МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ, профессор
А.М. Кориков

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Тема 2. Архитектура процессоров

Учебно-методическое пособие

для студентов уровня основной образовательной программы: **магистратура**
направление подготовки: **09.04.01 - Информатика и вычислительная техника**

Разработчик
доцент кафедры АСУ

В.Г. Резник

2017

Резник В.Г.

Архитектура вычислительных комплексов. Тема 2. Архитектура процессоров. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 108 с.

Учебно-методическое пособие предназначено для изучения темы №2 по дисциплине «Архитектура вычислительных комплексов» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Оглавление

Введение

Данное методическое пособие содержит учебный материал по второй теме дисциплины «*Архитектура вычислительных комплексов*», - сокращенно АВК.

Изложенный материал является обязательной частью процесса обучения магистранта по направлению подготовки 09.04.01 «Информатика и вычислительная техника» и содержит как теоретическую часть, так и методические указания по выполнению трех лабораторных лабораторных работ.

Последовательность и тематическая направленность учебного материала данного пособия предполагает, что магистрант успешно освоил теоретический материал по первой теме дисциплины, а также выполнил первые три лабораторные работы.

Изложенный материал разбит на ряд разделов, последовательность которых определяет сам порядок процесса обучения.

В первом разделе представлен торетический материал по объявленной теме «Архитектура процессоров», который рассчитан на 8 академических часов и, в дальнейшем, закрепляется с помошью трех лабораторных работ.

Второй, третий и четвертый разделы содержат методические описания самих лабораторных работ, выполняемых в рамках данной темы, пронумерованных в общем порядке и озаглавленных:

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

Замечание

Технология проведения всех работ, предполагает, что каждый студент должен иметь flash-память не менее 2 Гб для сохранения личного материала и запуска УПК АСУ.

1 Тема 2. Архитектура процессоров

В модели системы обработки данных (СОД), отдельный компьютер (ЭВМ) является простейшим элементом архитектуры.

Как было показано в предыдущей теме, архитектурные представления определяются уровнем, на котором рассматривается система. Согласно модели общей архитектуры, ЭВМ представляется в виде четырех составляющих:

- центрального процессора (ЦП);
- основной памяти (ОП);
- устройств ввода/вывода (УВВ);
- системы шин.

Как в историческом плане, так и в технологических аспектах развития, центальный процессор (***ЦП***) претерпел наиболее сильные архитектурные изменения, сконцентрировав в себе самые современные достижения науки и техники.
Изучению этих *архитектурных изменений* и посвящена данная тема.

Поскольку, процессор является не монолитным блоком, а *набором вычислительных элементов*, то рассмотрение этих архитектурных изменений является актуальным и полезным занятием.

1.1 Микропрограммный способ выполнения команд

В большинстве современных ЭВМ непосредственная связь между аппаратурой и программными средствами осуществляется через ***микропрограммный уровень***.

Любая машинная команда:

- ***сначала интерпретируется*** в последовательность более простых действий,
- а затем ***непосредственно исполняется*** аппаратурой.

Таким образом, ***микропрограммирование*** - это задача программирования машинных команд из более простых действий.

Впервые, этот термин был введен в ***1953 году Уилксом*** и применялся только к аппаратным средствам.

В середине 60-х годов, усилиями разработчиков IBM, идеи Уилкса превратились в принцип организации ЭВМ.

Микропрограммирование обеспечило переход к ***модульному построению ЭВМ***.

Академик ***Глушков Виктор Михайлович***, развил идеи микропрограммирования и показал, что:

- в любом устройстве обработки информации можно выделить ***операционный***

и ***управляющий автоматы***;

- ***структура обработки*** любой информации, включая центральный процессор (ЦП), может быть представлена рисунком 1.1.

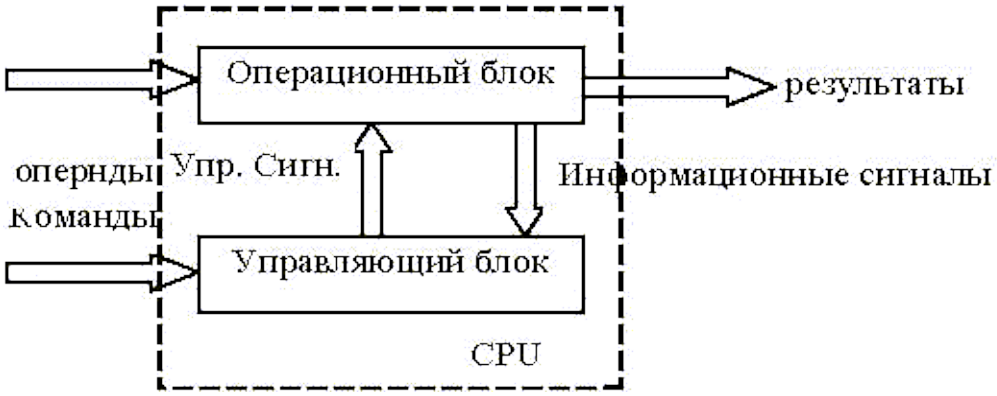

Рисунок 1.1 - Структура обработки информации в ЦП

Управляющий блок выдает последовательность сигналов, которые обеспечивают выполнение данной команды.

Информационные сигналы зависят не только от исходных значений обрабатываемых данных, но и от результатов, получаемых в процессе обработки.

Порядок функционирования такого устройства базируется на следующих положениях:

- **Любая машинная команда** рассматривается, как некоторое сложное действие, которое состоит из последовательности элементарных действий над словами информации - ***микроопераций***.
- **Порядок следования микроопераций** зависит не только от значений преобразуемых слов, но также от их информационных сигналов, вырабатываемых операционным автоматом. Примерами таких сигналов могут быть признаки результата операции, значения отдельных битов данных и другие.
- **Процесс выполнения машиной команды** описывается в виде некоторого алгоритма в терминах ***микроопераций*** и ***логических условий***. Описание информационных сигналов - ***микропрограмма***.
- **Микропрограмма** служит не только для обработки данных, но и обеспечивает управление работой всего устройства в целом - ***принцип микропрограммного управления***.
- **Операционный блок** обеспечивает выполнение определенного набора микроопераций и вычисление необходимых логических условий.
- **Управляющий автомат**, согласно заданной машинной команде, генерирует необходимую последовательность сигналов, инициирующих соответствующие микрооперации, согласно микропрограмме и значениями логических условий, формируемых операционным блоком, в ходе обработки по микропрограмме.

Таким образом, **понятие микропрограммы** имеет дуальность:

- с одной стороны, - ***это закон***, по которому выполняется ***обработка***,
- с другой стороны, - ***это закон***, по которому работает ***управляющий блок***.

Операционный блок процессора представляет собой некоторую композицию ***операционных элементов***.

Операционный элемент имеет ***аппаратную реализацию*** в виде "отдельной" части.

Операционный элемент обеспечивает:

- *хранение* слов и *выполнение* микроопераций над словами и их полями,
- а также *вычисление* логических условий.

Все элементы в операционном автомате соединяются между собой с помощью ***шин***, которые обеспечивают передачу слов ***с выхода одного*** операционного элемента ***на вход другого***.

Операционные элементы процессора разделяются на:

- шины,
- регистры,
- счетчики,
- сумматоры,
- логические устройства,
- устройства сдвига,
- преобразователи и формирователи кодов,
- комбинированные операционные элементы.

1.2 CISC и RISC архитектуры

Важным понятием архитектуры процессора является «***Архитектура набора команд***».

Набор команд служит границей между аппаратурой и программным обеспечением.

Набор команд представляет ту часть системы, которая видна программисту или разработчику компиляторов.

Двумя основными классами, различающимися наборами команд, являются процессора ***CISC*** и ***RISC*** архитектуры.

1.2.1 CISC-архитектура

Основоположником **CISC-архитектуры** считается компанию ***IBM*** с ее базовой архитектурой ***System/360***.

Ядро с такой архитектурой используется ***с 1964 года*** и дошло до наших дней в ***мейнфреймах IBM ES/9000***.

CISC (*Complete Instruction Set Computers*) - архитектура вычислений с полной системой команд.

CISC реализует на уровне машинного языка комплексные наборы команд различной сложности:

- **простые**, характерные для микропроцессоров первого поколения;
- **сложные**, характерные для современных 32-разрядных микропроцессоров типа 80486, 68040 и других.

Лидером CISC-микропроцессоров считается компания ***Intel*** со своей серией ***x86*** и ***Pentium***.

Эта архитектура является практическим стандартом для рынка микрокомпьютеров.

Для **CISC-процессоров** характерно:

- сравнительно небольшое число *регистров* общего назначения;
- большое количество *машинных команд*, которые нагружены семантически аналогично операторам высокоуровневых языков программирования и выполняются за много тактов;
- большое количество *методов адресации*;
- большое количество *форматов команд различной разрядности*;
- преобладание *двухадресного* формата команд;
- наличие команд обработки типа *регистр-память*.

1.2.2 RISC-архитектура

RISC (*Reduced Instruction Set Computers*) используют сравнительно небольшой (***сокращенный***) набор наиболее употребительных команд.

Набор команд определен в результате статистического анализа большого числа программ *из основных областей применения CISC - процессоров* исходной архитектуры.

Система команд разрабатывалась со стремлением *выполнить команду за один машинный такт*. Сама логика выполнения команд ориентировалась на *аппаратную*, а не на *микропрограммную* реализацию.

Все команды работают с *операндами* и имеют *одинаковый формат*.

Обращение к памяти выполняется с помощью специальных команд *загрузки регистра* и *записи*.
Простота структуры и **небольшой набор команд** позволяет реализовать процессор при небольшом объеме оборудования.

Арифметику RISC - процессоров отличает *высокая степень дробления конвейера*. Этот прием позволяет ***увеличить тактовую частоту*** процессора.

RISC-процессора в ***2 - 4 раза быстрее*** CISC - процессоров, с обычной системой команд, и производительней их в условиях равной тактовой частоты.

Четыре основных принципа RISC:

- Любая операция должна выполняться *за один такт*, вне зависимости от ее типа.
- Система команд должна содержать *минимальное количество* наиболее часто используемых *простейших инструкций одинаковой длины*.
- Операции обработки данных реализуются только в формате "*регистр — регистр*": операнды выбираются из оперативных регистров процессора, и результат операции записывается также в регистр, а обмен между оперативными регистрами и памятью выполняется только с помощью команд загрузки-записи.
- *Состав системы команд должен быть "удобен"* для компиляции операторов языков высокого уровня.

1.2.3 Краткое сравнение архитектур

Основная особенность RISC-архитектур - *большое количество регистров*:

- RISC-процессора содержат порядка ***32 или большее число*** регистров; это упрощает дешифрацию команд и дает возможность сохранять большее число переменных в регистрах без их последующей перезагрузки;
- CISC-процессора — содержат порядка ***8 - 16*** регистров и, как правило, используют *трехадресные команды*.

Основные характеристики архитектур типовых микропроцессоров:

Характеристика	***CISC***	***RISC***
Формат команд	Переменный	Фиксированный
Структура команд	Сложная	Простая
Выполнение всех команд	Аппаратно-программное	Аппаратное
Число команд	Большое	Небольшое

Число регистров	Небольшое	Большое
Время обработки прерывания	Среднее	Очень малое
Среднее число тактов за инструкцию	4 - 6	1 - 2

Современные тенденции развития CISC-процессоров идут в направлении:

- более совершенного ***управления*** машинными ресурсами;
- ***сближения*** машинных языков с языками высокого уровня.

С другой стороны, быстрый рост сложности схем привел к пределу возможностей CISC- архитектуры в рамках существующей кремниевой технологии:

- 80386 содержит 270 тыс. транзисторов,
- 80486 - 1 млн. транзисторов.

Современные тенденции RISC-процессоров – их усложнение, что фактически приближает их архитектуру к CISC-архитектуре.

В настоящее время, МП с RISC-архитектурой производят все ведущие фирмы США, в том числе фирмы Intel и Motorola.

1.3 Скалярные и векторные процессоры

Скалярный процессор (*SISD, Single Instruction Single Data*) — это простейший класс микропроцессоров.
Скалярный процессор обрабатывает *один элемент данных* за *одну инструкцию*.
Типичными элементами данных могут быть *целые числа* или *числа с плавающей запятой*.

Векторный процессор (*SIMD, Single Instruction Multiple Data*), когда одна инструкция работает с несколькими элементами данных.
Векторный процессор выполняет команды, в которых *операндами могут быть упорядоченные массивы данных — векторы*.
Векторные процессоры были распространены в сфере научных вычислений, где они являлись основой большинства суперкомпьютеров, начиная с ***1980-х*** до ***1990-х годов***.

Замечание

Резкое увеличение производительности и активная разработка новых процессоров *привели к вытеснению* векторных процессоров со сферы повседневного применения.

Многие современные микропроцессоры имеют:

- ***векторные расширения*** (SSE - *Streaming SIMD Extensions*), ***потоковое*** SIMD-расширение процессора,

- ***векторные сопроцессоры*** — современные *видеокарты* и физические ***ускорители***.

SSE - набор инструкций, разработанный фирмой Intel и впервые представленный в процессорах серии Pentium III.
SSE включает в архитектуру процессора ***восемь 128-битных регистров*** и ***набор инструкций***, работающих со скалярными и упакованными типами данных. Преимущество достигается, когда необходимо произвести одну и ту же последовательность действий над разными данными. Тогда *блок SSE* осуществляет ***распараллеливание вычислительного процесса*** между данными.

1.4 Конвейеры

Главным препятствием высокой скорости выполнения команд является *необходимость их вызова из памяти*.

Для устранения этой проблемы можно вызывать команды из памяти заранее и хранить в специальном наборе регистров.

Впервые эта идея стала использоваться **в 1959 году** при разработке компанией ***IBM*** компьютера ***Stretch***.

Такой набор регистров был назван *буфером выборки с упреждением*.

При выборке с упреждением команда обрабатывается за ***два шага***:
- сначала происходит *вызов команды*,
- а затем — *ее выполнение*.

Идея конвейера еще больше продвинула эту идею:
- *команда обрабатывается за несколько шагов*, каждый из которых реализуется определенным аппаратным компонентом,
- *все аппаратные компоненты работают параллельно*.

1.4.1 Пятиступенчатый конвейер

Первая ступень (блок C1) *вызывает команду* из памяти и помещает ее в буфер, где она хранится до тех пор, пока не потребуется.

Вторая ступень (блок C2) *декодирует эту команду*, определяя ее тип и тип ее операндов.

Третья ступень (блок C3) *определяет местонахождение операндов* и вызывает их из регистров или из памяти.

Четвертая ступень (блок С4) *выполняет команду*, обычно проводя операнды через тракт данных.

Пятая ступень (блок С5) *записывает результат* обратно в нужный регистр.

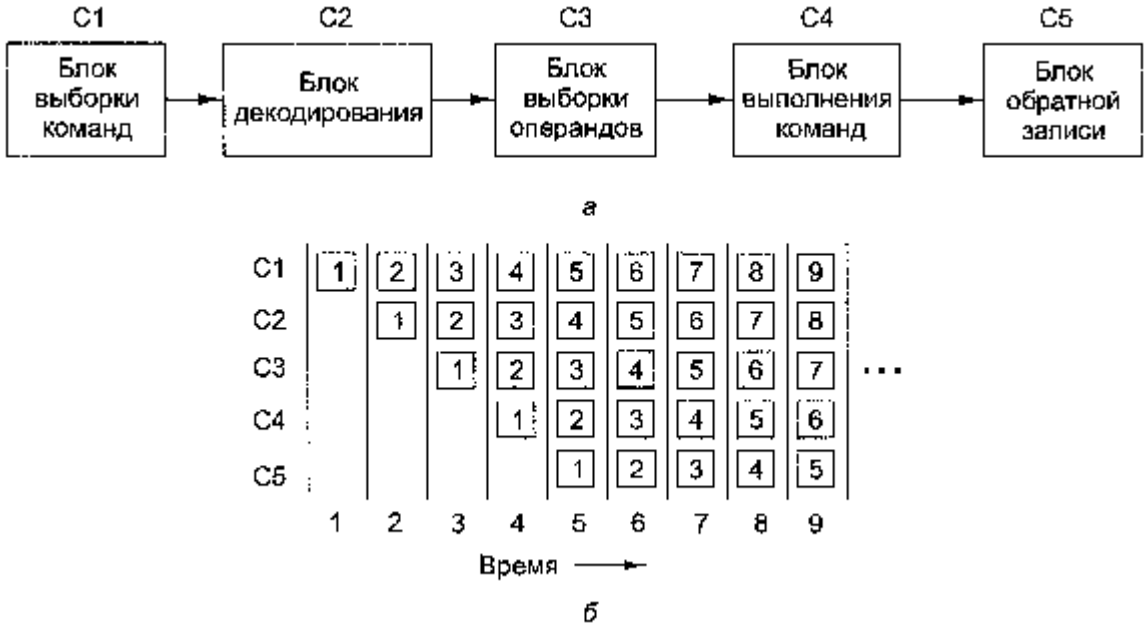

Рисунок 1.2 - Обобщенная схема работы конвейера

Рассмотрим **действие конвейера во времени**:

- Во время цикла 1 блок С1 ***обрабатывает команду 1***, вызывая ее из памяти.
- Во время цикла 2 блок С2 ***декодирует команду 1***, в то время как блок С1 вызывает из памяти команду 2.
- Во время цикла 3 блок С3 ***вызывает операнды для команды 1***, блок С2 декодирует команду 2, а блок С1 вызывает команду 3.
- Во время цикла 4 блок С4 ***выполняет команду 1***, С3 вызывает операнды для команды 2, С2 декодирует команду 3, а С1 вызывает команду 4.
- Наконец, во время цикла 5 блок С5 ***записывает результат*** выполнения команды 1 обратно в регистр, тогда как другие ступени конвейера обрабатывают следующие команды.

1.4.2 Увеличение частоты работы конвейеров

Предположим, что процессор выполняет отдельные команды с ***частотой N***.

Если разделить вычисление каждой команды на **M** одинаковых по длительности стадий, то в пределе можно считать, что конвейер работает с ***частотой N*M***.

Если такой конвейер *непрерывно загружать командами без задержки*, то после переходного процесса в **M-1** стадий мы будем иметь скорость выполнения стадий:

```
K = N*M**2.
```

Замечание

Все было бы хорошо, если бы в работе конвейера не возникали ***конфликты***.

1.5 Конфликты

Конфликты - это ситуации в конвейерной обработке, которые препятствуют выполнению очередной команды в предназначенном для нее такте.
Конфликты делятся на **три группы**:

- *структурные,*
- *по управлению,*
- *по данным.*

1.5.1 Структурные конфликты

Структурные конфликты возникают в том случае, когда аппаратные средства процессора не могут поддерживать все возможные комбинации команд *в режиме одновременного выполнения с совмещением*.

Причины структурных конфликтов:

1) Не полностью конвейерная структура процессора, при которой некоторые ступени отдельных команд выполняются более одного такта.
В этом случае, команда на некоторой стадии не может выполниться, потому что не выполнена предыдущая команда.

Для синхронизации такой ситуации процессор вводит одну или несколько «пустых команд»: NOP или 90H.

Такая ситуация *вносит задержку в работу всего конвейера*.
Образуемый при этом "**пузырь**" должен пройти от места своего возникновения до самого конца конвейера.

2) Недостаточное дублирование некоторых ресурсов.
Типичный пример - доступ к запоминающим устройствам, когда операнды и команды находятся в одном запоминающем устройстве.

В этом случае, различные команды в одном и том же такте обращаются к памяти на *считывание команды*, *выборку операнда*, *запись результата*.

1.5.2 Конфликты по управлению

Конфликты по управлению возникают при конвейеризации команд переходов и других команд, *изменяющих значение счетчика команд*.

Наиболее серьезной является проблема прерывания МП внешними устройствами.

При переходе на программу - **обработчику прерывания** необходимо *надежно очистить конвейер* и *сохранить состояние процессора* таким, чтобы повторное выполнение команды после возврата из прерывания осуществлялось корректно.

1.5.3 Конфликты по данным

Конфликты по данным возникают в случаях, когда выполнение одной команды зависит от результата выполнения предыдущей команды.

Существует несколько типов конфликтов по данным **RAW**, **WAR** и **WAW**:

1) Конфликты типа **RAW** (***Read After Write*** *- чтение после записи*): когда команда пытается прочитать операнд прежде, чем другая команда запишет на это место свой результат.

Пусть выполняемые команды имеют следующий вид:

```
i)   ADD R1,R0; R1=R1+R0
i+1) SUB R2,R1; R2=R2-R1
```

Команда i изменит состояние регистра R1 в такте 5. Но команда i+1 должна прочитать значение операнда R1 в такте 4. Если не приняты специальные меры, то из регистра R1 будет прочитано значение, которое было в нем до выполнения команды i.

Конфликты типа **RAW** обусловлены *именно конвейерной организацией* обработки команд.

Они называются **истинными взаимозависимостями**.

2) Конфликты типа **WAR** (***Write After Read*** *- запись после чтения*): когда **команда i+1** пытается записать результат в приемник, прежде чем он считается оттуда командой i. При этом команда i может получить некорректное новое значение операнда:

```
i)   ADD R1,R0; R1=R1+R0
i+1) SUB R0,R2; R0=R0-R2
```

Этот конфликт возникнет, если команда i+1 вследствие неупорядоченного выпол-

нения ***завершится раньше***, чем команда i прочитает старое содержимое регистра ***R0***.

3) Конфликты типа **WAW** (***Write After Write*** - *запись после записи*): когда **команда i+1** пытается записать результат в приемник, прежде чем в этот же приемник будет записан результат выполнения команды i, то есть запись заканчивается в неверном порядке, оставляя в приемнике результата значение, записанное командой i:

```
i) ADD R1,R0; R1=R1+R0
. . .
j) SUB R1,R2; R1=R1-R2
```

Устранение конфликтов по данным типов **WAR** и **WAW** достигается путем *отказа от неупорядоченного исполнения команд*, но чаще всего путем введения *буфера восстановления последовательности команд*.

В целом, для устранения конфликтов в конвейере процессора используется множество методов.

1.6 Динамическое исполнение команд

Основная идея динамической оптимизации - снятие *требования о выполнении команд в строгом порядке*.

Обычно, производится расщепление блока декодирования на две ступени:

- *Выдача* - декодирование команд, проверка структурных конфликтов.
- *Чтение операндов* - ожидание отсутствия конфликтов по данным и последующее чтение операндов.

В общем случае, необходимо строгую последовательность команд разложить в другую последовательность, *чтобы команды могли выполняться параллельно*.

CPI (*Clocks Per Instruction*) – количество тактов процессора на инструкцию.

```
Общий CPI конвейера = CPI идеального конвейера +

           +  Приостановки из-за структурных конфликтов +

           +  Приостановки из-за  конфликтов  типа  RAW +

           +  Приостановки из-за конфликтов типа  WAR   +

           +  Приостановки из-за  конфликтов  типа  WAW +

           +  Приостановки из-за конфликтов по управлению
```

CPI идеального конвейера - максимальная пропускная способность, достижимая при реализации.

Уменьшая каждое из слагаемых в правой части выражения, мы минимизируем общий CPI конвейера.

Перечислим методы, которые снижают общую CPI:

Метод	***Снижает***
Разворачивание циклов	Приостановки по управлению
Базовое планирование конвейера	Приостановки RAW
Динамическое планирование с централизованной схемой управления	Приостановки RAW
Динамическое планирование с переименованием регистров	Приостановки WAR и WAW
Динамическое прогнозирование переходов	Приостановки по управлению
Выдача нескольких команд в одном такте	Идеальный CPI
Анализ зависимостей компилятором	Идеальный CPI и приостановки по данным
Программная конвейеризация и планирование трасс	Идеальный CPI и приостановки по данным
Выполнение по предположению	Все приостановки по данным и управлению
Динамическое устранение неоднозначности памяти	Приостановки RAW, связанные с памятью

Самый общий способ увеличения степени параллелизма – «*разворачивание цикла*».

Он называется **параллелизмом уровня итеративного цикла**.

Эти методы выполняются:

- либо **статически**, используя компилятор;
- либо **динамически** с помощью аппаратуры.

Альтернативный метод - *использование векторных команд*.
Векторная команда оперирует с последовательностью элементов данных.
Эти команды могут выполняться конвейером и иметь относительно большие задержки выполнения, но эти задержки могут перекрываться.

1.6.1 Динамическая оптимизация с централизованной схемой обнаружения конфликтов

Впервые подобная схема была применена в компьютере **CDC 6600**:

- Все команды проходят через **ступень выдачи** строго в порядке, предписанном программой (упорядоченная выдача).
- команды могут **приостанавливаться** и **обходить** друг друга на ступени чте-

ния операндов и, тем самым, поступать на ступени выполнения неупорядочено.

Задача такой схемы - *обеспечить как можно более раннее начало выполнения команд*.

Все это требует одновременного нахождения нескольких команд на стадии выполнения.

Достигается это двумя способами:

- реализацией в процессоре множества *неконвейерных функциональных устройств*,
- путем *конвейеризации всех функциональных устройств*.

1.6.2 Алгоритм Томасуло

Схема приписывается ***Р. Томасуло***, названа его именем и имеет много общего со схемой централизованного управления ***CDC 6600***.

Впервые алгоритм был использован в устройстве *плавающей точки* (ПТ) в машине IBM 360, спустя три года после выпуска ***CDC 6600*** и прежде, чем кэш-память появилась в коммерческих машинах.

Существенные отличия алгоритма:

- **Во-первых**, обнаружение конфликтов и управление выполнением являются распределенными. В каждом функциональном устройстве используются *станции резервирования* **(reservation stations)**. Эти станции определяют, когда команда может начать выполняться в данном функциональном устройстве.
- **Во-вторых**, результаты операций *не проходят через регистры*, а посылаются прямо в функциональные устройства.

Алгоритм Томасуло имеет всего ***три стадии***:

1. **Выдача** - Берет команду из очереди команд ПТ.

- Если операция является операцией ПТ, выдает ее при наличии свободной станции резервирования и посылает операнды на станцию резервирования, если они находятся в регистрах.
- Если операция является операцией загрузки или записи, она может выдаваться при наличии свободного буфера. При отсутствии свободной станции резервирования или свободного буфера возникает **структурный конфликт** и команда приостанавливается до тех пор, пока не освободится станция резервирования или буфер.

2. **Выполнение** - Если один или более операндов команды недоступны, по каким-либо причинам, то ожидается завершение вычисления значений нужного регистра. На этой стадии выполняется контроль конфликтов типа RAW.

Когда оба операнда доступны, выполняется операция.

3. **Запись результата** - Когда становится доступным результат, он записывается на общую шину и оттуда в регистры или любое функциональное устройство, ожидающее этот результат.

1.7 Спекулятивное исполнение

Спекулятивное исполнение (***speculative***) - построение любой, желательно, оптимизированной последовательности, которая строится на ***методах предсказания***, т.е. - ***спекулятивно***.

Примером таких спекуляций являются последовательности команд, построенных ***на основе предсказания команд ветвлений*** (условные операторы).
Процессор вычисляет команды ***на обоих концах команды ветвления***, а результаты вычисления сохраняются как предположительные («*спекулятивные*»).
На некотором конечном этапе ***порядок инструкций восстанавливается***.

Варианты спекулятивного выполнения:

1. **предикация** (***predication***) - одновременное исполнение нескольких ветвей программы вместо предсказания переходов;
2. **опережающее чтение данных** (***speculative loading***), то есть загрузка данных в регистры с опережением, до того, как определилось реальное ветвление программы (переход управления).

Эти возможности осуществляются комбинированно: *при компиляции* и *выполнении программы*.

1.7.1 Предикации

Обычный компилятор транслирует оператор ветвления (например, ***if-then-else***) в блоки машинного кода, расположенные последовательно в потоке.

Обычный процессор в зависимости от исхода условия исполняет один из этих базовых блоков, пропуская все другие.

Более развитые процессоры пытаются прогнозировать исход операции и предварительно выполняют предсказанный блок.

В случае ошибки много тактов тратится впустую:

- Сами блоки зачастую ***весьма малы*** - две или три команды.
- Ветвления встречаются в коде программ, в среднем ***каждые шесть команд***.

При использовании предикации, ***компилятор***, обнаружив оператор ветвления в исходной программе, анализирует все возможные ветви (блоки) и помечает их ***метками*** или ***предикатами*** (***predicate***).
После этого, он определяет, какие из них могут быть выполнены параллельно.

В процессе выполнения программы, ***ЦП*** выбирает команды, которые являются *взаимно независимыми* и *распределяет их на параллельную обработку*.
Если ЦП обнаруживает оператор ветвления, то он не пытается предсказать переход, а начинает выполнять все возможные ветви программы.
Таким образом, могут быть обработаны ***все ветви программы***, но ***без записи*** полученного результата.

В определенный момент, процессор, **«узнает» о реальном исходе** условного оператора, записывает в память результат «***правильной ветви***» и отменяет остальные результаты.
Если ***компилятор*** *не «отметил» ветвление*, то процессор действует как обычно - пытается предсказать путь ветвления и так далее.

Испытания показали, что описанная технология позволяет устранить более половины ветвлений в типичной программе, и, следовательно, уменьшить ***более чем в 2 раза*** число возможных ошибок в предсказаниях.

1.7.2 Опережающее чтение данных

Опережающее чтение - предварительная загрузка данных или чтение *по предположению*.

Оно **разделяет** загрузку данных в регистры и их реальное использование, избегая ситуации, когда процессору приходится ожидать прихода данных, чтобы начать их обработку.

Сначала, компилятор **анализирует** программу, определяя **команды,** которые требуют приема данных из оперативной памяти.

Определив команды, компилятор ***вставляет команды опережающего чтения*** и парную команду ***контроля опережающего чтения*** (*speculative check*).

Одновременно компилятор ***переставляет команды*** таким образом, чтобы ЦП мог их обрабатывать параллельно.

Когда ЦП встречает команду опережающего чтения, он пытается выбрать данные из памяти.
Если данные еще не готовы, то процессор получает сообщение об ошибке.
Система откладывает этот «сигнал тревоги» до момента прихода процесса в точку «***команда проверки опережающего чтения***».

Если к этому моменту все предшествующие подпроцессы завершены и данные считаны, то *обработка продолжается*.

Если имеются незавершенные подпроцессы, то вырабатывается *сигнал прерывания*.

1.7.3 Буфера прогнозирования условных переходов

Простейшей схемой динамического прогнозирования направления условных переходов является *буфер прогнозирования условных переходов* (***branch-prediction buffer***) или *таблица "истории" условных переходов* (***branch history table***).

Буфер прогнозирования условных переходов - это небольшая память, адресуемая *с помощью младших разрядов адреса команды перехода*.
Каждая ячейка этой памяти содержит **n-бит**, которые говорят о том, был ли предыдущий переход выполняемым или нет.

Рассмотрим диаграмму состояний *двухбитовой схемы прогнозирования* направления перехода, показанную на рисунке 1.3.

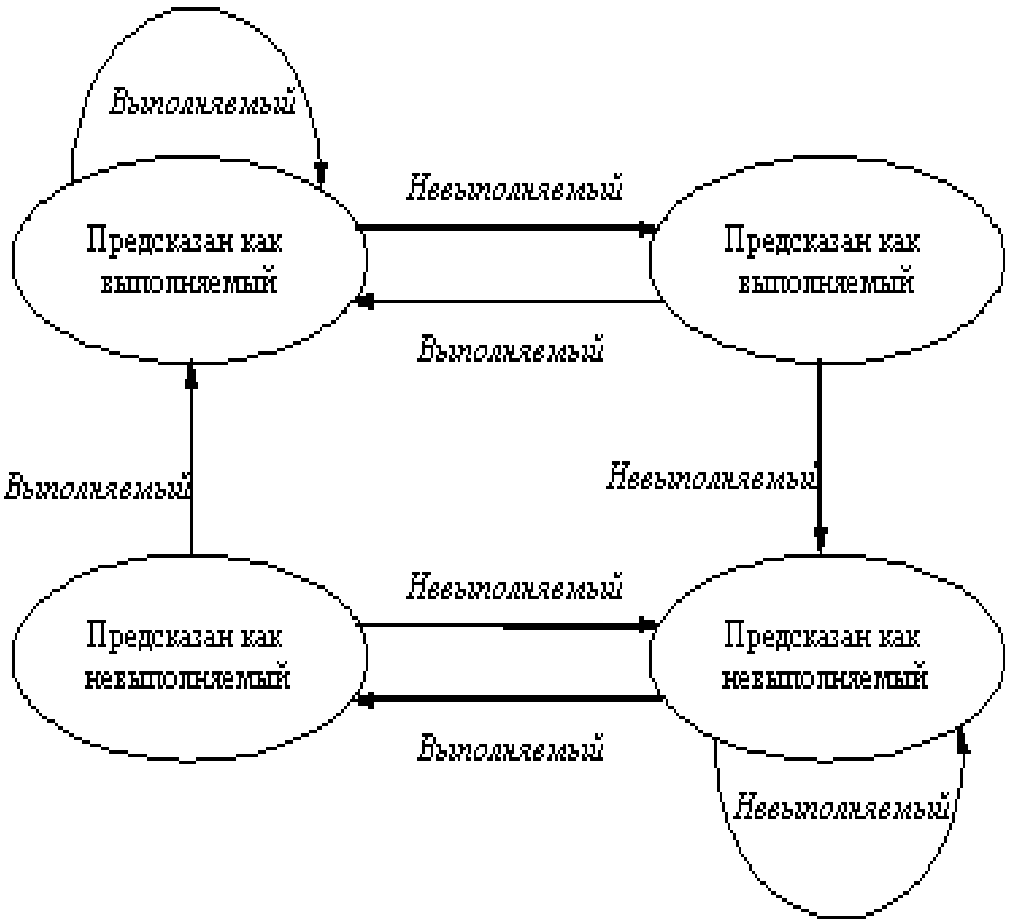

Рисунок 1.3 - Диаграмма состояния двухбитовой схемы прогнозирования

Для общей схемы прогнозирования счетчик принимает значения ***от 0 до 2^n - 1***.

Тогда **схема прогноза** будет следующей:

- Если значение счетчика **больше или равно** 2^{n-1} (точка на середине интервала), то переход **прогнозируется как выполняемый**.
- Если направление перехода предсказано правильно, к значению счетчика добавляется единица (если только оно не достигло максимальной величины); если прогноз был неверным, из значения счетчика вычитается единица.
- Если значение **счетчика меньше**, чем 2^{n-1}, то переход прогнозируется ***как невыполняемый***: *Если направление перехода предсказано правильно*, из значения счетчика вычитается единица (если только не достигнуто значение 0); *Если прогноз был неверным*, к значению счетчика добавляется единица.

Исследования n-битовых схем прогнозирования показали, что:

- *двухбитовая схема* работает почти также хорошо, как ***n***-битные;
- *поэтому*, в большинстве систем, применяются ***двухбитовые*** схемы прогноза, а не **n**-битовые.

В настоящее время, **Буфер прогнозирования переходов** реализуется в виде небольшой специальной кэш-памяти.

Для набора оценочных тестов SPEC-89 *2-х битный буфер прогнозирования переходов с 4096 строками* дает точность прогноза *от 99% до 82%*.

Замечание

Считается, что буфер емкостью **4К строк** является очень большим.
Буферы меньшего объема дают худшие результаты.

1.7.4 Один конвейнер хорошо, а два лучше

Сначала, как сдвоенные, так и обычные конвейеры использовались только в RISC-компьютерах. У процессора ***i386*** и его предшественников конвееров не было.

Конвейеры в процессорах компании ***Intel*** появились, только начиная с модели ***i486***:

- *Процессор* ***i486*** имел один пятиступенчатый конвейер,
- *Pentium* — два таких конвейера.

Приблизительную схему для ***Pentium*** можно отобразить рисунком 1.4.

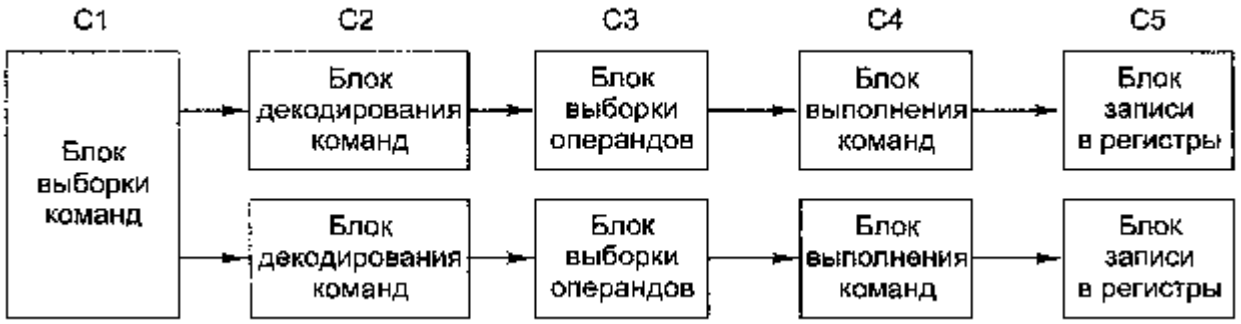

Рисунок 1.4 - Схема конвейеров процессора Pentium

Главный конвейер (***u-конвейер***) мог выполнять произвольные команды.
Второй конвейер (***v-конвейер***) мог выполнять только простые команды с целыми числами, а также ***одну простую команду с плавающей точкой*** (FXCH).

Процессор Pentium содержал *особые компиляторы*, которые объединяли совместимые команды в пары и могли порождать программы, выполняющиеся быстрее, чем в предыдущих версиях.

Измерения показали, что программы, в которых применяются операции с целыми числами, при той же тактовой частоте на Pentium выполняются почти **в два раза быстрее**, чем на ***i486***.

Замечание

Переход к четырем конвейерам возможен, но требует громоздкого аппаратного обеспечения.

1.8 Суперскалярная архитектура

Ранее было показано, что конвейерная обработка позволяет значительно увеличить скорость обработки команд процессором.
Увеличение числа конвейеров, хотя и увеличивает скоростные характеристики процессора, но сталкивается с серьезными техническими проблемами.

Имеется другой подход, который предполагает *наличие одного конвейера с большим количеством функциональных блоков*.

Для такого подхода, ***в 1987 году***, был введен термин *суперскалярная архитектура*.

Суперскалярная архитектура вычислительного ядра, использует *несколько декодеров команд*, которые могут нагружать работой множество исполнительных блоков.

При этом, *планирование исполнения потока команд* является динамическим и осуществляется самим вычислительным ядром.
Если команды *не противоречат друг другу* и *не зависят одна от результата другой*, то такое устройство может осуществить *параллельное выполнение команд*.

В суперскалярных системах решение о запуске инструкции на исполнение принимает сам вычислительный модуль. Это **требует много ресурсов**.
Данная идея нашла воплощение в компьютере CDC 6600.
Этот компьютер вызывал команду из памяти каждые **100 нс** и помещал ее в один из 10 функциональных блоков для параллельного выполнения.
Пока команды выполнялись, центральный процессор вызывал следующую команду.

Со временем, понятие «суперскалярный» несколько изменилось.
Теперь, суперскалярными называют процессоры, способные *запускать несколько команд*, зачастую от четырех до шести, *за один тактовый цикл*.

Поскольку в процессорах этого типа, как правило, предусматривается один конвейер, его устройство обычно соответствует рисунку 1.5.

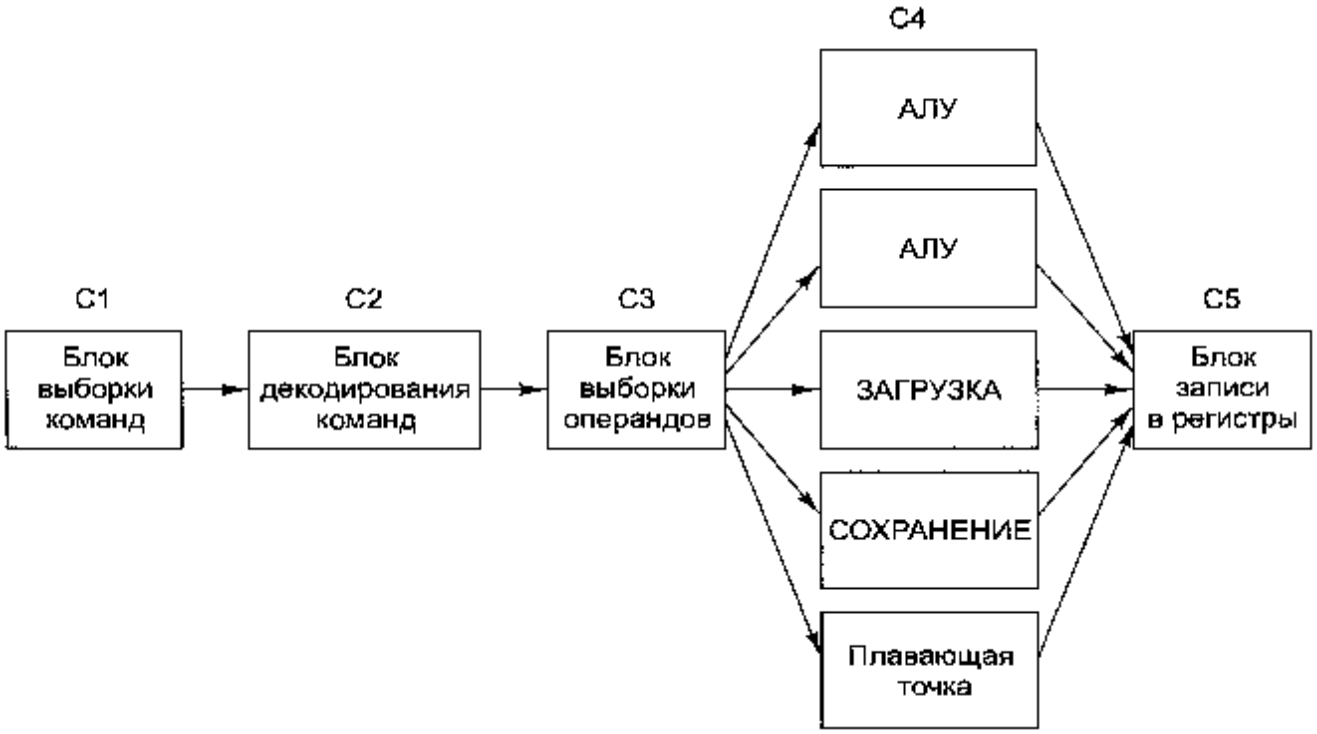

Рисунок 1.5 - Суперскалярная архитектура

В свете *такой терминологической динамики*, на сегодняшний день, можно утверждать, что компьютер 6600 не был суперскалярным с технической точки зрения — ведь за один тактовый цикл в нем запускалось не больше одной команды.
Однако, при этом был достигнут аналогичный результат - команды запускались быстрее, чем выполнялись.

Замечание

На самом деле, разница в производительности между ЦП с циклом в 100 нс, передающим за этот период по одной команде четырем функциональным блокам, и ЦП с циклом в 400 нс, запускающим за это время четыре команды, трудноуловима.

В суперскалярных процессорах соблюдается *принцип превышения скорости запуска над скоростью выполнения*.
При этом, рабочая нагрузка распределяется между несколькими функциональными блоками.

Замечание

На выходе ступени 3 команды появляются значительно быстрее, чем ступень 4 способна их обрабатывать.
Если бы на выходе ступени 3 команды появлялись каждые 10 нс, а все функциональные блоки делали свою работу также за 10 нс, то на ступени 4 всегда функционировал бы только один блок, что сделало бы саму идею конвейера бессмысленной.

В действительности, большинству функциональных блоков ступени 4 (точнее, обоим блокам доступа к памяти и блоку выполнения операций с плавающей точкой) для обработки команды требуется значительно больше времени, чем занимает один цикл.

Как видно, из рисунка 1.5, на ступени 4 может быть несколько АЛУ.

1.9 VLIW процессоры

Аббревиатура (***VLIW, Very Long Instruction Word***) известна ***с начала 80-х годов***, из ряда университетских проектов.
Она соответствует архитектуре процессоров с *командными словами сверхбольшой длины* или со **сверхдлинными командами**.

VLIW - это набор команд, организованных наподобие горизонтальной микрокоманды в микропрограммном устройстве управления.
Идея *VLIW* базируется на том, что задача эффективного планирования параллельного выполнения нескольких команд *возлагается на «разумный» компилятор*.

Такой компилятор, вначале *исследует исходную программу*, с целью обнаружить все команды, которые могут быть выполнены одновременно, причем так, *чтобы это не приводило к возникновению конфликтов*.
В процессе анализа, компилятор может даже частично имитировать выполнение рассматриваемой программы.
Затем, компилятор пытается *объединить команды в пакеты*, каждый из которых рассматривается так ***одна сверхдлинная команда***.

Объединение нескольких простых команд в одну сверхдлинную производится по следующим правилам:

- *количество простых команд*, объединяемых в одну команду сверхбольшой длины, равно числу имеющихся в процессоре функциональных (исполнительных) блоков (ФБ);
- *в сверхдлинную команду* входят только такие простые команды, которые исполняются разными ФБ, то есть обеспечивается одновременное исполнение всех составляющих сверхдлинной команды.

Длина *сверхдлинной команды* обычно составляет ***от 256 до 1024 бит***.

Такая метакоманда содержит несколько полей (по числу образующих ее простых команд).
Каждое поле описывает операцию для конкретного функционального блока.

Сказанное можно иллюстрировать рисунком 1.6:

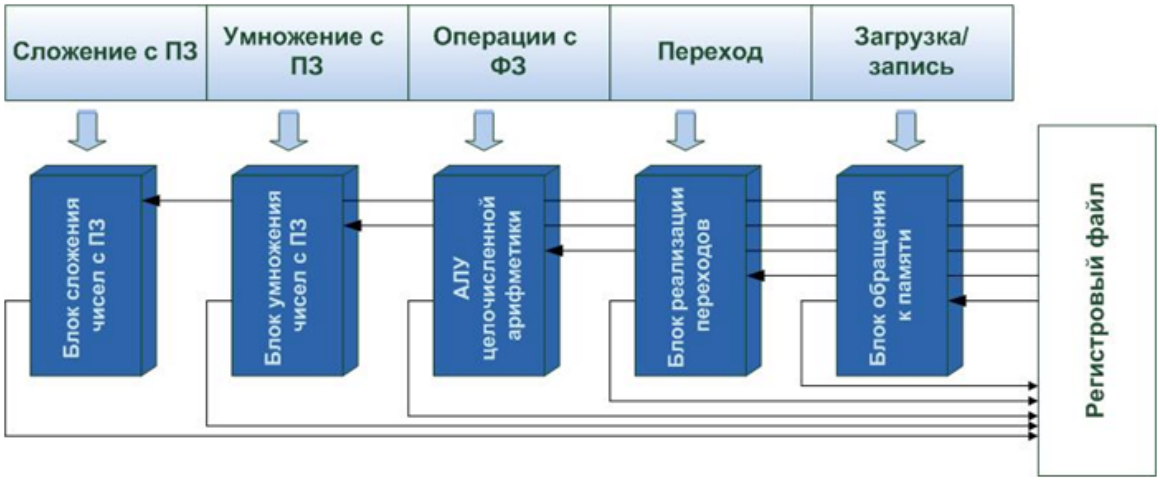

Рисунок 1.6 - Архитектура VLIW

VLIW-архитектуру можно рассматривать как *статическую суперскалярную архитектуру*.
Здесь имеется в виду, что ***распараллеливание кода*** производится *на этапе компиляции*, а не динамически, во время исполнения.
В такой, сверх-длинной команде, *исключена возможность конфликтов*.
Это позволяет предельно *упростить аппаратуру* ***VLIW-процессора*** и, как следствие, позволяет добиться более высокого быстродействия.

В качестве простых команд, образующих сверхдлинную, обычно используются команды RISC-типа.
Поэтому, архитектуру ***VLIW*** иногда называют *пост-RISC-архитектурой*.
Максимальное число полей в сверхдлинной команде равно числу вычислительных устройств и обычно колеблется *в диапазоне от 3 до 20*.
Все вычислительные устройства имеют доступ к данным, хранящимся *в едином многопортовом регистровом файле*.

Отсутствуют *сложные аппаратные механизмы*: предсказание переходов, внеочередное исполнение и другие.
Это дает *значительный выигрыш в быстродействии* и возможность более эффективно использовать площадь кристалла процессора.

Замечание

Подавляющее большинство цифровых сигнальных процессоров и мультимедийных процессоров *с производительностью более 1 млрд операций/с* базируется на VLIW-архитектуре.

Серьезная проблема VLIW - усложнение регистрового файла и связей этого файла с вычислительными устройствами.

Преимущества:

1. *Использование компилятора* позволяет устранить зависимости между командами до того, как они будут реально выполняться,
2. *Отсутствие зависимостей* между командами в коде, сформированном компилятором, ведет к упрощению аппаратных средств процессора и за счет этого к существенному подъему его быстродействия.
3. *Наличие множества функциональных блоков* дает возможность выполнять несколько команд параллельно.

Недостатки:

- *Требуется новое поколение компиляторов*, способных проанализировать программу, найти в ней независимые команды, связать такие команды в строки длиной от 256 до 1024 бит, обеспечить их параллельное выполнение.
- *Компилятор должен учитывать* конкретные детали аппаратных средств.
- *При определенных ситуациях* программа оказывается недостаточно гибкой.

Основные сферы применения:

- ***VLIW***-процессоры пока еще распространены относительно мало.
- цифровые сигнальные процессоры и вычислительные системы, ориентированные на архитектуру IA-64.
- *В России*, VLIW-концепция была реализована в суперкомпьютере ***Эльбрус 3-1*** и получила дальнейшее развитие в его последователе - *Эльбрус-2000 (Е2к)*.
- К VLIW можно причислить семейство сигнальных процессоров ***TMS320C6x*** фирмы Texas Instruments.
- **С 1986** года ведутся исследования VLIW-архитектуры в IBM.
- **В начале 2000 года**, фирма ***Transmeta*** заявила процессор ***Crusoe***, представляющий собой программно-аппаратный комплекс. В нем команды микропроцессоров серии ***x86*** транслируются в слова ***VLIW*** длиной 64 или 128 бит. Оттранслированные команды хранятся в кэш-памяти, а трансляция при многократном их использовании производится только один раз. Ядро процессора исполняет элементы кода в строгой последовательности.

1.10 EPIC архитектура

Архитектура **EPIC** является *дальнейшим развитием VLIW*.

EPIC (*Explicitly Parallel Instruction Computing*) – микропроцессорная архитектура с явным параллелизмом команд.
Термин введен, ***в 1997 году***, альянсом ***HP*** и ***Intel*** для разрабатываемой *архитектуры Intel Itanium*.

EPIC позволяет микропроцессору выполнять инструкции параллельно, опираясь на работу компилятора.
Удалось выявить возможность параллельной работы инструкций при помощи специальных схем.
В теории, это помогло упростить масштабирование вычислительной мощности процессора без увеличения тактовой частоты.

Современным типичным представителем архитектуры **EPIC** является *процессор IA-64*.
Процессор IA-64 содержит:

- 128 **64-разрядных** регистров общего назначения (РОН);
- 128 **80-разрядных регистров** с плавающей запятой;
- 64 **однобитовых регистра** предикатов.

Архитектура EPIC имеет следующие *основные особенности*:

- *поддержка явно выделенного компилятором параллелизма*. Формат команд имеет много общего с архитектурой с длинным командным словом - параллелизм так же явно выделен. К каждой длинной команде (128 бит) прилагается небольшой (3-5 битный) ярлык, который специфицирует формат команды.
- *наличие большого регистрового файла*.
- *наличие предикатных регистров*. Множественный предикатный флаговый регистр позволяет избавится от паразитных связей по управлению из-за регистра флагов.
- *спекулятивная загрузка данных*, позволяющая избежать простоев конвейера при загрузке данных из оперативной памяти.

1.11 Архитектуры x86, x86-64, IA-32, IA-64

x86 (*Intel 80x86*) – архитектура процессора с набором команд, которая была реализована в процессорах компании *Intel*.

Традиционно, для этой архитектуры использовались обозначения: 80086, 80186, 80286 (i286), 80386 (i386), 80486 (i486).
Со временем, этот набор команд постоянно расширялся и сохранял совместимость с предыдущими поколениями.
Со временем, также появились другие производители процессоров с таким же набором команд: AMD, VIA, Transmeta, IDT и другие.

Когда процессора стали 32-битными, ***Intel*** стала применять обозначение ***IA-32***.
Стали появляться и другие обозначения, подчеркивающие совместимость с ***x86*** по набору команд, например:

- x86-64 - фирма ***AMD*** опубликовала свои первые предварительные спецификации 64-битного процессора, совместимого с ***x86***.
- *X64* - официальное название версий операционных систем Windows и Solaris, а также - название архитектур фирм Microsoft и Sun Microsystems.

1.11.1 Систематизация обозначений

Со временем, чтобы избежать разночтения, корпорация ***Intel*** разработала обозначения своих процессоров, которые ориентируются на ***различные секторы рынка***:

- *Архитектура IA-32* предназначена для выполнения массовых 32-разрядных приложений на ПК **начального уровня**. Она реализована на семействах процессоров:
  1) Intel Celeron и Intel Pentium (в корпусе FC-PGA2);
  2) процессорах Intel, использующих технологии ультранизкого напряжения питания;
  3) Intel Core Duo – двухядерные процессора технологии 65 нм.
- *Архитектура Intel 64* предназначена для современных ПК и серверов среднего уровня, оптимизированных для выполнения 64-разрядных приложений. Эта архитектура реализована в процессорах:
  1) Intel Xeon;
  2) Intel Core 2 Duo.
- *Архитектура набора команд IA-64* реализована в семействе процессоров Intel Itanium.

Замечание

Обычно путают архитектуру Intel 64 и IA-64.
Следует помнить, что **Intel 64 и IA-64 -** *это совершенно разные, несовместимые друг с другом, микропроцессорные архитектуры.*

Представители ***Intel 64*** - это Pentium 4 (последние модели), ряд моделей семейства Celeron D, семейство Core 2, Core i3, Core i5, Core i7 и некоторые модели Intel Atom.

Представители ***IA-64*** — это семейства Itanium и Itanium 2.

1.11.2 Архитектура IA64

IA-64 (*Intel Architecture-64*) — 64-битная аппаратная платформа, разработанная совместно компаниями ***Intel*** и ***Hewlett Packard***.
Это – одновременно:

- *микропроцессорная архитектура*;
- соответствующая *архитектура набора команд*.

IA-64 основана на ***VLIW*** архитектуре или, в терминах Intel: ***EPIC***, поскольку обеспечивает вычисления с явной параллельностью инструкций.
Она является несовместимой с архитектурой x86.

Явный параллелизм представлен начиная с уровня команд, управляющих одновременной работой функциональных исполнительных устройств (ФИУ).
Соответствующие "*широкие команды*" ***HP/Intel*** назвали *связками* (*bundle*).

127 87 86 46 45 5 4 0

Команда 2	Команда 1	Команда 0	Маска
41	41	41	5

Связка имеет длину 128 разрядов и включает:

- *3 поля - "слота"* для команд длиной 41 разрядов каждая,
- *5-разрядное поле шаблона*.

Предполагается, что команды связки могут выполняться *параллельно разными ФИУ*.
Возможные взаимозависимости, препятствующие параллельному выполнению команд связки, отражаются в ***поле шаблона (маски)***.
Шаблон указывает, какого типа команды находятся в слотах связки.
В общем случае, команды одного типа могут выполняться в более чем одном типе ФИУ.
Шаблоном задаются *остановки*.
Остановки определяют слот команды, требующей ожидания ее завершения.

Порядок слотов в связке - возрастание справа-налево, что отвечает и порядку байт - *little endian*.

Данные в памяти могут располагаться и ***в режиме*** слева-направо: *big endian*.

Режим устанавливается специальным разрядом в регистре маски пользователя.

Типы команд** и **исполнительных устройств

Тип команд	***Тип исполнительного устройства***	***Описание команд***
A	I или M	Целочисленные, АЛУ
I	I	Целочисленные неарифметические
M	M	Обращение в память
F	F	С плавающей запятой
B	B	Переходы
L+X	I	Расширенные

Замечание

*При использовании ассемблера, **остановки** отмечаются двумя подряд знаками "точка с запятой" - ";;".*

Последовательность команд от остановки до остановки *(или выполняемого перехода) называется группой команд*.
Она начинается с заданного адреса команды (адрес связки плюс номер слота) и включает все последующие команды - с увеличением номера слота в связке, а затем и адресов связок, пока не встретится остановка.

1.12 Процессоры Itanium

В 1989 году, корпорация ***Intel*** приступила к разработке нового семейства процессоров, которое получило название ***Itanium***.

Семейство процессоров **Itanium** является **64-разрядным** и созданым на базе архитектуры ***EPIC***.
Разработка велась совместными усилиями компаний ***Intel*** и ***HP***.
Предполагалась обратная совместимость процессоров ***Itanium*** с процессорами ***Intel x86*** и ***HP PA-RISC***.

В 2001 году, первая партия процессоров ***Itanium*** поступила в продажу.
Производство этого семейства было прекращено ***в июле 2002 года*** одновременно с выходом нового процессора: ***Itanium 2***.
На рисунке 1.7, показана заявленная архитектура первого варианта процессора.

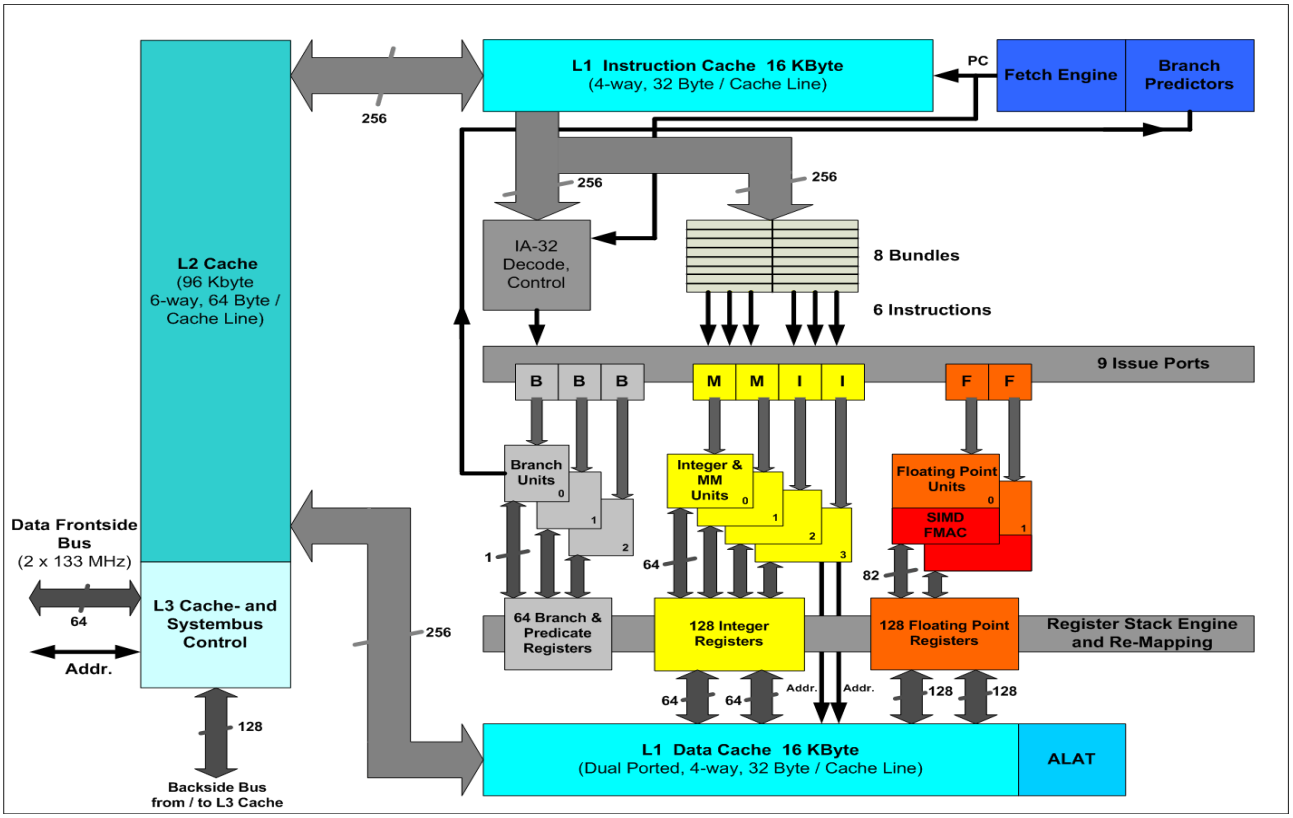

Рисунок 1.7 — Архитектура Intel Itanium

Первая реализация процессора оказалась не столь удачной, как расчитывали разработчики. Причин этому было несколько:

- *большие задержки* (латентность) в кеше 3-го уровня ОЗУ, что сводило не нет все преимущества массово-параллельной обработки, в режиме эмуляции ***x86***;
- *проблемы конкуренции* с корпорацией AMD, вызванные задержкой выпуска, по сравнению с запланированым на 1998-1999 год.

В 2002 году, ***Intel*** совместно с ***HP*** выпустили процессор Itanium2.
В Itanium 2, уже применяются *64-разрядные инструкции* непосредственно на аппаратном уровне, чего не было на процессорах ***Itanium***.
В Itanium 2 стало больше функциональных устройств:

- *Itanium 2* имеется *6 АЛУ*, в то время как Itanium может использовать лишь 4 за такт;
- *имеется 4 порта памяти*, позволяющие по 2 целочисленных загрузки и сохранения за такт, в то время как в Itanium есть только 2 порта;
- *может выполнить одну SIMD инструкцию* с плавающей точкой за такт, в то время как Itanium — две;
- *при определенных условиях* Itanium 2 может перенаправлять выполнение инструкций на непрофильные функциональные элементы;
- *при обработке операций* Itanium 2 учитывает многократно повторяющиеся операции.

Архитектура Itanium 2 показана на рисунке 1.8.

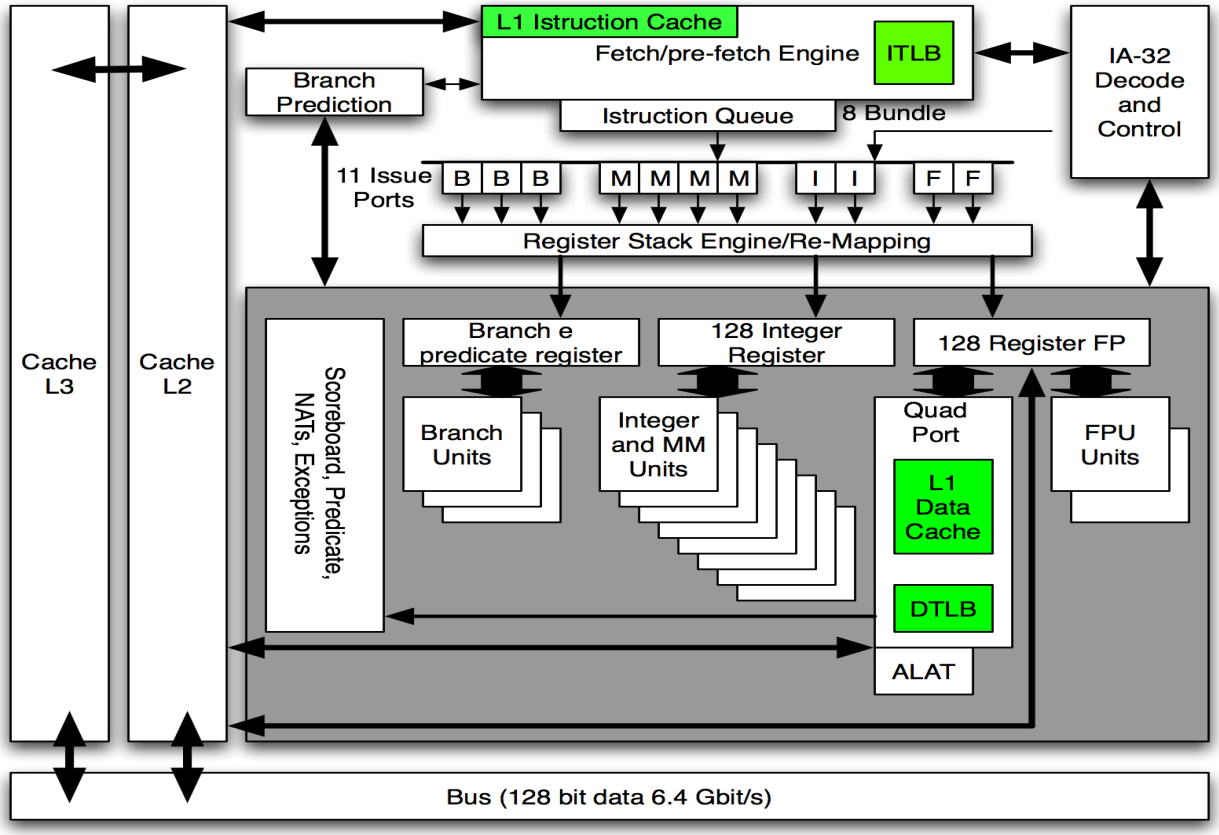

Рисунок 1.8 — Архитектура процессора Itanium2

В большинстве случаев, латентности (реальное время отклика) у ***Itanium 2*** такие же или меньше, чем у ***Itanium***.
В случае промаха, при чтении из кэша, в Itanium конвейер задерживается на 10 тактов, в Itanium 2 - на 8.
В случае неудачи, у Itanium происходит вызов обработчика ОС, а Itanium 2 обычно обходится без этого, разрешая проблемы на аппаратном уровне, что *снижает накладные расходы примерно с 200 до 18 тактов*.

Как у Itanium, так и у Itanium 2 *кэш трехуровневый*.
У ***Itanium 2*** *размер строки вдвое больше, чем у Itanium*.
TLB у ***Itanium 2*** *не только больше*, но и состоит из двух уровней против одного у Itanium.
Кэш третьего уровня у Itanium 2 располагается *на самом чипе*.
Конвейер в Itanium 2 стал на 2 ступени короче по сравнению с Itanium. Функции FET и WLD были перенесены на соседние ступени.
За один такт на конвейер могут быть поданы до 6 инструкций, максимум 32 байта.
За первыми стадиями находится буфер инструкций из 8 связок, сглаживающий и промахи чтения из памяти, и простой финальных стадий в результате обработки ветвлений первыми.

Процессоры семейства Itanium 2 поддерживают *выполнение кода и для архитектур IA-32*. Поддержка включает в себя возможность одновременной работы с приложениями, как созданными специально для Itanium, так и для IA-32 и на системах с одним процессором, и на многопроцессорных.

В ноябре **2007 года**, ***Intel*** переименовала серию процессоров ***Itanium 2*** обратно в ***Itanium***.

1.13 Процессоры UltraSPARC

SPARC (*Scalable Processor ARChitecture***)** — масштабируемая процессорная архитектура.
Это — архитектура RISC-микропроцессоров, первоначально разработанная в ***1985 году*** компанией ***Sun Microsystems***.

UltraSPARC T1 — многоядерный микропроцессор с аппаратной поддержкой многопоточности.
Разработан *Sun Microsystems* и, до анонса 14 ноября 2005 года, известен как *Niagara*.

Процессор базируется *на RISC-архитектуре UltraSPARC Architecture 2005 specification с поддержкой набора команд SPARC v9.*

Выпускается в различных модификациях, отличающихся:
1) *тактовыми частотами (1 — 1.4 ГГц),*
2) количеством ядер (4, 6 и 8 ядер),
3) ядра с аппаратной поддержкой четырёх потоков (чередование 4 «лёгких» процессов — Light Weight Processes, LWP) на ядро.

Процессор UltraSPARC T1 представляет собой производительный, *высокоинтегрированный суперскалярный процессор*, реализующий 64-битовую архитектуру **SPARC-V9**.
В его состав входят:
- устройство предварительной выборки и диспетчеризации команд,
- целочисленное исполнительное устройство,
- устройство работы с вещественной арифметикой и модуль графики,
- устройства управления памятью, загрузки/записи и управления внешней кэш-памятью,
- модули управления интерфейсом памяти и кэш-памяти команд и данных.

Аппаратная архитектура процессора UltraSPARC показана на рисунке 1.9 (блок FPU и L1 Cache ядер не показаны).

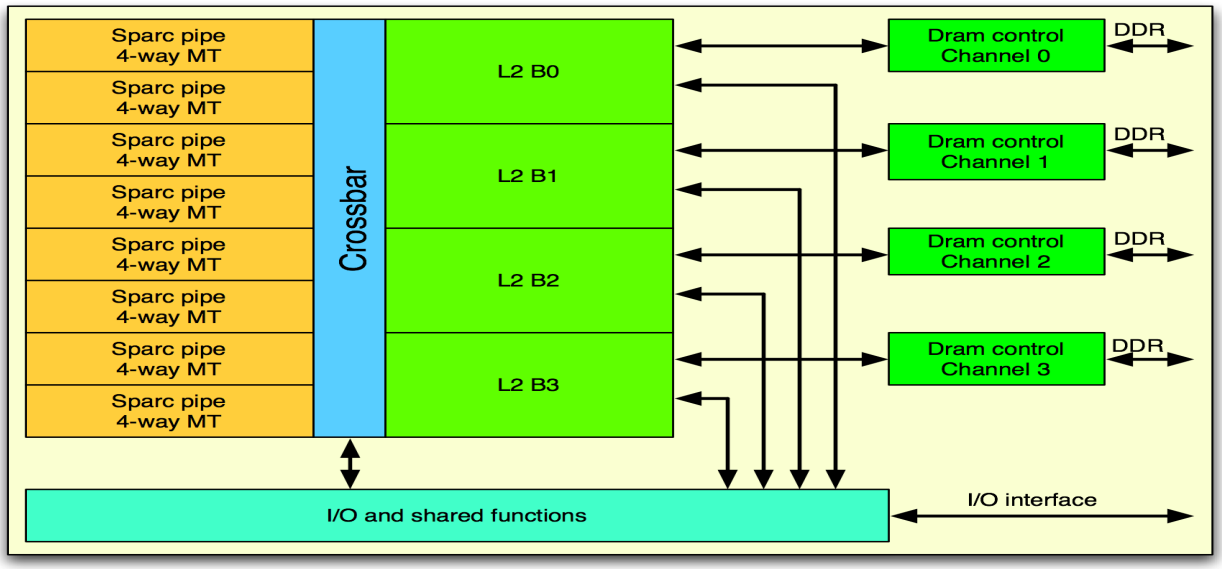

Рисунок 1.9 — Архитектура процессора UltraSPARC

UltraSPARC T1 представляет собой кристалл, на котором размещаются:

- *до 8 ядер* SPARC V9 с 16 КБайт L1 кэша инструкций,
- *8 КБайт L1* кэша данных,
- *блок операций* с плавающей точкой (FPU), объединяемых *внутрипроцессорным коммутатором (crossbar)* с пропускной способностью *132 ГБайт/сек*.

К коммутатору присоединены 4 банка L2 кэша суммарной емкостью 3 Мбайт, которые разделяются всеми процессорными ядрами.
Каждый из банков обслуживается контроллером памяти DDR-II DRAM.
Используются *144-битные интерфейсы*.
Агрегированная пиковая пропускная способность контроллеров - *25 ГБайт/сек*.
В качестве интерфейса ввода-вывода используется 128-битная шина J-Bus interface (JBI).

Логическая архитектура UltraSPARC T1 показана на рисунке 1.10.

Имеется устройство *предварительной выборки и диспетчеризации команд*.
Это устройство (PDU) обеспечивает:

1) выборку команд в буфер команд;
2) окончательную их дешифрацию команд;
3) *группировку и распределение* для параллельного выполнения в конвейерных функциональных устройствах процессора;
4) *буфер команд* емкостью 12 инструкций позволяет согласовать скорость работы памяти со скоростью обработки исполнительных устройств процессора;
5) Команды могут быть *предварительно выбраны из любого уровня иерархии памяти*, например, из кэш-памяти команд (***I***-кэша), внешней кэш-памяти (***E***-кэша) или из основной памяти системы.

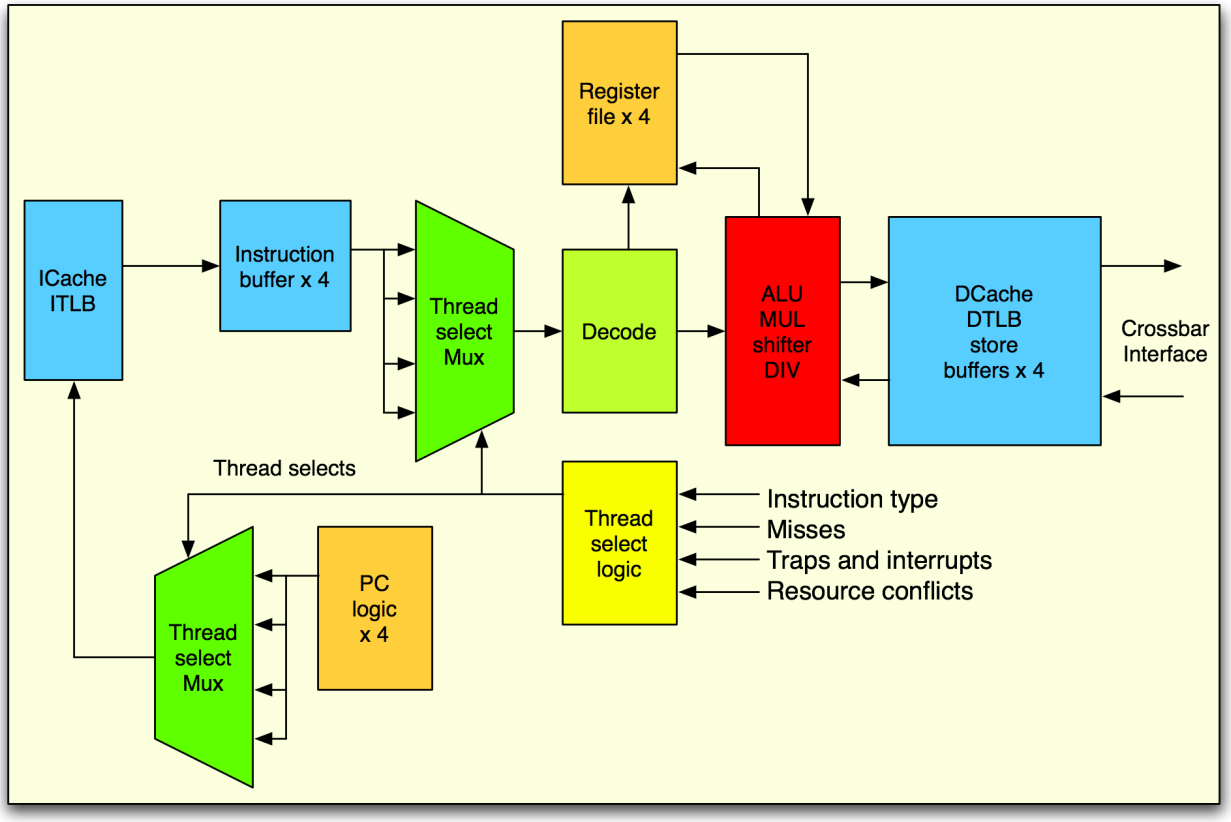

Рисунок 1.10 — Логическая архитектура процессора UltraSPARC T1

В процессоре реализована *схема динамического прогнозирования* направления ветвлений программы, основанная на двухбитовой истории переходов.
Для реализации этой схемы с каждыми двумя командами в I-кэше связано специальное поле, хранящее *двухбитовое значение прогноза*.

Таким образом, UltraSPARC T1 позволяет хранить информацию о направлении *2048 переходов*, что на сегодняшний день превышает потребности многих современных прикладных программ.
Поскольку направление перехода может меняться каждый раз, когда обрабатывается соответствующая команда, состояние двух бит прогноза должно каждый раз модифицироваться для отражения реального исхода перехода. Эта схема особенно эффективна при обработке циклов.

Кроме того, в процессоре UltraSPARC T1 с каждыми четырьмя командами в I-кэше связано *специальное поле, указывающее на следующую строку кэш-памя-ти*, которая должна выбираться вслед за данной.
Использование этого поля позволяет осуществлять выборку командных строк в соответствии с выполняемыми переходами, что обеспечивает для программ с большим числом ветвлений практически ту же самую пропускную способность команд, что и на линейном участке программы.

Замечание

Способность быстро выбрать команды по прогнозируемому целевому адресу команды перехода является очень важной для оптимизации производительности суперскалярного процессора и позволяет UltraSPARC-1 эффективно выполнять "по предположению" (*speculative*) достаточно хитроумные последовательности условных переходов.

Используемые в UltraSPARC-1 механизмы динамического прогнозирования направления и свертки переходов сравнительно просты в реализации и обеспечивают высокую производительность.

По результатам контрольных испытаний в UltraSPARC T1 предсказываются успешно:

- *88% переходов* по условиям целочисленных операций,
- *94% переходов* по условиям операций с плавающей точкой.

2 Лабораторная работа №4

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

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

- *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;
- *архитектура* современных ***микропроцессоров*** имеет тенденцию влючения в себя множества функциональных элементов, которые сами претендуют на роль отдельных процессоров, с возможностью самостоятельно обеспечивать вычислительные процессы компьютера (многоядерные архитектуры);
- *вычислительные элементы* современных ***микропроцессоров*** имеют тенденцию к самостоятельному программированию (микропрограммированию), что порождает необходимость их *асинхронного 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:

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

Непосредственно, в данной лабораторной работе, мы:

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

2.1 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 извлеч 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.

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 ключем ***-html***, например:

```
lshw -html > hw.html
```

После этого, файл ***hw.html*** можно просмотрет в браузере, как показано на рисунке 2.1.

Имеется и другой способ — запустить утилиту командой:

```
sudo lshw -X
```

тогда запустится графическая оболочка утилиты.

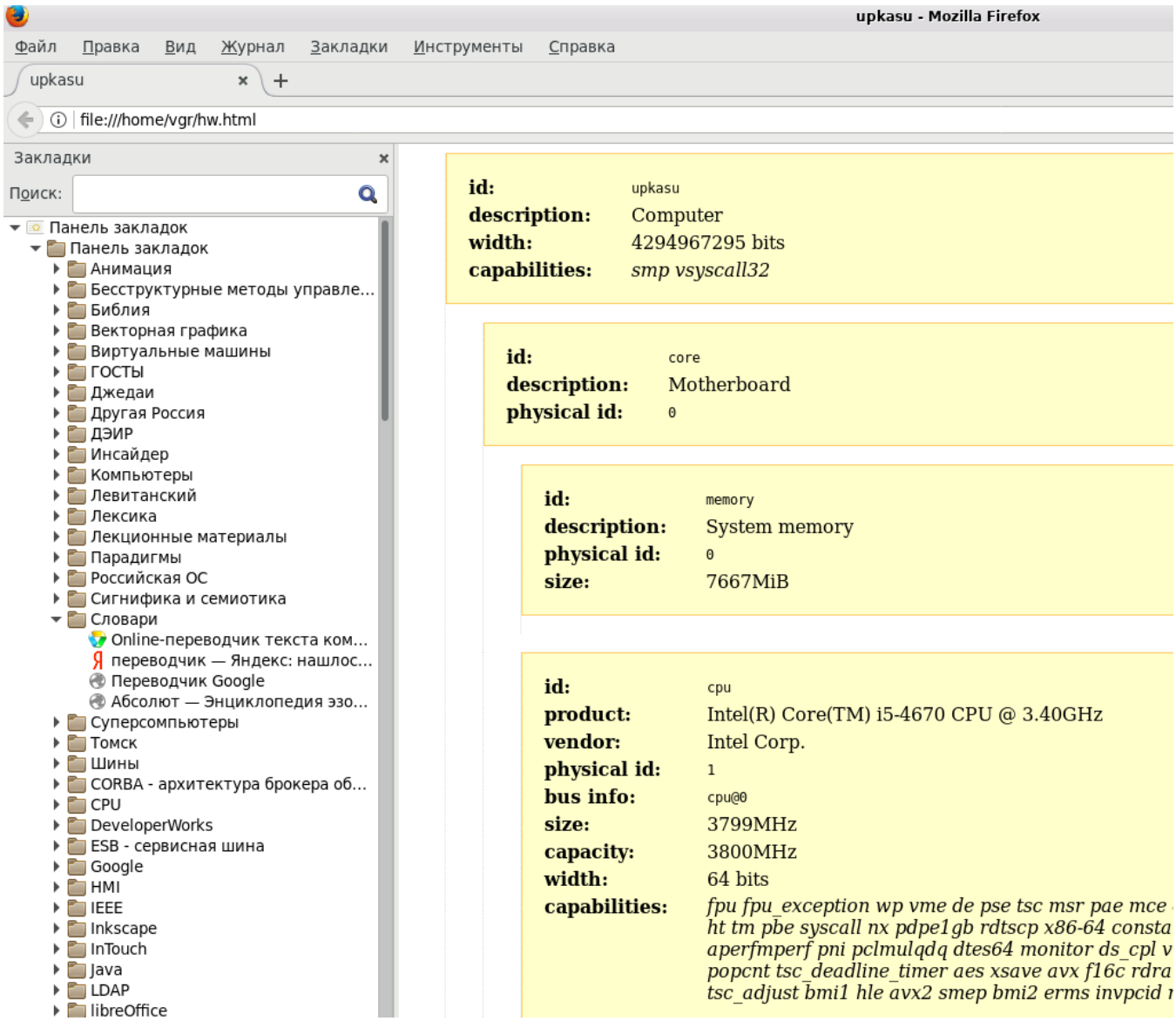

Рисунок 2.1 — Перечень аппаратного обеспечения ЭВМ, представленный html-страницей, полученной с помощью утилиты lshw

Задание

Изучить и отобразить в личном отчете:

- перечень компонентов рабочей станции, на которой работает студент;
- основные характеристики, выделенных компонент ЭВМ.

2.2 Характеристики виртуального терминала

Обычная рабочая станция (компьютер, ЭВМ) имеют клавиатуру, дисплей и графический адаптер, обеспечивающий вывод текстовой и/или графической (пиксельной) информации на дисплей.
Как уже известно, из курса «*Современные операционные системы*», перечисленные устройства (клавиатура, дисплей и графический адаптер) системно объединены понятием консоли (терминала).
Современное общесистемное программное обеспечение, которое является частью любого комплекса ЭВМ, расширено понятием *виртуальных терминалов*, обеспечивающих взаимодействие с самим комплексом.

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

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

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

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

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

Чтобы выяснить, на каком терминале работает пользователь, необходимо использовать утилиту ***tty***, которая выведет на экран полное имя устройства терминала.

Когда пользователь вошел 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 интерпретатора ***shell***, то:

- сами символы сохраняются во внутреннем буфере терминала, а на экран выводится «*эхо*» нажатых клавиш;
- 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 ЭВМ на уровне командных строк или на уровне запуска файла сценария команд.

Замечание

Терминал, на котором пользователь провел вход в систему посредством утилиты ***login***, является для всех программ (процессов) пользователя *управляющим терминалом*, что накладывает на выполнение всех его программ следующие ограничения:

- утилита ***login***, в процессе своей работы, проводит идентификацию и авторизацию пользователя, создает сессию и среду для выполнения его программ, а также запускает интерпретатор ***shell***;
- в случае закрытия терминала, все программы (процессы) пользователя уничтожаются и сессия закрывается; нормальное закрытие терминала осуществляется последовательностью команд ***exit***;
- каждый процесс, запущенный в терминале, имеет системный 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 можно найти в файле <*bits/termios.h*>. При большом желании изучить все детали, следует начать с руководства ***man termios***.

В пределах данной лабораторной работы, мы ограничемся рассмотрением набора функций, реализованных в проекте ***avk_tty***.

Цель проекта — демонстрация ПО, обеспечивающего:

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

Программное обеспечение данного проекта будет использовано в последующих лабораторных работах.

Исходный текст проекта 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 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;
}


/**
 * 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

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

Замечание

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

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

- запустить терминал и перейти в директорию ***~/bin*** командой: ***cd ~/bin*** ;
- сделать символьную ссылку командой:

```
ln -s ~/workspaceC/avk_tty/Debug/avk_tty  avk_tty
```

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

2.3 Классическое устройство мыши

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

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

Замечание

Устройство ***mouse0*** первым устройством мыши, которое увидел и создал демон ***udevd***. Следующее подключенное устройство мыши называется ***mouse1*** и так далее.

Само устройство ***mouse0*** является символьным устройством доступным для чтения и записи владельцу ***root*** и группе ***input***.
Поскольку пользователь ***upk*** включен в группу ***input***, то не имеется никаких препятствий к использованию этого устройства.

Как компонет ЭВМ, устройство мыши предназначено для асинхронного взаимодействия ним.
Как источник информации, это устройство выдает состояние трех кнопок и относительное перемещение самого устройства в системе координат (***x,y***).
Величины этой информации поддерживаются стандартным драйвером ОС и сохраняются в структуре ***input_event***.
Прикладные возможности такого устройства наиболее просто демострируются исходным текстом проекта ***avk_mouse***, представленного 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 - состояния 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 правая кнопка.
            printf("\nАбсолютные координаты TOP_LEFT = %i %i\n\n",
                        absolute_x, absolute_y);
    }

    return 0;
}
```

Задание

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

2.4 Устройство фреймбуфера

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

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***, предназначенные для ***IBM PC***-совместимых компьютеров.

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 жестких ограничений, присущих *доступу к общим разделяемым ресурсам*.
Возможности такого взаимодействия можно оценить по модифицированной подсистеме ***login***, на виртуальных терминалах ***/dev/tty1*** и ***/dev/tty2***, а также по демонстрационной программе, доступной при переключении на виртуальный терминал ***/dev/tty5***.

В среде ОС, фреймбуфер доступен как символьное устройство ***/dev/*fb0**, разрешающее чтение и запись владельцу ***root*** и группе ***video***.
Поскольку пользователь ***upk*** входит в группу ***video***, то не никаких ограничений в использовании этого устройства как в учебных целях, так и при разработке приложений в среде ОС Linux, не требующих зависимости от сложных графических подсистем ОС.
Это утверждение демонстрируется примерами использования устройства ***/dev/fb0***, при загрузке системы:

- показ логотипа кафедры АСУ на черном форне экрана, *на первом этапе* загрузки, когда еще 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 еще 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 учета следующих ограничений:

- фреймбуфер является *единственным устройством* на все виртуальные терминалы ОС и может быть открыт в каждом из них;
- фрембуфер *синхронизирован с переключениями* на каждый виртуальный терминал ОС, что имеет последствия очистки его содержимого, в момент пере-

ключения, и отсутствие восстановления содержимого, при восстановлении активности терминала;

- *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 структурах ***fb_var_screeninfo*** и ***fb_fix_screeninfo***;
- указанные структуры извлекаются из системы и устанавливаются в нее с помощью надежного 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, зеленый и синий, а четвертый байт используется графическими библиотеками для отображения прозрачности изображения.

Для учебной демонстрации работы с фреймбуфером, рассморим простейшую задачу закраски окна монитора синим цветом и отображения на этом фоне динамической картинки циклического перемещения серого прямоугольника.
Реализацию этого примера проведем в рамках проекта ***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
 ============================================================================
 */

#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>
```

```c
/**
 * 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

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

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

Подводя итог проделанной работы, мы видим:

- 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;
- изучение работы таких общедоступных устройств как *устройство мыши* и *устройство фрембуфера* имеет целью расширить теоретические представления и обеспечить практические навыки работы с отдельными компонентами компьютера, попутно демонстрируя ряд доступных технологических возможностей работы с устройствами ОС.

Хотя студенту может оказаться недостаточно полученных знаний для самостоятельной разработки системного ПО, освоенные навыки должны стать основой при выпонении последующих лабораторных работ.

В частности, 5-я и 6-я лабораторные работы полностью опираются на изученный здесь учебный материал.

3 Лабораторная работа №5

Целью данной лабораторной работы является расширение знаний и практических навыков системного программирования, на основе знаний и навыков, полученных в предыдущей лабораторной работе.

Поскольку методологической основой системного программирования является *синхронизация* (***упорядочение взаимодействия***) *асинхронно работающих компонент*, в качестве объектов взаимодействия будут рассмотрены уже изученные нами устройства виртуального терминала, мыши и фреймбуфера.

Замечание

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 компонет 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*».

3.1 Асинхронное взаимодействие на уровне виртуального терминала

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

3.1.1 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 «***Тео-***

рия 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 внедрен 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 четкое определение целевой установки, заменяя ее *набором функций*, которым должна удовлетворять система.
Именно описание набора функций состаляет содержание любого «***Технического задания*** (ТЗ)» на систему, которая по традиции продолжает называться АСУ.

3.1.2 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 абревиатуру ***SCADA***.
Тем не менее, по традиции, такие системы продолжают называться АСУ, например, «***АСУ технологическими процессами ...***».

3.1.3 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, оснащенных стандартизированными интерфесами 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 съемка (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:

- системное ПО работает с отдельными, достаточно независимыми по технологии функционирования устройствами: процессоры, клавиатура, мыш, экран монитора и другие компоненты отдельной ЭВМ или комплекса средств вычислительной техники;
- 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.

3.1.4 Раскраска экрана монитора с помощью устройства мыши

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 отчете полученных результатов.

Учебная технология программной реализации поставленной задачи:

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

3.2 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.

3.2.1 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 фуркций проекта ***avk_tty***.

Листинг 3.1 — 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;


/** ******************************************************
 * Обявления функций.
 ********************************************************/
/**
 * Функция восстановления состояния терминала,
```

```
 * которое сохранено в struct trmios oldtty.
 */
void
avk_restore_tty(avk_tty_t *p);

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

/**
 * 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);


#endif /* AVK_MONITOR_H_ */
```

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

3.2.2 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:

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

В результате указанных действий, данная компонента ПО примет вид показанный на листинге 3.2.

Листинг 3.2 — 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 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 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;
}
```

Замечание

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

3.2.3 Этап 3. Компонента ПО устройства мыши

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

- в проекте ***avk_fb_monitor*** необходимо открыть файле ***avk_mouse.c*** и скопировать туда исходный текст проекта ***avk_mouse***;
- как и на предыдущем этапе, необходимо заменить подключаемые файлы заголовков, а потом удалить функцию обработчика прерывания и ее установку в функции ***main()***;
- 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 3.3.

Замечание

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 3.3 — 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
 ==============================================================================
 */
```

```c
#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 - 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.
     */
    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()***.

3.2.4 Этап 4. Компонента ПО устройства фреймбуфера

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

- в проекте создается файл ***avk_fb.c*** в который копируется содержимое проекта ***avk_fb***;
- 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;
- далее, функция ***main()*** также преобразуется в следующие три функции: ***avk_fb_open()***, ***avk_fb_close()*** и ***avk_fb_paint()***, причем последняя функция необходима исключительно для восстановления фона экрана после переключений виртуальных терминалов.

Исходный код этой части проекта представлен на листинге 3.4.
После отладки этой части кода, описания функций также переносятся в файл заголовка ***avk_monitor.h***.

Листинг 3.4 — 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;
}
```

3.2.5 Этап 5. Компонента ПО динамического рисунка

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

- в нашем проекте открываем файл ***avk_dynamic.c***;
- затем, реализуем функцию рисования с именем ***avk_dynamic_set()***, как показано на листинге 3.5.

Листинг 3.5 — 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;
}
```

3.3 Реализация проекта avk_fb_monitor

Выполнив организацию ПО отдельных компонент нашего проекта, переходим к реализации главной части ПО, проводящей синхронизацию (композицию) всех функциональных элеменов системы.
Для этого, необходимо:

- *перейти* к ранее созданному файлу проекта: ***avk_fb_monitor.c***;
- *включить* в него заголовочный файл ***avk_monitor.h***;
- *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:

- первая часть, состоит в *инициализации* всех глобальных переменных и *открытия* используемых устройств;
- вторая часть, обычно в цикле, *собственно и реализует* алгоритм синхронизации (композитинга) решаемой задачи;
- 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 распределенные системы.

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

В пределах заданной учебной цели, вполне достаточным является уровень прикладного программирования, на котором и реализован наш проект.
Результат такой реализации представлен на листинге 3.6.

Листинг 3.6 — 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.

4 Лабораторная работа № 6

Учебной целью данной лабораторной работы является практическое освоение синхронизации параллельно работающих компонет 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***».

4.1 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.
При увеличении числа взаимодействующих компонент, резко увеличиваетя число взаимосвязей, которые необходимо учитывать в реализуемом алгоритме.
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.
Для некоторых компонет, выделенный тайм-аут может оказаться слишком маленьким, а для некоторых — слишком большим, но требования качества исполнения заставляют использовать наименьший тайм-аут.
При изменении числа взаимодействующих компонент изменяется и период времени ожидания для каждого из них, что требует усложнения алгоритма реализации для обеспечения нужного качества.

4.2 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 объемами данных, что характерно для графического способа представления информации, при условии обеспечения нужного качества.
Другой аспект этой проблематики — необходимость синхронизации работы:

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

Таким образом, задача реализации даже простейшего композитора, является сферой применения самых современных технологических приемов системного программирования.

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

Для успешного решения задачи, учтем опыт уже существующих разработок композиторов:

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

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

- отслеживают появление новых окон и удаление уже существующих;
- показ (show) и прятание (hide) окон;
- размещение фонового окна и обеспечение иерархии других (нормальных) окон;
- обеспечение режима работы с диалоговыми окнами, меню и окнами систем-

ных сообщений;

- 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, приведенной 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 своем буфере отдельной нитью*; композитор перерисовывает в свой буфер только измененую 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:

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

Замечание

Обсуждение деталей синхронизации перенесем в описательную часть самого программного обеспечения проекта.

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

- нити (***threads***);
- мютексы (***mutex***) - упрощенные средства синхронизации;
- графические средства библиотеки ***cairo***.

4.2.1 Инструментальные средства нитей

Инструментальные средства нитей (***threads***) являются стандартными для всех ОС UNIX/Linux, в редакции стандарта POSIX.

Поскольку студенты уже знакомы с этим инструментом из курса «Опрерационные системы», в качестве справки, на листинге 4.1 приведен перечень базовых функций, необходимых для реализации алгоритмов проекта.

Листинг 4.1 — 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.

4.2.2 Синхронизация 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, приведенных на листинге 4.2.

Листинг 4.2 — Базовые функции для работы с мютексами

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

/**
 * Инициализация мютекса.
 * Если второй аргумент функции равен NULL,
 * то инициализируется быстный 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 своем буфере.

Если изображение находится в процессе создания, композитор может взять ошибочные данные.
Чтобы исключить подобные ситуации, *защитим буфер композитора* мютеском, требуя, чтобы все участники генерации новых данных блокировали этот буфер до окончания своей работы и снимали блокировку, когда такая работа завершена.

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, что выполняется двумя записями целых чисел и не требует блокировки буфера композитора.

4.2.3 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 билиотеки ***cairo***.

Cairo — библиотека, которая ***с 2003 года*** применяется во многих операционных системах, включая MS Window, для отрисовки двумерной векторной графики.
К достоинствам библиотеки следует также отнести удобную работу с тестом, используя национальные языки.
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 являются объектны типа:
- ***cairo_surface_t*** — прямоугольные области, определнные на массивах данных;
- ***cairo_t*** — контекст для конкретной прямоугольной области, который можно рассматривать как дескриптор при работе с файлами.

На каждой созданной прямоугольной области (поверхности) ***surface*** можно рисовать замкнутые кривые, которые затем можно закрашивать некоторым цветом и окантовывать линиями.
Выбираемые цвета задаются из набора красный, зеленый и синий, а также можно задавать прозрачность объекта закраски.
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 фрейм-буфера*.

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

4.3 Модификация компонент взаимодействующих устройств

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 мы будем использовать исходный текст ее проекта в данной работе.
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.

Замечание

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.

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

4.3.1 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 остается прежним, поскольку добавляемый или изменяемый функционал будет реализован в рамках отдельных компонент.
Основные изменения касаются объявления структуры типа ***avk_context_t***, в которую добавляются:

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

Полный исходный текст заголовочного файла ***avk_compositor.h*** приведен на листинге 4.3:

- *добавляемые* элементы файла отмечены коментариями;
- *удаленные* элементы — просто закоментированы.

*Листинг 4.3 — 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_contect {
      /**
       * 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;
```

```
/** ******************************************************
 * 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 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_ */
```

4.3.2 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:

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

На листинге 4.4 приведен результат проведенных преобразований в ***avk_tty.c***.

Листинг 4.4 — 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 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){
      /**
       * 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 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;
}
```

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

- отсутсвуют сообщения об ошибках в самом файле;
- отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_tty.c***;
- 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.

Замечание

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***.

4.3.3 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 определенных функций:

- 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;
- 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;
- 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 4.5.

Листинг 4.5 — 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:

- отсутсвуют сообщения об ошибках в самом файле;
- отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_mouse.c***;
- 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.

Замечание

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***.

4.3.4 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 определенных функций:

- 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()***;
- функция ***avk_fb_open()***, кроме обязанности открытия и инициализации устройства фреймбуфера, дополняется функционалом создания объектов библиотеки ***cairo*** для двух окон;
- 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 4.6.

Листинг 4.6 — 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;
      }
    /**
     * Создаем cairo_surface_t - поверхность композитора.
     */
    ctx->sur = cairo_image_surface_create_for_data(ctx->data,
            CAIRO_FORMAT_ARGB32, ctx->w, ctx->h, ctx->stride);
    /**
     * Создаем 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-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;

    /**
     * Создаем 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;
}
```

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

- отсутсвуют сообщения об ошибках в самом файле;
- отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_fb.c***;
- 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.

4.3.5 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:

- *функция* ***avk_dynamic_set()***, кроме основного цикла нити, должна содержать части кода, которые создают объекты библиотеки cairo и уничтожают их,

освобождая выделенную память компьютера;

- *данная функция нити* должна быть запущена композитором, поскольку она не может запустить сама себя;
- *нить функции*, имитируя работу некоторого приложения, функционирует независимо от переключений виртуальных терминалов и записывает изображение в свой собственный буфер;
- *координаты прямоугольной области*, отображаемые нитью этой функции ***avk_dynamic_set()***, в глобальной структуре данных, должны быть относительными, указывать композитору на измененную часть изображения;
- *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;
- 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 4.7.

Листинг 4.7 — 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.
    }
  /**
   * Создаем cairo_surface_t - поверхность изображения.
   */
  ctx->surd = cairo_image_surface_create_for_data(ctx->datad,
          CAIRO_FORMAT_ARGB32, w, h, 4*w);
  /**
   * Создаем 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);    // Задаем 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);
                cairo_rectangle(ctx->crd, (kk-2)*w0, 0, w0, h);// Задаем область.
                cairo_fill(ctx->crd);   // Заполняем цветом.
                if(ctx->k0 == 0)// Если композитор исполнил предыдущие изменения.
                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); // Задаем 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:

- отсутствуют сообщения об ошибках в самом файле;
- отсутствуют сообщения об ошибках в файле ***avk_compositor.h***, касающиеся определения функций, реализованных в файле ***avk_dynamic .c***.

4.4 Реализация проекта avk_fb_compositor

Выполнив организацию ПО отдельных компонент нашего проекта, переходим к реализации главной части ПО, проводящей синхронизацию (композицию) всех функциональных элеменов системы.
Для этого, необходимо:

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

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

- первая часть, состоит в *инициализации* всех глобальных переменных и *открытии* используемых устройств;
- вторая часть, в цикле, *собственно и реализует* алгоритм синхронизации (композитинга) решаемой задачи;
- 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 инициализаци программных компонет 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 серьезные 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 освобожден 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:

- *отреагировать* на команду завершения работы, по значению переменной ***if_exit***;
- *прекратить* вывод данных в фреймбуфер, по значению переменной ***no_fb***;
- *обеспечить* полное восстановление изображения экрана, по значению переменной ***dirty***;
- *синхронизировать* свое взаимодействие с нитью динамического изображения посредством мютекса и значению переменной **k0**;
- *переписывать* в свой буфер измененную часть динамического изображения, с последующим выводом в фреймбуфер;
- *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***, приведенного на листинге 4.8.

Листинг 4.8 — 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 измененной
//    ctx.y0 = ...;           // части динамического рисунка.

      ctx.n0 = 10;            // Число прямоугольников динамического изображения.
      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.
      ctx.h0 = 40;            // Высота прямоугольника динамического изображения.
      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.

            usleep(dt);       // Засыпаем на заданный период (для экспериментов).
            /**
             * 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 устовии, 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);
                  /**
                   * Задаем 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);
                        /**
                         * Задаем 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);
```

```
                    /**
                     * Задаем 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);
                    /**
                     * Задаем 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:

- отсутствуют сообщения об ошибках в самом файле;
- 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 отчете. **Особо** следует уделить внимание сравнительному анализу полученных результатов по всем работам данной темы.

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

Список использованных источников

1. Резник В.Г. Архитектура вычислительных комплексов. Самостоятельная и индивидуальная работа студента. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 13 с.

2. Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 38 с.

Учебное издание

Резник Виталий Григорьевич

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ КОМПЛЕКСОВ

Учебно-методическое пособие предназначено для изучения темы №2 по дисциплине «Архитектура вычислительных комплексов» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Учебно-методическое пособие

Усл. печ. л. . Тираж . Заказ .
Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40