0, &g);
340         }
341
342         // запоминаем текущую позицию мыши
343         compositor.action.grab = *position;
344     }
345
346     // Устанавливаем координаты мышки и возвращаем управление композитору
347     wlc_pointer_set_position(position);
348     return (compositor.action.view ? true : false);
349 }
350
351 // функция логирования
352 static void cb_log(enum wlc_log_type type, const char *str) {
353     (void)type;
354     printf("%s\n", str); // выводим все в стандартный поток вывода
355 }
356
357 int main(int argc, char** argv) {
358     // parse input arguments
359     char* config = DEFAULT_CONFIG;
360     if (argc < 2) {
361         printf("No␣config␣file␣specified,␣using␣default:␣%s\n",
    config);
362     } else {
363         config = argv[1];
364         printf("Using␣config␣file:␣%s\n", config);
365     }
366     // parse config
367     init_config(config);
368
369
370     // устанавливаем функциюлоггер–
```

```
371     wlc_log_set_handler(cb_log);
372
373     // устанавливаем функцию, которая отвечает за перерисовку всех окон при
        смене разрешения
374     wlc_set_output_resolution_cb(output_resolution);
375     // устанавливаем функцию, которая отвечает за обработку запущенный
        приложений
376     wlc_set_view_created_cb(view_created);
377     // устанавливаем функцию, которая отвечает за обработку завершившихся
        приложений
378     wlc_set_view_destroyed_cb(view_destroyed);
379     // устанавливаем функцию, которая отвечает за смену фокуса между окнами
        выбирает( активное приложение)
380     wlc_set_view_focus_cb(view_focus);
381     // устанавливаем функцию, которая отвечает за передвижение окна ОМ по
        экрану
382     wlc_set_view_request_move_cb(view_request_move);
383     // устанавливаем функцию, которая отвечает за смену размеров окна ОМ
384     wlc_set_view_request_resize_cb(view_request_resize);
385     // устанавливаем функцию, которая отвечает за смену изменения отображения
        окна ОМ
386     wlc_set_view_request_geometry_cb(view_request_geometry);
387     // устанавливаем функциюобработчик– нажатий на клавиатуру
388     wlc_set_keyboard_key_cb(keyboard_key);
389     // устанавливаем функцию, которая отвечает за обработку нажатия кнопок
        мыши
390     wlc_set_pointer_button_cb(pointer_button);
391     // устанавливаем функцию, которая отвечает за передвижение мыши
392     wlc_set_pointer_motion_cb(pointer_motion);
393
394     // инициализируем композитор
395     if (!wlc_init())
396         return EXIT_FAILURE;
397
398     // spawning process that starts statusbar
399     statusbar_pid = fork();
400     if (statusbar_pid < 0) {
401         printf("Startup␣launch␣failure\n");
402         return EXIT_FAILURE;
403
404     } else if (statusbar_pid == 0) {
405         const char *statusbar = get_statusbar();
406         printf("Running␣statusbar␣%s\n", statusbar);
407         execv(statusbar, (char *const[]) {statusbar, NULL});
408
409     } else {
410         // spawning process that starts desktop
```

```
411         desktop_pid = fork();
412         if (desktop_pid < 0) {
413             printf("Startup launch failure\n");
414             return EXIT_FAILURE;
415 
416         } else if (desktop_pid == 0) {
417             // sleep for some time, wait until the WM actually
    starts
418             usleep(250000);
419             const char *desktop = get_desktop();
420             printf("Running desktop %s\n", desktop);
421             execv(desktop, (char *const[]) {desktop, NULL});
422 
423         }
424 
425         printf("Running WM\n");
426         wlc_run();
427     }
428 
429     return EXIT_SUCCESS;
430 }
```

Листинг 4: Файл config.h

```
1  //
2  // Created by kivi on 17.05.17.
3  //
4 
5  #ifndef XXONWM_CONFIG_H
6  #define XXONWM_CONFIG_H
7 
8  void init_config(const char* config_file);
9  const char* get_statusbar();
10 const char* get_desktop();
11 
12 #endif //XXONWM_CONFIG_H
```

Листинг 5: Файл config.c

```
1  //
2  // Created by kivi on 17.05.17.
3  //
4  #include <string.h>
5  #include <stdlib.h>
6 
7  #include "ini.h"
8 
9  typedef struct {
```

```
10      char* statusbar;
11      char* desktop;
12  } configuration;
13
14  configuration config = { NULL, NULL };
15
16  static int handler(void* user, const char* section, const char* name
        ,
17                     const char* value)
18  {
19      configuration* pconfig = (configuration*)user;
20
21  #define MATCH(s, n) strcmp(section, s) == 0 && strcmp(name, n) == 0
22      if (MATCH("statusbar", "exe")) {
23          pconfig->statusbar = strdup(value);
24      } else if (MATCH("desktop", "exe")) {
25          pconfig->desktop = strdup(value);
26      } else {
27          return 0;  /* unknown section/name, error */
28      }
29      return 1;
30  }
31
32  void init_config(const char* config_file) {
33      if (ini_parse(config_file, handler, &config) < 0) {
34          printf("Can't load %s\n", config_file);
35          exit(1);
36      }
37  }
38
39  const char* get_statusbar() {
40      return config.statusbar;
41  }
42
43  const char* get_desktop() {
44      return config.desktop;
45  }
```

Листинг 6: Файл сборки CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.7)
2  project(xxonwm)
3  set(CMAKE_C_STANDARD 99)
4
5  # includes for wlc
6  include_directories(/usr/include)
7  include_directories(/usr/local/include)
8  include_directories(/usr/local/include/lib/chck)
```

```
9  link_directories(/usr/local/lib64/)
10
11 # include inih
12 add_subdirectory(inih)
13 include_directories(inih)
14 link_directories(inih)
15
16
17 set(SOURCE_FILES main.c config.c)
18 add_executable(xxonwm ${SOURCE_FILES})
19
20 target_link_libraries(xxonwm wlc inih)
```