Министерство образования и науки Российской Федерации
Томский государственный университет систем управления и радиоэлектроники
(ТУСУР)

УТВЕРЖДАЮ

Проректор по учебной работе

_______________ Л.А. Боков

«___»_________ 2011 г.

УЧЕБНОЕ ПОСОБИЕ

по дисциплине «**Архитектура систем на кристалле**»

Для подготовки магистров по направлению 220600 «Инноватика»,
магистерская программа «Мультимедийные многопроцессорные системы на кристалле»
Институт инноватики, факультет инновационных технологий

Профилирующая кафедра ___кафедра «Управление инновациями»____
(наименование)

Составитель:

Ассистент кафедры УИ Н.В. Милованов

Томск 2011

Оглавление

Оглавление 2
Архитектура микропроцессоров. Введение 4
Архитектура процессоров с параллелизмом уровня команд. 12
Классификация систем на кристалле и ЦСП. Общая архитектура. 14
Введение. Операционная система GNU/Linux. Особенности 14
Специальные типы файлов. Файловые системы, монтирование 24
Командная оболочка bash. Скрипты оболочки. Переменные окружения.... 28
Процесс загрузки GNU/Linux. Модули ядра. Загрузка по сети. 49
Компиляция программ. Препроцессор, транслятор, компоновщик. 59
Компиляция программ. Препроцессор, компилятор, линкер. 63
Линковка, препроцессинг. 63
Сборка программ из исходных текстов 63
Кросс-компиляция 63
Разработка программного обеспечения для ЦСП PNX 8950 64
Модульная архитектура TSSA. 64
Использование ассемблера для ЦСП PNX 1500 64
Пространство ядра и пространство пользователя 65
Виртуальная память 65
Файлы символьных устройств 67
Перемещение данных из пространства ядра в пространство приложений и обратно 68
Модуль символьного устройства 68
Файловая система /proc 71
Линейка цифровых сигнальных процессоров фирмы TI 73
Знакомство с процессорами DaVinci TI 74
Средства разработки программного обеспечения от фирмы TI 77
Модульная архитектура XDAIS 78

Видео подсистема. Framebuffer. DirectFB 80
Оптимизация C кода и директивы компилятора при разработке для ЦСП. 81
Структура ассемблера для C64x. Использование специализированных инструкций ЦСП 83
Использование ассемблера для ЦСП C64x 83
Литература 83

Архитектура микропроцессоров. Введение

Введение. Составные части курса.

История. История развития производства процессоров полностью соответствует истории развития технологии производства прочих электронных компонентов и схем.

Первым этапом, затронувшим период с 40-х по конец 50-х годов, было создание процессоров с использованием электромеханических реле, ферритовых сердечников (устройств памяти) и вакуумных ламп. Они устанавливались в специальные разъёмы на модулях, собранных в стойки. Большое количество таких стоек, соединённых проводниками, в сумме представляли процессор. Отличительной особенностью была низкая надёжность, низкое быстродействие и большое тепловыделение.

Вторым этапом, с середины 50-х до середины 60-х, стало внедрение транзисторов. Транзисторы монтировались уже на близкие к современным по виду платам, устанавливаемым в стойки. Как и ранее, в среднем процессор состоял из нескольких таких стоек. Возросло быстродействие, повысилась надёжность, уменьшилось энергопотребление.

Третьим этапом, наступившим в середине 60-х годов, стало использование микросхем. Первоначально использовались микросхемы низкой степени интеграции, содержащие простые транзисторные и резисторные сборки. Затем по мере развития технологии стали использоваться микросхемы, реализующие отдельные элементы цифровой схемотехники (сначала элементарные ключи и логические элементы, затем более сложные элементы — элементарные регистры, счётчики, сумматоры), позднее появились микросхемы, содержащие функциональные блоки процессора — микропрограммное устройство, арифметико-логическое устройство, регистры, устройства работы с шинами данных и команд.

Четвёртым этапом, в начале 70-х годов, стало создание, благодаря

прорыву в технологии создания БИС и СБИС (больших и сверхбольших интегральных схем, соответственно), микропроцессора — микросхемы, на кристалле которой физически были расположены все основные элементы и блоки процессора. Фирма Intel в 1971 году создала первый в мире 4-х разрядный микропроцессор 4004, предназначенный для использования в микрокалькуляторах. Постепенно практически все процессоры стали выпускаться в формате микропроцессоров. Исключением долгое время оставались только мелкосерийные процессоры, аппаратно оптимизированные для решения специальных задач (например, суперкомпьютеры или процессоры для решения ряда военных задач), либо процессоры, к которым предъявлялись особые требования по надёжности, быстродействию или защите от электромагнитных импульсов и ионизирующей радиации. Постепенно, с удешевлением и распространением современных технологий, эти процессоры также начинают изготавливаться в формате микропроцессора. Сейчас слова микропроцессор и процессор практически стали синонимами, но тогда это было не так, потому что обычные (большие) и микропроцессорные ЭВМ мирно сосуществовали ещё, по крайней мере, 10-15 лет, и только в начале 1980-х годов микропроцессоры вытеснили своих старших собратьев. Тем не менее, центральные процессорные устройства некоторых суперкомпьютеров даже сегодня представляют собой сложные комплексы, построенные на основе микросхем большой и сверхбольшой степени интеграции. Надо сказать, что переход к микропроцессорам позволил потом создать персональные компьютеры, которые теперь проникли почти в каждый дом.

Первым общедоступным микропроцессором был 4-разрядный Intel 4004, представленный 15 ноября 1971 года корпорацией Intel. Он содержал 2300 транзисторов, работал на тактовой частоте 92,6 кГц[1] и стоил 300 долл. Далее его сменили 8-разрядный Intel 8080 и 16-разрядный 8086, заложившие основы архитектуры всех современных настольных процессоров. Из-за

распространённости 8-разрядных модулей памяти был выпущен дешевый 8088, упрощенная версия 8086, с 8-разрядной шиной памяти. Затем проследовала его модификация 80186. В процессоре 80286 появился защищённый режим с 24-битной адресацией, позволявший использовать до 16 Мб памяти. Процессор Intel 80386 появился в 1985 году и привнёс улучшенный защищённый режим, 32-битную адресацию, позволившую использовать до 4 Гб оперативной памяти и поддержку механизма виртуальной памяти. Эта линейка процессоров построена на регистровой вычислительной модели.

Параллельно развиваются микропроцессоры, взявшие за основу стековую вычислительную модель.

За годы существования микропроцессоров было разработано множество различных их архитектур. Многие из них (в дополненном и усовершенствованном виде) используются и поныне. Например, Intel x86, развившаяся вначале в 32-битную IA-32, а позже в 64-битную x86-64 (которая у Intel называется EM64T). Процессоры архитектуры x86 вначале использовались только в персональных компьютерах компании IBM (IBM PC), но в настоящее время всё более активно используются во всех областях компьютерной индустрии, от суперкомпьютеров до встраиваемых решений. Также можно перечислить такие архитектуры как Alpha, POWER, SPARC, PA-RISC, MIPS (RISC-архитектуры) и IA-64 (EPIC-архитектура).

В современных компьютерах процессоры выполнены в виде компактного модуля (размерами около 5×5×0,3 см), вставляющегося в ZIF-сокет. Большая часть современных процессоров реализована в виде одного полупроводникового кристалла, содержащего миллионы, а с недавнего времени даже миллиарды транзисторов.

Архитектура фон Неймана и Гарвардская архитектура.

Большинство современных процессоров для персональных компьютеров в общем основаны на той или иной версии циклического

процесса последовательной обработки данных, изобретённого Джоном фон Нейманом.

Дж. фон Нейман придумал схему постройки компьютера в 1946 году. Отличительной особенностью архитектуры фон Неймана является то, что инструкции и данные хранятся в одной и той же памяти.

В различных архитектурах и для различных команд могут потребоваться дополнительные этапы. Например, для арифметических команд могут потребоваться дополнительные обращения к памяти, во время которых производится считывание операндов и запись результатов.

Этапы цикла выполнения:

1. Процессор выставляет число, хранящееся в регистре счётчика команд, на шину адреса и отдаёт памяти команду чтения.
2. Выставленное число является для памяти адресом; память, получив адрес и команду чтения, выставляет содержимое, хранящееся по этому адресу, на шину данных и сообщает о готовности.
3. Процессор получает число с шины данных, интерпретирует его как команду (машинную инструкцию) из своей системы команд и исполняет её.
4. Если последняя команда не является командой перехода, процессор увеличивает на единицу (в предположении, что длина каждой команды равна единице) число, хранящееся в счётчике команд; в результате там образуется адрес следующей команды.

Данный цикл выполняется неизменно, и именно он называется *процессом* (откуда и произошло название устройства).

Во время процесса процессор считывает последовательность команд, содержащихся в памяти, и исполняет их. Такая последовательность команд называется программой и представляет алгоритм работы процессора. Очерёдность считывания команд изменяется в случае, если процессор считывает команду перехода, — тогда адрес следующей команды может

оказаться другим. Другим примером изменения процесса может служить случай получения команды остановка или переключение в режим обработки прерывания.

Команды центрального процессора являются самым нижним уровнем управления компьютером, поэтому выполнение каждой команды неизбежно и безусловно. Не производится никакой проверки на допустимость выполняемых действий, в частности, не проверяется возможная потеря ценных данных. Чтобы компьютер выполнял только допустимые действия, команды должны быть соответствующим образом организованы в виде необходимой программы.

Скорость перехода от одного этапа цикла к другому определяется тактовым генератором. Тактовый генератор вырабатывает импульсы, служащие ритмом для центрального процессора. Частота тактовых импульсов называется тактовой частотой.

Конвейерная архитектура (pipelining) была введена в центральный процессор с целью повышения быстродействия. Обычно для выполнения каждой команды требуется осуществить некоторое количество однотипных операций, например: выборка команды из ОЗУ, дешифровка команды, адресация операнда в ОЗУ, выборка операнда из ОЗУ, выполнение команды, запись результата в ОЗУ. Каждую из этих операций сопоставляют одной ступени конвейера. Например, конвейер микропроцессора с архитектурой MIPS-I содержит четыре стадии:

- получение и декодирование инструкции,
- адресация и выборка операнда из ОЗУ,
- выполнение арифметических операций,
- сохранение результата операции.

После освобождения k-й ступени конвейера она сразу приступает к работе над следующей командой. Если предположить, что каждая ступень конвейера тратит единицу времени на свою работу, то выполнение команды на

конвейере длиной в n ступеней займёт n единиц времени, однако в самом оптимистичном случае результат выполнения каждой следующей команды будет получаться через каждую единицу времени.

Действительно, при отсутствии конвейера выполнение команды займёт n единиц времени (так как для выполнения команды по-прежнему необходимо выполнять выборку, дешифровку и т. д.), и для исполнения m команд понадобится $n*m$ единиц времени. При использовании конвейера (в самом оптимистичном случае) для выполнения m команд понадобится всего лишь $n + m$ единиц времени.

Факторы, снижающие эффективность конвейера:

Простой конвейера, когда некоторые ступени не используются (напр., адресация и выборка операнда из ОЗУ не нужны, если команда работает с регистрами).

Ожидание: если следующая команда использует результат предыдущей, то последняя не может начать выполняться до выполнения первой (это преодолевается при использовании внеочередного выполнения команд — out-of-order execution).

Очистка конвейера при попадании в него команды перехода (эту проблему удаётся сгладить, используя предсказание переходов).

Некоторые современные процессоры имеют более 30 ступеней в конвейере, что увеличивает производительность процессора, однако приводит к большому времени простоя (например, в случае ошибки в предсказании условного перехода). Не существует единого мнения по поводу оптимальной длины конвейера: различные программы могут иметь существенно различные требования.

Суперскалярная архитектура.

Способность выполнения нескольких машинных инструкций за один такт процессора путем увеличения числа исполнительных устройств. Появление этой технологии привело к существенному увеличению производительности,

в то же время существует определенный предел роста числа исполнительных устройств, при превышении которого производительность практически перестает расти, а исполнительные устройства простаивают. Частичным решением этой проблемы являются, например, технология Hyper Threading.

CISC-процессоры.

Complex instruction set computer — вычисления со сложным набором команд. Процессорная архитектура, основанная на усложнённом наборе команд. Типичными представителями CISC являются микропроцессоры семейства x86 (хотя уже много лет эти процессоры являются CISC только по внешней системе команд: в начале процесса исполнения сложные команды разбиваются на более простые микрооперации (МОП'ы), исполняемые RISC-ядром).

RISC-процессоры.

Reduced instruction set computer — вычисления с упрощённым набором команд (в литературе слово «reduced» нередко ошибочно переводят как «сокращённый»). Архитектура процессоров, построенная на основе упрощённого набора команд, характеризуется наличием команд фиксированной длины, большого количества регистров, операций типа регистр-регистр, а также отсутствием косвенной адресации. Концепция RISC разработана Джоном Коком (John Cocke) из IBM Research, название придумано Дэвидом Паттерсоном (David Patterson).

Упрощение набора команд призвано сократить конвейер, что позволяет избежать задержек на операциях условных и безусловных переходов. Однородный набор регистров упрощает работу компилятора при оптимизации исполняемого программного кода. Кроме того, RISC-процессоры отличаются меньшим энергопотреблением и тепловыделением.

Среди первых реализаций этой архитектуры были процессоры MIPS, PowerPC, SPARC, Alpha, PA-RISC. В мобильных устройствах широко используются ARM-процессоры.

MISC-процессоры.

Minimum instruction set computer — вычисления с минимальным набором команд. Дальнейшее развитие идей команды Чака Мура, который полагает, что принцип простоты, изначальный для RISC-процессоров, слишком быстро отошёл на задний план. В пылу борьбы за максимальное быстродействие, RISC догнал и перегнал многие CISC процессоры по сложности. Архитектура MISC строится на стековой вычислительной модели с ограниченным числом команд (примерно 20-30 команд).

VLIW-процессоры.

Very long instruction word — сверхдлинное командное слово. Архитектура процессоров с явно выраженным параллелизмом вычислений, заложенным в систему команд процессора. Являются основой для архитектуры EPIC. Ключевым отличием от суперскалярных CISC-процессоров является то, что для них загрузкой исполнительных устройств занимается часть процессора (планировщик), на что отводится достаточно малое время, в то время как загрузкой вычислительных устройств для VLIW-процессора занимается компилятор, на что отводится существенно больше времени (качество загрузки и, соответственно, производительность теоретически должны быть выше). Примером VLIW-процессора является Intel Itanium.

Структура и функциональная организация процессора. Архитектура фон Неймана, функциональная схема процессора, выборка команд, счетчик команд, команда перехода, три типа команд, форматы команд, кол-во регистров, запись возврата, скалярные и векторные команды.

Прерывания. Добавление к процессору и памяти устройств ввода-вывода. Программный ввод-вывод. Соотношение быстродействия программы и внешнего устройства. Процессор должен сам смотреть регистры. Сохранение значений регистров в памяти – стек.

Организация многоуровневой памяти. Кэш-память. Повышаем тактовую частоту, но достигаем предела. Всегда разрыв по скорости с памятью. Нельзя

делать быструю память и чтобы ее было много. Чем больше объем памяти, тем дольше до нее будет доступ. Цена быстрой памяти. Кэш – ассоциативная память, состоит из адреса и данных. Кэш-линии. Hit-miss. Write-through, write-back. Три способа организации кэша (прямое отображение, ассоциативная, множественно-ассоциативная).

Расслоение памяти. Разделение памяти на N блоков. Параллельное обращение к последовательным данным.

Направление развития микроэлектронных компонентов вычислительных систем. Факторы интеграции: пропускная способность между подсистемами, уменьшение системной платы (дешевле монтаж, больше надежность), уменьшается количество выводов, возможность размещения миллионов транзисторов на одном кристалле.

Архитектура процессоров с параллелизмом уровня команд.

Подходы к использованию ресурса транзисторов. Повышение производительности МП (микропроцессора) идет за счет увеличения тактовой частоты, совершенствование конвейерной и параллельной обработки данных за счет уменьшения времени доступа к памяти. Можно внутри МП делать функциональные блоки либо увеличивать кэш. Невозможно увеличить память так, чтобы ее хватило. Привлекательным выглядит использование ресурсов транзисторов для построения функциональных блоков. Но тут же возникает проблема загрузки блоков полезной работой.

Стирка белья как пример функциональных блоков с различным временным интервалом. Замачивание белья (5 мин), стирка (30), полоскание (10), выжим (5), сушка (10), глажка (5). 1-6-2-1-2-1.

Суперскаляры и длинное командное слово (VLIW**).** Получение на одном такте нескольких результатов. MMX. Обеспечивается аппаратурой либо

компилятором. Суперскалярам в программе не указывается, что нужно использовать параллелизм. При использовании VLIW процессоров наоборот, указывается, какой сопроцессор, что и когда будет исполнять. Оба эти подхода принадлежат к классу архитектур, использующих параллелизм уровня команд.

Зависимости между командами, препятствующие их параллельному исполнению. Программы пишутся последовательно, однако с целью достижения большей эффективности можно исполнять несколько команд одновременно, в некоторых случаях в порядке, отличном от исходного. ILP процессоры и компиляторы преобразуют упорядоченное множество команд в частично упорядоченное множество, структурированное зависимостями по данным и управлению. Зависимости по управлению – главное препятствие высокопараллельному исполнению.

If (a > 0) (t=t+1,a=a-t, f=a*2); else (t=t+5,a=t-a, f=a*4);

Зависимости по данным RAR, RAW, WAW, WAR

Структурный параллелизм микропроцессоров с разнесенной архитектурой. Использование естественного параллелизма вычислений целочисленных адресных выражений и собственно обработки данных с плавающей точкой. Адресный и исполнительный подпроцессоры. Разнесенная архитектура позволяет достигать при скалярной обработке производительности, характерной для векторных процессоров, за счет предвыборки и развертки последовательных витков в процессоре А.

Предварительная выборка команд и предсказание переходов. Пустые такты при командах ветвления. Отложенные переходы (исполнение команды после ветвления, безусловно), предсказание переходов (выборка команд по разным условным веткам, их исполнение), выполнение с измененным порядком следования команд (выхватывание тех команд, которые уже имеют готовые операнды), условное исполнение. Условное выполнение команд в VLIW-процессорах. Демонстрация примера условного выполнения команд.

Исполнение команд, завершение.

Направления развития архитектуры процессоров с параллелизмом уровня команд. Суперскаляр выбирает команды и исследует их с целью выявления переходов, типа команд, действия по смягчению зависимостей между данными. VLIW процессор все это возлагает на компилятор. Вложенные условия будут выбивать конвейер суперскаляра. К тому же, окно просмотра команд ограничивает параллелизм исполнения. VLIW же распараллелит на этапе компиляции, и его АЛУ простые, можно достигнуть большей тактовой частоты. Но команды ветвления будут давать большие простои. Окно исполнения у VLIW процессоров маленькое, нет смысла делать его большим, так как оно работает на уровне компилятора. Идеология VLIW требует большего набора регистров, большего числа перекрестных связей.

Классификация систем на кристалле и ЦСП. Общая архитектура.

Классификация СнК. Микропроцессоры на одном ядре. Их операционные системы. Возможные применения. Недостатки и преимущества данных процессоров. Многоядерные микропроцессоры, их необходимость. Сложность разработки и доведения до промышленного использования. Микропроцессоры с неизменяемым кодом, их преимущества и недостатки.

Общая архитектура. Общая схема. Гетерогенные и гомогенные ЦСП. Структурная архитектура СнК TI и Trident.

Введение. Операционная система GNU/Linux. Особенности

Введение. Состав и цели курса.

Операционная система GNU/Linux. Историческая справка.

1969. Первая версия UNIX. Подразделение Bell Labs компании AT&T

Кен Томпсон, Деннис Ритчи, Дуглас МакИлрой.

Написана на ассемблере.

1973. Третья редакция UNIX, со встроенным компилятором языка Си.

1975. Пятая редакция, полностью переписанная на Си.

1978. Седьмая редакция была последней единой версией UNIX. В ней появился близкий к современному интерпретатор командной строки Bourne shell.

1983. Ричард Столлмэн объявил о открытии проекта GNU — попытки создания свободной UNIX-подобной операционной системы с нуля, без использования оригинального исходного кода.

1991. Первая версия Linux. Линус Торвальдс.

1992. Версия 0.12 Linux была опубликована под лицензией GPL. Стало возможно появление свободной операционной системы GNU/Linux

Linux - общее название Unix-подобных операционных систем на основе одноимённого ядра и собранных для него библиотек и системных программ, разработанных в рамках проекта **GNU**.

Linux обладает следующими характеристиками:

1. Многопользовательский доступ.

2. Страничная организация памяти.

Системная память Linux организована в виде страниц объемом 4 КБайт. Если оперативная память полностью исчерпана, ОС будет искать давно не использованные страницы памяти для их перемещения из памяти на жесткий диск. При этом переносятся только неиспользованные страницы, а не все, относящиеся к какому-либо приложению.

3. Загрузка выполняемых модулей "по требованию".

Ядро Linux поддерживает выделение страниц памяти по требованию, при котором только необходимая часть кода исполняемой программы находится в оперативной памяти, а неиспользуемые в данный момент части остаются на диске.

4. Совместное использование исполняемых программ.

Если необходимо запустить одновременно несколько копий какого-то приложения (либо один пользователь запускает несколько идентичных задач, либо разные пользователи запускают одну и ту же задачу), то в память загружается только одна копия исполняемого кода этого приложения, которая используется всеми одновременно исполняющимися идентичными задачами.

5. Общие библиотеки.

Существует некоторое количество стандартных библиотек, используемых одновременно более чем одним процессом. В новых системах (в частности, в Linux), обеспечивается работа с динамически и статически разделяемыми библиотеками, что позволяет сократить размер отдельных приложений.

6. Динамическое кэширование диска

Кэширование диска — это использование части оперативной памяти для хранения часто используемых данных с диска, что существенно ускоряет доступ к часто используемым программам и задачам. Linux использует динамичную систему кэширования: память, зарезервированная под кэш, увеличивается, когда память не используется, и уменьшается, если системе или процессу пользователя требуется больше памяти.

7. 100%-ное соответствие стандарту POSIX 1003.1. Частичная поддержка возможностей System V и BSD.

POSIX 1003.1 (Portable Operating System Interface — интерфейс мобильной операционной системы) задает стандартный интерфейс Unix-систем, который описывается набором процедур языка Си. Сейчас он поддерживается всеми новыми ОС. Microsoft Windows NT также поддерживает POSIX 1003.1. Linux 100%-но соответствует POSIX. Дополнительно поддерживаются некоторые возможности System V и BSD для увеличения совместимости. Для разработчика это означает то, что написав программу, также соответствующую стандарту POSIX, он получает программу, работающую на всех платформах, поддерживаемых Linux.

8. System V IPC.

Linux использует технологию IPC (InterProcess Communication) для обмена сообщениями между процессами, использования семафоров и общей памяти.

9. Возможность запуска исполняемых файлов других ОС.

Для запуска Windows-приложений создана программная прослойка Wine. Wine воспринимает системные вызовы Windows-приложений к библиотекам операционной системы и подменяет их своими. ОС Linux способна также выполнять бинарные файлы других Intel-ориентированных Unix-платформ, соответствующих стандарту iBCS2 (intel Binary Compatibility).

10. Поддержка различных форматов файловых систем.

Linux поддерживает большое число форматов файловых систем, включая файловые системы DOS и OS/2, а также современные журналируемые файловые системы. При этом и собственная файловая система Linux, которая называется Second Extended File System (ext2fs), позволяет эффективно использовать дисковое пространство.

11. Сетевые возможности.

Linux можно интегрировать в любую локальную сеть. Поддерживаются все службы Unix, включая Networked File System (NFS), удаленный доступ (telnet, rlogin), работа в TCP/IP сетях, dial-up-доступ по протоколам SLIP и PPP, и т. д.. Также поддерживается включение Linux-машины как сервера или клиента для другой сети, в частности, работает общее использование (sharing) файлов и удаленная печать в Macintosh, NetWare и Windows.

Работа на разных аппаратных платформах.

Linux написан на языке Си, и может быть легко портирован на новые платформы реализацией необходимого HAL и компилирования исходного кода.

13. Не имеет единой “официальной” комплектации.

Вместо этого Linux поставляется в форме дистрибутивов.

Дистрибутив содержит программы для начальной инициализации системы (инициализация аппаратной части, загрузка урезанной версии системы и

запуск программы-установщика), программу-установщик (для выбора режимов и параметров установки) и набор специальных файлов, содержащих отдельные части системы (так называемые пакеты).

Linux, GNU tools и свободное ПО – это детали. Дистрибутив – это готовая конструкция.

Учетные записи пользователей.

Linux — многопользовательская операционная система, и все процессы в ней запускаются от имени определенного пользователя. Каждый пользователь в системе имеет определенный цифровой идентификатор (uid) и связанное с ним символьное имя. Кроме того, каждый пользователь принадлежит определенной **группе пользователей**, или нескольким. У каждой группы также есть свой идентификатор (gid) и имя. Узнать свой uid и к каким группам принадлежит пользователь можно с помощью команды id.

В Linux всегда есть особый пользователь, чей uid == 0. Это так называемый **суперпользователь (root)**. Его приблизительным аналогом в Windows является Администратор.

Во время работы в Linux необходимо представиться системе. После авторизации пользователя все процессы запускаются от его имени. Процесс загрузки системы выполняется от имени суперпользователя.

Запуск программ. Межпрограммное взаимодействие. Потоки ввода/вывода.

GNU/Linux — современная операционная система с разнообразными графическими возможностями. Однако свою историю она ведет от UNIX, с времен, когда основным способом общения с ЭВМ был текстовый терминал (командная строка).

Для **запуска программы** необходимо в командной строке написать ее имя. Программа выполняется после выполнения перевода строки (клавиша Enter). Обычно программа выводит сообщения о своей деятельности на экран

и система переходит к ожиданию следующей команды. Примером такой программы является uname. Для управления такими программами используются *аргументы командной строки*. Аргументы передаются после имени программы и разделяются пробелом. Чтобы узнать, какие аргументы можно передавать программе, можно запустить эту программу с аргументом '--help' (например, 'uname --help', либо воспользоваться программой справки man).

Другие программы могут читать команды пользователя с клавиатуры, выполняя их по мере возможности (обычно после перевода строки). Примеры - интерактивный калькулятор bc, текстовые редакторы vi и nano. Для остановки этих команд служит либо специальное сочетание клавиш (например, Alt+X для nano), специальная команда (quit в bc, :q в vi), либо команде можно отправить сигнал остановки.

Сигнал — это асинхронное уведомление о каком-то событии. Для того, чтобы программа отозвалась на сигнал, в ее коде должен быть реализован обработчик этого сигнала. Если в программе не установлен обработчик, то выполняется обработчик по умолчанию. Также программа может игнорировать определенные сигналы. Сигналы представляют собой числовые константы, зарезервированные системой. Сигналы имеют также символьные имена. Примером системного сигнала является SIGINT. Этот сигнал отправляется текущему процессу сочетанием Ctrl+C. Если процесс не игнорирует этот сигнал, то получив его, он завершает свою работу. Другим системным сигналом является SIGTSTP — приостановка работы, сочетание клавиш Ctrl+Z. Получивший этот сигнал процесс приостанавливается, а управление возвращается командной строке. Получить список приостановленных задач можно командой jobs. Продолжить выполнение приостановленной задачи — командой fg. Кроме того, приложения в Linux могут выполняться в фоновом режиме. В этом случае они будут продолжать свою работу, не забирая себе фокус ввода команд от пользователя. Для

запуска приложения в фоновом режиме используется символ '&', например: 'bc &'. Запущенную фоном программу можно вернуть в интерактивный режим также с помощью команды fg.

Важной концепцией программ для Linux являются **потоки ввода/вывода**. У каждой программы есть три стандартных потока данных — **stdin**, **stdout** и **stderr**, они имеют файловые дескрипторы 0, 1 и 2. С первого потока программа берет свои данные. На второй поток эти данные выводятся. Третий используется для сообщений об ошибках. По умолчанию stdin связан с клавиатурой, а stdout и stderr — с текущим терминалом. Именно поэтому программа получает команды, набранные с клавиатуры, и мы видим на экране результаты ее выполнения. Но стандартные потоки можно *перенаправить*, например, в файл. При этом, для перенаправления потока ввода используется символ '<', а для вывода - '>'. Например:

bc <commands.txt >results.txt

Для перенаправления потоков вывода также могут использоваться два символа '>>', в этом случае выводимые данные будут добавлены к файлу вместо его перезаписи.

Одной из важнейших особенностей GNU/Linux является **конвейер** — возможность нескольких программ работать совместно, когда выход одной программы непосредственно идет на вход другой без использования промежуточных временных файлов. Для составления конвейера используется символ '|' (вертикальная черта):

ps | grep sh | wc -l

Особым типом программ в Linux являются так называемые **демоны**. Названные в честь демона Максвелла, невидимого сортировщика молекул, демоны Linux сразу после запуска выполняются в фоновом режиме, без явного вмешательства пользователя. Обычно демоны во время запуска закрывают стандартные файловые потоки и создают свою копию (fork), которая занимается непосредственно работой, а запущенный процесс

завершается и возвращает управление.

Все запущенные процессы в системе имеют уникальный номер — pid. Именно он используется в качестве адресата назначения при отправке сигнала, благодаря чему можно управлять фоновыми приложениями. Для отправки сигнала используется команда kill. Названа она так потому, что по умолчанию отправляет процессу сигнал SIGTERM, заставляющий эту программу прекратить выполнение. Если же программа не обрабатывает этот сигнал, но ее требуется немедленно остановить, приходится использовать особый сигнал SIGQUIT (9), который немедленно завершает выполнение программы и который невозможно игнорировать. Команда kill используется для отправки сообщения процессу по его номеру, но обычно удобнее останавливать процесс по имени. Для этого служит команда killall. С помощью нее также можно определить, запущена ли та или иная программа, попробовав отправить ей сигнал 0 (который просто всегда игнорируется), например: 'killall -0 bc'. Если ни одной программы не запущено, сообщение об этом будет выведено на stderr.

Файловая структура Linux. Права доступа

Принцип «всё есть файл». Простое описание системы UNIX, также применимое к Linux, заключается в следующем: "В системе UNIX всё есть файл; а если что-то не файл, то это процесс". Файлами являются программы; потоки ввода-вывода ведут себя как файлы. Неудивительно, что тому, где располагаются файлы в Linux, уделено особое внимание.

Файловая структура Linux представляет собой дерево. У каждого файла есть путь, который может записываться относительно текущего каталога, обозначаемого точкой '.', либо от корня, обозначаемого '/'. Директории разделяются также с помощью символа '/'. Рассмотрим типичное расположение системных файлов в Linux:

/	Корневой каталог. С него всё берёт своё начало.

/boot	Постоянные файлы для загрузки системы, в частности, ядро
/dev	Каталог специальных файлов или файлов устройств
/etc	Основные конфигурационные файлы
/etc/rc.d	Скрипты загрузки системы.
/home	Домашние каталоги пользователей
/lib	Разделяемые библиотеки, необходимые программам на языке С
/lib/modules	Содержит подключаемые модули для всех установленных ядер
/usr	Дополнительное программное обеспечение
/proc	Информация о процессах, ядре, оборудовании
/root	Домашний каталог суперпользователя
/sys	Информация ядра Linux об устройствах и драйверах
/tmp	Временные файлы. Сохранность содержимого не обязательна.
/var	Различные переменные данные. Например, базы данных.
/var/log	Журналы событий (логи)
/bin	Содержит программы, необходимы во время старта системы
/sbin	Административные программы
/usr/bin	Исполняемые файлы пользовательских приложений
/usr/include	Заголовочные файлы
/usr/lib	Объектные библиотеки подпрограмм, динамические библиотеки
/usr/local	Повторяет структуру /usr. Содержит приложения, не входящие в дистрибутив
/usr/man	Страницы интерактивного руководства man
/usr/sbin	Административные приложения
/usr/share	Содержит неизменяемые данные программ, значки, звуки
/usr/src	Исходные тексты для различных частей Linux
/usr/src/linux	Исходные тексты для ядра Linux
/mnt /media	Каталог для монтирования внешних носителей

Права доступа к файлам. В системе Linux каждый файл принадлежит пользователю и группе пользователей. Есть также третья категория пользователей, которые не являются пользователем-владельцем и не принадлежат группе, владеющей файлом. Для каждой категории пользователей разрешение на чтение, запись и выполнение может быть предоставлено или отклонено. Чтобы увидеть разрешения на доступ к файлу, используется команда ls с ключом -l, например:

ls -l /var/log/messages

-rw-r----- 1 root root 2365991 Май 31 15:15 /var/log/messages

Первый символ вывода — это тип файла. Для простых файлов это прочерк '-'. За ним идет серия из трех групп по три символа, описывающие разрешения для трех категорий пользователей. Категории расположены в следующем порядке: владелец, группа, прочие. Разрешения для всех категорий указываются в порядке: чтение, запись и выполнение (исполнение, запуск). В примере выше, владельцу файла разрешены чтение и запись, группе только чтение, всем остальным доступ к файлу запрещен.

После прав доступа указывается счетчик ссылок на файл. Далее указываются имя владельца и имя группы файла. В данном случае это root и root. После идет размер файла в байтах, время (последней модификации) файла, и, наконец, его имя.

Бит исполнения также имеет особое значение в Linux. Он определяет, возможно ли выполнение данного файла как программы. Именно этим, а не особым расширением типа .exe, отличаются исполняемые файлы в Linux. Кроме того, важно понимать, откуда Linux берет программы, когда мы вводим их имя и запускаем. Linux ищет их в списке путей, в которые по умолчанию входят /bin,/usr/bin, а также /sbin и /usr/sbin. Все программы, которые не находятся в этих директориях, необходимо вызывать с указанием пути до самого файла программы. Например, для запуска программы foo из

текущей директории, применяется команда ./foo. Кроме того, программа должна иметь разрешение на исполнение для текущего пользователя.

Специальные типы файлов. Файловые системы, монтирование

Особые типы файлов.

В соответствии с принципами «всё есть файл», **директории** также представляют собой файлы. При просмотре их свойств с помощью 'ls -l' тип файла обозначается символом 'd', а вместо количества ссылок указывается количество вложенных элементов. Содержимое «файла» директории представляет собой собственно список файлов директории.

Символические ссылки являются другим примером специального типа файлов. Символические ссылки содержат не данные, а путь к файлу, данные содержащему. При этом символические ссылки могут указывать на другие символические ссылки. Обозначаются символом 'l'. Так как символическая ссылка указывает на путь к файлу, то нет разницы, где находится файл назначения — на том же разделе, или даже на смонтированном сетевом носителе.

Существуют также **жесткие ссылки**. Они представляются системе как обычные файлы, и связаны с особенностью реализации файловых систем UNIX. Каждый файл представляет собой структуру блоков данных на диске, имеющую уникальный индексный дескриптор (или i-node) и набор атрибутов (метаинформацию). Жёсткая ссылка связывает индексный дескриптор файла с каталогом и дает ему имя. Файл может иметь несколько жёстких ссылок, то есть одновременно фигурировать на диске под различными именами и/или различных каталогах. Количество жёстких ссылок файла (то есть количество его имён) хранится в метаинформации на уровне файловой системы. Обычно, файл имеет только одну жесткую ссылку, автоматически записываемую в каталог при его (файла) создании. Дополнительную жесткую ссылку можно

создать с помощью команды ln. Все жёсткие ссылки одного файла равноправны и неотличимы друг от друга, удаление одной из них не приводит к удалению файла — файл удаляется автоматически только после удаления всех жестких ссылок (и после его закрытия всеми программами).

В связи с тем, что жесткие ссылки ссылаются на индексный дескриптор, уникальный в пределах дискового раздела, создание жесткой ссылки на файл в каталоге другого раздела невозможно. Кроме UFS, жесткие ссылки также реализованы в файловой системе NTFS.

Именованные трубы (FIFO). Особый тип файлов, используемый для одностороннего обмена данными между приложениями. Обозначаются символом 'p' (pipe). Данные, записываемые в эту трубу, можно прочитать исключительно в том же порядке, в котором они в нее поступают. Аналогичным образом данные двигаются между программами в конвейере. Именованные трубы часто применяются в том случае, если программа не умеет получать данные из стандартного потока ввода.

Сокеты UNIX — особый тип файлов, используемый для двустороннего обмена данными. Работа с сокетами UNIX аналогична работе с сетевыми сокетами, только вместо сетевого адреса указывается путь к файлу сокета, и передаваемые данные не используют сетевой стек и не покидают машины, на которой выполняется обмен.

Символьные и блочные устройства являются абстракциями над аппаратным обеспечением компьютера. Располагаются они обычно в специальной директории /dev. Отличаются они только способом представления данных: в символьных устройствах это поток символов, а в блочных — блоки данных. Так, например, устройство «терминал» представлено символьным устройством /dev/console, а жесткий диск — блочным устройством /dev/sda.

Чаще всего файлы устройств используются для взаимодействия с аппаратной частью системы. Например, нажатие клавиши на клавиатуре

заставляет появиться данным на устройстве /dev/console. За функционирование файлов устройств отвечают драйверы этих устройств, выполняющиеся в пространстве ядра.

Другой специальной директорией для взаимодействием с ядром является директория /proc. Она позволяет получить доступ к информации о системных процессах из ядра, необходимых для выполнения таких команд, как ps, w, top.

Монтирование файловых систем. Файловая структура объединяет все доступные системе файлы в одно дерево. При этом физически эти файлы могут храниться на самых разных носителях, в том числе в сети, в ОЗУ и даже существовать только виртуально, подобно файлам устройств.

Для того, чтобы подключить хранилище к дереву файлов, используется *монтирование*. В процессе монтирования Linux определяет файловую систему источника, проверяет ее целостность и включает структуру файлов носителя в имеющуюся. Отметим, что после монтирования файловой системы в каталог /mnt/disk2 прежнее содержимое этого каталога станет недоступно (так же, как информация о прежнем владельце и правах доступа к этому каталогу) до тех пор, пока вы не размонтируете вновь подключенную файловую систему. Прежнее содержимое не уничтожается, а просто становится временно недоступным. Поэтому в качестве точек монтирования лучше использовать пустые директории (заранее заготовленные). Монтирование выполняется командой mount, в самом простом случае достаточно указать,, что и куда монтировать, например, команда:

mount /dev/sda1 /mnt/storage

монтирует первый раздел первого жесткого диска в /mnt/storage. В этом случае определение типа файловой системы производится автоматически. Команды mount и umount (см. ниже) поддерживают в актуальном состоянии таблицу (или перечень) смонтированных файловых систем. Этот перечень сохраняется на диске в виде файла /etc/mtab. Этот файл можно просмотреть

непосредственно, или вывести на экран командой mount без аргументов.

Перед тем как отключить от компьютера носитель, на котором расположена файловая система, (чаще всего это требуется делать с дисками в дисководе, флэш-брелками и носителями других типов), необходимо "размонтировать" файловую систему (другими словами, "размонтировать носитель"). Эта операция выполняется с помощью команды umount (замечание для тех, кто знает английский язык: имя команды не является правильным английским словом, так что не вставляйте в него лишнюю букву "n"). В качестве аргумента команде umount надо дать либо имя устройства, либо точку монтирования. Демонтировать файловую систему может только тот пользователь, который ее смонтировал (и суперпользователь, конечно).
Размонтирование файловой системы возможно только тогда, когда в ней нет открытых файлов (в частности, не должно быть запущено программ, файлы которых расположены в данной системе) и в системе нет процессов, использующих эту файловую систему (т. е. демонтируемая файловая система не должна быть занятой).

Ядро Linux поддерживает несколько файловых систем. Наиболее распространенными являются файловые системы семейства Extended File System: **ext2**, **ext3** и **ext4**. Ext2 самая старая и простая файловая система, отличается поддержкой файловых атрибутов POSIX, высокой скоростью работы, но также отсутствием журнала, что означает возможные потери данных в случае сбоя и долгого времени восстановления. Ext3 добавляет к функциям ext2 поддержку журнала, из-за накладных расходов по действиям с журналом отличается чуть меньшей производительностью, но большей надежностью. Ext4 - современная версия файловой системы, поддерживает файлы и разделы большего объема и механизм пространственной (extent) записи файлов (новая информация добавляется в конец заранее выделенной по соседству области файла), уменьшающий фрагментацию и повышающий производительность. Благодаря этому в режиме работы без журнала файловая

система ext4 демонстрирует большую производительность в сравнении с ext2. Кроме того, существует ряд специализированных файловых систем, таких как **jffs2** и **ubifs**, специально адаптированных для работы с флэш-накопителями, а также **cramfs** и **squashfs** — файловые системы с доступом только для чтения, все содержимое которых сжимается для уменьшения занимаемого на носителе места.

Linux также имеет поддержку сторонних файловых систем, таких как **FAT** и **NTFS**. Linux способен читать и писать на разделы с этими файловыми системами, но не может грузиться с них, так как они не поддерживают файловые атрибуты POSIX. Другие файловые системы: fat, ntfs. Сетевые файловые системы: nfs, cifs.

Командная оболочка bash. Скрипты оболочки. Переменные окружения.

Командная оболочка bash обеспечивает выполнение всех приложений: нахождение вызываемых программ, их запуск и организацию ввода/вывода. Кроме того, оболочка отвечает за работу с переменными окружения и выполняет некоторые преобразования (подстановки) аргументов. Но главное свойство оболочки, которое делает ее мощным инструментом пользователя — это то, что она включает в себя простой язык программирования.

Bash позволяет с удобством пользоваться текстовым интерфейсом. Например, bash поддерживает **автодополнение команд** — достаточно написать часть имени или пути и нажать tab — и bash автоматически из контекста предложит возможные варианты, или сразу подставит единственный доступный.

Одна из основных функций оболочки состоит в том, чтобы организовать **исполнение команд** пользователя, вводимых им в командной строке. В частности, оболочка предоставляет пользователю два специальных

оператора для организации задания команд в командной строке: ; и &.

Оператор ;

Хотя чаще всего пользователь задает команды в командной строке по одной, имеется возможность задать в одной строке несколько команд, которые будут выполнены последовательно, одна за другой. Для этого используется специальный символ - оператор ;. Если не поставить этот разделитель команд, то последующая команда может быть воспринята как аргумент предыдущей. Таким образом, если написать в командной строке что-то вроде:

[user]$ command1 ; command2

то оболочка вначале запустит на выполнение команду command1, дождется, пока ее выполнение завершится, после чего запустит command2, дождется ее завершения, после чего снова выведет приглашение командной строки, ожидая следующих действий пользователя.

Оператор &

Оператор & используется для того, чтобы организовать исполнение команд в фоновом режиме. Если поставить значок & после команды, то оболочка вернет управление пользователю сразу после запуска команды, не дожидаясь, пока выполнение команды завершится. Например, если задать в командной строке "command1 & command2 &", то оболочка запустит команду command1, сразу же затем команду command2, и затем немедленно вернет управление пользователю.

Код завершения команды

Каждая команда возвращает код завершения (код завершения также называют возвращаемым значением и статусом выхода). В случае успеха команда должна возвращать 0, а в случае ошибки - ненулевое значение, которое, как правило, интерпретируется как код ошибки. Практически все команды и утилиты UNIX возвращают 0 в случае успешного завершения, но имеются и исключения из правил.

Операторы && и ||

Операторы && и || являются управляющими операторами. Если в командной строке стоит command1 && command2, то command2 выполняется в том и только в том случае, если код завершения команды command1 равен нулю. Аналогично, если командная строка имеет вид command1 || command2, то команда command2 выполняется тогда и только тогда, когда код завершения команды command1 отличен от нуля.

Раскрытие выражений (expansion).

Когда оболочка получает какую-то командную строку на выполнение, она до начала выполнения команды осуществляет "грамматический разбор" полученной командной строки. Одним из этапов такого "разбора" является раскрытие или подстановка выражений (expansion). В bash имеется семь типов подстановки выражений:

- раскрытие скобок (brace expansion);
- замена знака тильды (tilde expansion);
- подстановка параметров и переменных;
- подстановка команд;
- арифметические подстановки (выполняемые слева направо);
- разделение слов (word splitting);
- раскрытие шаблонов имен файлов и каталогов (pathname expansion).

Все эти операции выполняются именно в том порядке, как они здесь перечислены. Рассмотрим их последовательно.

Раскрытие скобок

Раскрытие скобок проще всего пояснить на примере. Предположим, что нам нужно создать сразу несколько подкаталогов в каком-то каталоге, или поменять владельца сразу у нескольких файлов. Эти действия можно выполнить с помощью следующих команд:

```
[user]$ mkdir /usr/local/src/bash/{old,new,dist,bugs}
[root]# chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

В первом случае в каталоге /usr/local/src/bash/ будут созданы подкаталоги old, new, dist и bugs. Во втором случае владелец будет изменен у файлов

/usr/ucb/ex
/usr/lib/ex?.?*
/usr/ucb/edit
/usr/lib/ex?.?*
/usr/ucb/ex
/usr/lib/how_ex
/usr/ucb/edit
/usr/lib/how_ex

То есть для каждой пары скобок генерируются несколько отдельных строк (их число равно числу слов, стоящих внутри скобок) путем приписывания к каждому слову из скобок (спереди) того, что стоит перед скобкой, и приписывания в конец каждого полученного слова того, что стоит после скобки. Еще один пример: строка a{d,c,b}e при раскрытии скобок превращается в три слова "ade ace abe".

Раскрытие скобок выполняется до выполнения других видов подстановок в командной строке, причем все специальные символы, встречающиеся в командной строке, в том числе внутри скобок, сохраняются неизменными (они будут интерпретированы на следующих этапах анализа строки).

Замена тильды (Tilde Expansion)

Если слово начинается с символа тильды ('~'), все символы до первого слэша (или все символы, если слэша нет) трактуются как имя пользователя (login name). Если это имя есть пустая строка (т. е. вслед за тильдой идет сразу слэш), то тильда заменяется на значение переменной HOME. Если значение переменной HOME не задано, тильда заменяется на полный путь к домашнему каталогу пользователя, запустившего оболочку.

Если вслед за знаком тильды (до слэша) стоит слово, совпадающее с именем

одного из легальных пользователей, тильда и имя пользователя заменяются полным именем домашнего каталога этого пользователя. Если слово, следующее за тильдой, не является именем пользователя (и не пусто), то оно остается неизменным. Если вслед за знаком тильды стоит ‘+’, эти два знака заменяются на полное имя текущего каталога (т. е. значение переменной PWD). Если за знаком тильды следует ‘-‘, подставляется значение переменой OLDPWD.

Подстановка параметров и переменных

Символ $ используется для обозначения операций подстановки параметров, подстановки команд и подстановок арифметических выражений. Выражение или имя, следующее за $, может быть заключено в скобки, что не обязательно, но удобно, так как позволяет отделить заменяемое выражение от следующих за ним слов или символов. Таким образом, чтобы в командной строке вызвать значение параметра (в частности, любой переменной), нужно вставить выражение вида ${parameter}.

Скобки необходимы только в том случае, если имя параметра состоит из нескольких цифр, или когда за именем следует символ, который не должен интерпретироваться как часть имени.

Все значения переменных подвергаются подстановке знака тильды, раскрытию параметров и переменных, подстановке команд, подстановкам арифметических выражений, а также удалению специальных символов цитирования (см. ниже). Разделение слов не производится, за исключением случая "$@". Раскрытие шаблонов имен файлов и каталогов не производится.

Подстановка команд

Подстановка команд является очень мощным инструментов bash. Она заключается в замене имени команды на результат ее выполнения. Существует две формы подстановки команд:

$(command) и `command`

Если применяется вторая из этих форм, то обратный слэш внутри кавычек

трактуется как литерал, кроме тех случаев, когда за ним следует $, `, или \. Если же используется форма $(command), все символы внутри скобок составляют команду и ни один из них не считается специальным символом.

Если подстановка производится внутри двойных кавычек, то в результатах подстановки не осуществляется разделение слов и раскрытие шаблонов имен файлов и каталогов.

Арифметические подстановки (Arithmetic Expansion)

Арифметические подстановки позволяют вычислить значение арифметического выражения и подставить вместо него результат. Существует две формы задания арифметических подстановок:

$[expression]

$((expression)),

где expression трактуется так, как если бы оно было заключено в двойные кавычки, но встречающиеся в expression двойные кавычки трактуются как простой литерал. Внутри expression выполняются подстановки параметров и команд.

Синтаксис выражения expression подобен синтаксису арифметических выражений в языке C, подробнее об этом можно прочитать в разделе ARITHMETIC EVALUATION man-страницы по команде bash. Например, команда

```
[user]$ echo $(( 2 + 3 * 5 ))
```

в качестве результата выдает "17".

Если выражение некорректно, bash выдает сообщение об ошибке.

Разделение слов (word splitting)

После завершения подстановок параметров, команд и арифметических выражений оболочка снова анализирует командную строку (в том виде, который она приобрела к этому моменту) и осуществляет разделение слов (word splitting).

Эта операция заключается в том, что в командной строке ищутся все

вхождения символов-разделителей, определенных в переменой IFS, и в соответствующих местах строки разделяются на отдельные слова. Если значение IFS равно пустой строке, разделение слов не производится.

Если в командной строке не производилось никаких подстановок, то разбиение на слова не производится.

Раскрытие шаблонов имен файлов и каталогов (Pathname Expansion)

Подстановки имен путей и файлов (Pathname expansion) используются для того, чтобы с помощью краткого образца или шаблона указать несколько имен файлов (или каталогов), соответствующих данному шаблону. После разделения слов, если не была задана опция -f, bash производит поиск в каждом слове командной строки символов *, ?, and [. Если будет найдено слово с одним или несколькими вхождениями таких символов, то это слово рассматривается как шаблон, который должен быть заменен словами из лексикографически упорядоченного списка имен путей, соответствующих данному шаблону. Если имен, соответствующих шаблону, не найдено, и переменная nullglob не задана, слово не изменяется. Если эта переменная установлена, а путей, соответствующих шаблону не найдено, слово удаляется из командной строки.

Специальные символы шаблонов имеют следующее значение:

Символ	Правила замены
*	Соответствует произвольной строке символов, включая пустую строку. Например, my*.txt будет заменено на myday.txt, myweek.txt и mymonth.txt (если такие файлы существуют), а *.jpg соответствует всем файлам с расширением jpg в указанном каталоге
?	Соответствует любому одиночному символу. Например, вместо шаблона file?.txt будут подставлены имена file1.txt и filex.txt, но не file10.txt

[...]	Соответствует любому символу из числа символов, указанных в скобках. Пары символов, разделенные знаком минуса, обозначают интервал; любой символ стоящий лексически между этими двумя символами, включая и символы, задающие интервал, соответствует шаблону. Если первым символом внутри скобок является ! или ^, то считается, что шаблону (в данной позиции) соответствуют все символы, не указанные в скобках

Шаблоны имен файлов очень часто применяются в командных строках, содержащих команду ls. Представьте себе, что вы хотите просмотреть информацию о содержимом каталога, в котором находится огромное количество разных файлов различных форматов, например, файлов с изображениями форматов gif, jpeg, avi и т. д.. Чтобы получить только список файлов формата jpeg, вы можете использовать команду

[user]$ ls *.jpg

Если в каталоге имеется множество файлов, имена которых представлены четырехзначными номерами, то следующей командой можно вывести только список файлов с номерами от 0200 до 0499:

[user]$ ls -l 0[2-4]??.*

Удаление специальных символов

После того, как все подстановки в командной строке сделаны, из нее еще удаляются все вхождения символов \, ` и ", которые служили для отмены специального значения других символов.

Специальные символы

Оболочка bash использует несколько символов из числа 256 символов набора ASCII в специальных целях, либо для обозначения некоторых операций, либо для преобразования выражений. В число таких символов входят символы:

` ~ ! @ # $ % ^ & * ( ) _ — [ ] { } : ; ' " / \ > <

а также символ с кодом 0, символ возврата каретки (генерируемый клавишей <Enter>) и пробел. В зависимости от ситуации эти специальные символы могут трактоваться либо в их специальном значении, либо в буквальном, т. е. как литералы. Но мы в основном будем предполагать, что все эти символы зарезервированы и не должны использоваться в качестве литералов. Это касается в первую очередь использования их в именах файлов и каталогов. Однако символы _, - и . (знак подчеркивания, дефис и точка) часто используются в именах файлов, так что именно этот пример показывает, что специальное значение эти символы имеют не всегда. В именах файлов только символы точки (.) и слэша (/) имеют специальное значение. Символ слэша служит для разделения имен отдельных каталогов, а точка имеет специальное значение, только если она является первым символом в имени файла (что означает, что файл является «скрытым»).

Символ \ (обратный слэш) можно назвать "символом отмены специального значения" для любого из специальных символов, который стоит сразу вслед за \. Например, если мы хотим использовать символ пробела в имени файла, мы должны вместо простого пробела поставить \. Например, возможна следующая команда:

```
[user]$ cp two_words two\ words
```

Символы ' и " (одинарные и двойные кавычки) могут быть названы "символами цитирования". Любой из этих символов всегда используется в паре с его копией для обрамления какого-то выражения, совсем как в обычной прямой речи. Если какой-то текст взят в одинарные кавычки, то все символы внутри этих кавычек воспринимаются как литералы, никаким из них не придается специального значения. Если вернуться к тому же примеру с пробелами в имени файла, то можно сказать, что для того, чтобы дать файлу имя "two words" надо взять имя в кавычки:

```
[user]$ cp two_words 'two words'
```

Различие в использовании символов ' и " состоит в том, что внутри

одинарных кавычек теряют специальное значение все символы, а внутри двойных кавычек — все специальные символы кроме $, ' и \ (знака доллара, одинарных кавычек и обратного слэша).

Параметры и переменные

Понятие параметра в оболочке bash подобно понятию переменной в обычных языках программирования. Именем (или идентификатором) параметра может быть слово, состоящее из алфавитных символов, цифр и знаков подчеркивания (только первый символ этого слова не может быть цифрой), а также число или один из следующих специальных символов: *, @, #, ?, - (дефис), $, !, 0, _ (подчеркивание).

Говорят, что параметр задан или установлен, если ему присвоено значение. Значением может быть и пустая строка. Чтобы вывести значение параметра, используют символ $ перед его именем. Так, команда

[user]$ echo name

выдаст на экран слово name, а команда

[user]$ echo $name

выдаст значение переменной name (если таковое, конечно, задано).

Параметры разделяются на три класса: *позиционные параметры*, *специальные параметры* (именами которых как раз и служат перечисленные только что специальные символы) и *переменные оболочки*.

Имена (идентификаторы) позиционных параметров состоят из одной или более цифр (только не из одиночного нуля). Значениями позиционных параметров являются аргументы, которые были заданы при запуске оболочки (первый аргумент является значением позиционного параметра 1, и т. д.). Изменить значение позиционного параметра можно с помощью встроенной команды set. Значения этих параметров изменяются также на время выполнения оболочкой одной из функций.

Специальные параметры являются шаблонами, замена (подстановка) которых производится следующим образом.

Параметр	Правила замены
*	Заменяется позиционными параметрами, начиная с первого. Если замена производится внутри двойных кавычек, то этот параметр заменяется на одно единственное слово, составленное из всех позиционных параметров, разделенных первым символом специальной переменной IFS (о ней будет сказано ниже). То есть ``$*" эквивалентно ``$1c$2c...", где c — первый символ в значении переменной IFS. Если IFS присвоено пустое значение или ее значение не установлено, параметры разделяются пробелами
@	Заменяется позиционными параметрами, начиная с первого. Если замена производится внутри двойных кавычек, то каждый параметр заменяется отдельным словом. Так, `` $@" эквивалентно "$1" "$2" ... Если позиционных параметров нет, то значение не присваивается (параметр @ просто удаляется)
#	Заменяется десятичным значением числа позиционных параметров
?	Заменяется статусом выхода последнего из выполнявшихся на переднем плане программных каналов
- (дефис)	Заменяется текущим набором значений флагов, установленных с помощью встроенной команды set или при запуске самой оболочки
$	Заменяется идентификатором процесса (PID) оболочки
!	Заменяется идентификатором процесса (PID) последней из выполняющихся фоновых (асинхронно выполнявшихся) команд
0	Заменяется именем оболочки или запускаемого скрипта. Если bash запускается для выполнения командного файла, $0 равно имени этого файла. В противном случае это значение равно полному пути к оболочке
_ (подчеркивание)	Заменяется последним аргументом предыдущей из выполнявшихся команд (если это параметр или переменная, то подставляется ее значение)

Специальные параметры, перечисленные в приведенной выше таблице, отличаются тем, что на них можно только ссылаться; присваивать им значения нельзя.

Переменная с точки зрения оболочки — это параметр, обозначаемый именем. Значения переменным присваиваются с помощью оператора следующего вида

[user]$ name=value

где name — имя переменной, а value — присваиваемое ей значение (может

быть пустой строкой). Имя переменой может состоять только из цифр и букв и не может начинаться с цифры. Значением может быть любой текст. Если значение содержит специальные символы, то его надо взять в кавычки. Присвоенное значение этих кавычек не содержит, естественно. Если переменная задана, то ее можно удалить, используя встроенную команду оболочки unset.

Набор всех установленных переменных оболочки с присвоенными им значениями называется окружением (environment) или средой оболочки. Вы можете просмотреть его с помощью команды set без параметров (только, может быть, следует организовать конвейер "set | less"). В выводе этой команды все переменные окружения перечисляются в алфавитном порядке. Для того чтобы просмотреть значение одной конкретной переменной, можно вместо команды set (в выводе которой нужную переменную еще искать и искать) можно воспользоваться командой

[user]$ echo $name

(правда, в этом случае вы должны знать имя интересующей вас переменной). Среди переменных, которые вы увидите в выводе команды set, встречаются очень интересные переменные. Обратите, например, внимание на переменную RANDOM. Если вы несколько раз подряд выполните команду

[user]$ echo $RANDOM

вы каждый раз будете получать новое значение. Дело в том, что эта переменная возвращает случайное целое из интервала 0 — 32 768.

Переменная PATH

Еще одна очень важная переменная имеет имя PATH. Она задает перечень путей к каталогам, в которых bash осуществляет поиск файлов (в частности, файлов с командами) в тех случаях, когда полный путь к файлу не задан в командной строке. Отдельные каталоги в этом перечне разделяются двоеточиями. По умолчанию переменная PATH включает каталоги /usr/local/bin, /bin, /usr/bin, /usr/X11R6/bin, т. е. имеет вид:

/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:

Для того, чтобы добавить каталог в этот список, нужно выполнить следующую команду:

[root]# PATH=$PATH:new_path.

При осуществлении поиска оболочка просматривает каталоги именно в том порядке, как они перечислены в переменной PATH.

Отметим, что можно включить в этот список и текущий каталог, добавив в переменную PATH точку. Однако этого не рекомендуется делать по соображениям безопасности: злоумышленник может положить в общедоступный каталог команду, имя которой совпадает с одной из часто выполняемых суперпользователем команд, но выполняющую совершенно другие действия (особенно если текущий каталог стоит в начале перечня путей поиска).

Переменная LD_LIBRARY_PATH

Аналогично переменной PATH, данная переменная задает перечень путей к каталогах, в которых осуществляется поиск общих библиотек. Обычно эта переменная не проинициализирована, так как библиотеки чаще всего находятся в нужных местах.

Команда export

Когда оболочка запускает на выполнение какую-то программу или команду, она передает им часть переменных окружения. Для того, чтобы переменная окружения передавалась запускаемому из оболочки процессу, ее нужно задавать с помощью специальной команды export, т. е. вместо

[user]$ name=value

надо записать

[user]$ export name=value

В таком случае все запускаемые из оболочки программы (в том числе вторичные экземпляры самой оболочки) будут иметь доступ к заданным таким образом переменным, т. е. могут вызывать их значения по именам.

Запуск команд с предустановленными переменными окружения

bash позволяет запускать команды, передав им дополнительные переменные окружения. При этом данные переменные будут видны только запущенной команде и ее потомкам. Для этого список этих переменных в виде «имя=значение» должен предшествовать команде, например:

[user]$ echo $LANG

ru_RU.UTF-8

[user]$ date

Срд Июн 1 17:57:34 NOVST 2011

[user]$ LANG=en_US date

Wed Jun 1 17:58:01 NOVST 2011

[user]$ echo $LANG

ru_RU.UTF-8

Если необходимо передать несколько переменных, то их нужно разделять пробелами.

Shell как язык программирования

Как уже говорилось выше, для построения произвольных алгоритмов необходимо иметь операторы проверки условий. Оболочка bash поддерживает операторы выбора if … then … else и case, а также операторы организации циклов for, while, until, благодаря чему она превращается в мощный язык программирования.

Операторы if и test (или [ ])

Конструкция условного оператора в слегка упрощенном виде выглядит так:

if list1 then list2 else list3 fi

где list1, list2 и list3 — это последовательности команд, разделенные запятыми и оканчивающиеся точкой с запятой или символом новой строки. Кроме того, эти последовательности могут быть заключены в фигурные скобки: {list}.

Оператор if проверяет значение, возвращаемое командами из list1. Если в

этом списке несколько команд, то проверяется значение, возвращаемое последней командой списка. Если это значение равно 0, то будут выполняться команды из list2; если это значение не нулевое, будут выполнены команды из list3. Значение, возвращаемой таким составным оператором if, совпадает со значением, выдаваемым последней командой выполняемой последовательности.

Полный формат команды if имеет вид:

if list then list [elif list then list] ... [else list] fi

(здесь квадратные скобки означают только необязательность присутствия в операторе того, что в них содержится).

В качестве выражения, которое стоит сразу после if или elif, часто используется команда test, которая может обозначаться также квадратными скобками [ ]. Команда test выполняет вычисление некоторого выражения и возвращает значение 0, если выражение истинно, и 1 в противном случае. Выражение передается программе test как аргумент. Вместо того, чтобы писать

test expression,

можно заключить выражение в квадратные скобки:

[expression].

Заметьте, что test и [— это два имени одной и той же программы, а не какое-то магическое преобразование, выполняемое оболочкой bash (только синтаксис [требует, чтобы была поставлена закрывающая скобка). Заметьте также, что вместо test в конструкции if может быть использована любая программа.

В заключение приведем пример использования оператора if:

```
if [ -e textmode2.htm ] ; then
  ls textmode*
else
  pwd
```

fi

Условные выражения, используемые в операторе `test`, строятся на основе проверки файловых атрибутов, сравнения строк и обычных арифметических сравнений. Сложные выражения строятся из следующих унарных или бинарных операций ("элементарных кирпичиков"):

-e file	Верно, если файл с именем file существует.
-d file	Верно, если file существует и является каталогом.
-x file	Верно, если файл с именем file существует и является исполняемым.
-z string	Верно, если длина строки равна нулю.
-n string	Верно, если длина строки не равна нулю.
string1 == string2	Верно, если строки совпадают. Вместо == может использоваться =.
string1 != string2	Верно, если строки не совпадают.
arg1 OP arg2	Здесь OP — это одна из операций арифметического сравнения: -eq (равно), -ne (не равно), -lt (меньше чем), -le (меньше или равно), -gt (больше), -ge (больше или равно). В качестве аргументов могут использоваться положительные или отрицательные целые.

Из этих элементарных условных выражений можно строить сколь угодно сложные с помощью обычных логических операций ОТРИЦАНИЯ, И и ИЛИ:

!(expression)	Булевский оператор отрицания.
expression1 -a expression2	Булевский оператор AND (И). Верен, если верны оба выражения.
expression1 -o expression2	Булевский оператор OR (ИЛИ). Верен, если верно любое из двух выражений.

Оператор case

Формат оператора case таков:

case word in [[(] pattern [| pattern] ...) list ;;] ... esac

Команда case вначале производит раскрытие слова word и пытается сопоставить результат с каждым из образцов pattern поочередно. После нахождения первого совпадения дальнейшие проверки не производятся,

выполняется список команд, стоящий после того образца, с которым обнаружено совпадение. Значение, возвращаемое оператором, равно 0, если совпадений с образцами не обнаружено. В противном случае возвращается значение, выдаваемое последней командой из соответствующего списка.

Следующий пример использования оператора case заимствован из системного скрипта /etc/rc.d/rc.sysinit.

```
case "$UTC" in
 yes|true)
  CLOCKFLAGS="$CLOCKFLAGS -u";
  CLOCKDEF="$CLOCKDEF (utc)";
  ;;
 no|false)
  CLOCKFLAGS="$CLOCKFLAGS --localtime";
  CLOCKDEF="$CLOCKDEF (localtime)";
  ;;
esac
```

Если переменная принимает значение yes или true, то будет выполнена первая пара команд, а если ее значение равно no или false – вторая пара.

Оператор for

Оператор for работает немного не так, как в обычных языках программирования. Вместо того, чтобы организовывать увеличение или уменьшение на единицу значения некоторой переменной при каждом проходе цикла, он при каждом проходе цикла присваивает переменной очередное значение из заданного списка слов. В целом конструкция выглядит примерно так:

for name in words do list done.

Правила построения списков команд (list) такие же, как и в операторе if.

Пример. Следующий скрипт создает файлы foo_1, foo_2 и foo_3:

```
for a in 1 2 3 ; do
```

```
  touch foo_$a
done
```

В общем случае оператор for имеет формат:

```
for name [ in word; ] do list ; done
```

Вначале производится раскрытие слова word в соответствии с правилами раскрытия выражений, приведенными выше. Затем переменной name поочередно присваиваются полученные значения, и каждый раз выполняется список команд list. Если "in word" пропущено, то список команд list выполняется один раз для каждого позиционного параметра, который задан.

В Linux имеется программа seq, которая воспринимает в качестве аргументов два числа и выдает последовательность всех чисел, расположенных между заданными. С помощью этой команды можно заставить for в bash работать точно так же, как аналогичный оператор работает в обычных языках программирования. Для этого достаточно записать цикл for следующим образом:

```
for a in $( seq 1 10 ) ; do
cat file_$a
done
```

Эта команда выводит на экран содержимое 10-ти файлов: "file_1", ..., "file_10".

Операторы while и until

Оператор while работает подобно if, только выполнение операторов из списка list2 циклически продолжается до тех пор, пока верно условие, и прерывается, если условие не верно. Конструкция выглядит следующим образом:

while list1 do list2 done.

Пример:

```
while [ -d mydirectory ] ; do
  ls -l mydirectory >> logfile
```

```
  echo -- SEPARATOR -- >> logfile
  sleep 60
done
```

Такая программа будет протоколировать содержание каталога "mydirectory" ежеминутно до тех пор, пока директория существует.

Оператор until аналогичен оператору while:

until list1 do list2 done.

Отличие заключается в том, что результат, возвращаемый при выполнении списка операторов list1, берется с отрицанием: list2 выполняется в том случае, если последняя команда в списке list1 возвращает ненулевой статус выхода.

Функции

Оболочка bash позволяет пользователю создавать собственные функции. Функции ведут себя и используются точно так же, как обычные команды оболочки, т. е. мы можем сами создавать новые команды. Функции конструируются следующим образом:

function name () { list }

Причем слово function не обязательно, name определяет имя функции, по которому к ней можно обращаться, а тело функции состоит из списка команд list, находящегося между { и }. Этот список команд выполняется каждый раз, когда имя name задано как имя вызываемой команды. Отметим, что функции могут задаваться рекурсивно, так что разрешено вызывать функцию, которую мы задаем, внутри нее самой.

Функции выполняются в контексте текущей оболочки: для интерпретации функции новый процесс не запускается (в отличие от выполнения скриптов оболочки).

Когда функция вызывается на выполнение, аргументы функции становятся позиционными параметрами (positional parameters) на время выполнения функции. Они именуются как $n, где n — номер аргумента, к

которому мы хотим получить доступ. Нумерация аргументов начинается с 1, так что $1 — это первый аргумент. Мы можем также получить все аргументы сразу с помощью $*, и число аргументов с помощью $#. Позиционный параметр 0 не изменяется.

Если в теле функции встречается встроенная команда return, выполнение функции прерывается и управление передается команде, стоящей после вызова функции. Когда выполнение функции завершается, позиционным параметрам и специальному параметру # возвращаются те значения, которые они имели до начала выполнения функции.

Локальные переменные (local)

Если мы хотим создать локальный параметр, можно использовать ключевое слово local. Синтаксис ее задания точно такой же, как и для обычных параметров, только определению предшествует ключевое слово local: local name=value.

Вот пример задания функции, реализующей упоминавшуюся выше команду seq:

```
seq()
{
 local I=$1;
 while [ $2 != $I ]; do
 {
  echo -n "$I ";
  I=$(( $I + 1 ))
 };
 done;
 echo $2
}
```

Обратите внимание на опцию -n оператора echo, она отменяет переход на новую строку. Хотя это и несущественно для тех целей, которые мы здесь

имеем в виду, это может оказаться полезным для использования функции в других целях.

Скрипты оболочки и команда source

Скрипт оболочки — это просто файл, содержащий последовательность команд оболочки. Подобно функциям, скрипты можно выполнять как обычные команды. Синтаксис доступа к аргументам такой же, как и для функций.

В общем случае при запуске скрипта запускается новый процесс. Для того, чтобы выполнить скрипт внутри текущей сессии bash, необходимо использовать команду source, синонимом которой является просто точка ".". Скрипт оболочки служит просто аргументом этой команды. Ее формат:

source filename [arguments]

или

. filename [arguments]

Эта команда читает и выполняет команды из файла с именем filename в текущем окружении и возвращает статус, определяемый последней командой из файла filename. Если filename не содержит слэша, то пути, перечисленные в переменной PATH, используются для поиска файла с именем filename. Этот файл не обязан быть исполняемым. Если в каталогах, перечисленных в PATH, нужный файл не найден, его поиск производится в текущем каталоге.

Если заданы аргументы, на время выполнения скрипта они становятся позиционными параметрами. Если аргументов нет, позиционные параметры не изменяются. Значение (статус), возвращаемое командой source, совпадает со значением, возвращаемым последней командой, выполненной в скрипте. Если ни одна команда не выполнялась, или файл filename не найден, то статус выхода равен 0.

Отметим также, что символ # служит для выделения в скриптах комментариев. Все, что стоит в текущей строке после этого символа и до символа конца строки, оболочка будет считать комментариями и

игнорировать (т. е. оболочка не рассматривает этот текст как команды). Если хотите убедиться в действии этого символа, введите в командной строке любую команду, поставив перед ней символ #, например, "# ls", и вы увидите, что команда игнорируется оболочкой.

Команда sh

Вы всегда можете запустить новый экземпляр оболочки bash, дав команду bash или sh. При этом можно заставить новый экземпляр оболочки выполнить какой-то скрипт, если передать имя скрипта в виде аргумента команды bash. Так, для выполнения скрипта myscript надо дать команду "sh myscript".

Если вы заглянете в какой-нибудь файл, задающий скрипт (таких файлов в системе очень много), вы увидите, что первая строка в нем имеет вид: #!/bin/sh. Это означает, что когда мы запускаем скрипт на выполнение как обычную команду, /bin/sh будет выполнять ее для нас. Можно заменить эту строку ссылкой на любую программу, которая будет читать файл и исполнять соответствующие команды. Например, скрипты на языке Perl начинаются со строки вида #!/bin/perl.

Процесс загрузки GNU/Linux. Модули ядра. Загрузка по сети.

Загрузчик

После включения питания компьютера и завершения тестирования аппаратной части BIOS считывает из первого сектора загрузочного диска короткую программу-загрузчик. Эта программа запускает основной системный загрузчик (например, grub), который, в свою очередь, загружает в память ядро системы, которое обычно хранится в файле vmlinuz-x.y.z-a в каталоге /boot. Здесь x.y.z — это номер версии ядра, а вместо символа a часто стоит указание на какие-то конкретные модификации ядра. Впрочем, название файла ядра может быть и другим, для загрузчика это не имеет значения, только это имя надо указать в конфигурационном файле загрузчика.

Загрузчик может передавать ядру дополнительные параметры загрузки, например, включение или отключение определенных аппаратных возможностей, а также указать использовать ли при загрузке **initrd** - Initial RAM Disk, диск в оперативной памяти для начальной инициализации. Это временная файловая система, используемая ядром Linux при начальной загрузке. initrd обычно используется для начальной инициализации перед монтированием «настоящих» файловых систем. initrd призван решить проблему курицы и яйца для модульного ядра: для монтирования файловой системы необходим модуль для работы с диском и файловой системой, а для чтения модуля необходима файловая система, с которой этот модуль читается. initrd — необязательная часть загрузки Linux, так как необходимые для загрузки модули могут входить в само ядро. В современных системах initrd представляет собой сжатый gzip'ом cpio-архив.

При загрузке с использованием initrd, ядро монтирует этот диск в качестве корня, загружает в память необходимые модули и после этого считается загруженным.

Процесс init и файл /etc/inittab

Сразу после загрузки ядро монтирует корневую файловую систему (заменяя ей initrd, который вытесняется из памяти) и запускает процесс init. Процесс init — это программа, которая ответственна за продолжение процедуры загрузки, и перевод системы от начального состояния, возникающего после загрузки ядра, в стандартное состояние обработки запросов многих пользователей. Init выполняет еще массу различных операций, необходимых для дальнейшей работы системы: проверку и монтирование файловых систем, запуск различных служб (демонов), запуск процедур логирования, оболочек пользователей на различных терминалах и т. д.

Точный список этих операций зависит от так называемого уровня выполнения (run level). Уровень выполнения определяет перечень действий, выполняемых процессом init, и состояние системы после загрузки, т. е.

конфигурацию запущенных процессов. Уровень выполнения идентифицируется одним символом. В ОС Linux существует 8 основных уровней выполнения:

0 — остановка системы;

1 — однопользовательский режим (для специальных случаев администрирования);

2 — многопользовательский режим без NFS (то же, что и 3, если компьютер не работает с сетью);

3 — полный многопользовательский режим;

4 — использование не регламентировано;

5 — обычно используется для запуска системы в графическом режиме;

6 — перезагрузка системы;

S (или s) — примерно то же, что и однопользовательский режим, но S и s используются в основном в скриптах.

Как видите, уровни 0, 1 и 6 зарезервированы для особых случаев. Относительно того, как использовать уровни со 2 по 5, единого мнения не существует. Некоторые системные администраторы используют разные уровни для того, чтобы задать разные варианты работы, например, на одном уровне запускается графический режим, на другом работают в сети и т. д. Вы можете сами решить, как использовать разные уровни для создания разных вариантов загрузки. Но для начала проще всего воспользоваться тем способом определения разных уровней, который был задан при установке.

Первым делом после старта процесс init считывает свой конфигурационный файл /etc/inittab. Этот файл состоит из отдельных строк. Если строка начинается со знака # или пуста, то она игнорируется. Все остальные строки состоят из 4 полей, разделенных двоеточиями:

id:runlevels:action:process

где:

id — идентификатор строки. Это произвольная комбинация, содержащая от

1 до 4 символов. В файле inittab не может быть двух строк с одинаковыми идентификаторами;

runlevels — уровни выполнения, на которых эта строка будет задействована. Уровни задаются цифрами или буквами без разделителей, например, 345;

process — процесс, который должен запускаться на указанных уровнях. Другими словами, в этом поле указывается имя программы, вызываемой при переходе на указанные уровни выполнения;

action — действие.

В поле action стоит ключевое слово, которое определяет дополнительные условия выполнения команды, заданной полем process. Допустимые значения поля action:

respawn — перезапустить процесс в случае завершения его работы;

once — выполнить процесс только один раз при переходе на указанный уровень;

wait — процесс будет запущен один раз при переходе на указанный уровень и init будет ожидать завершения работы этого процесса, прежде, чем продолжать работу;

sysinit — это ключевое слово обозначает действия, выполняемые в процессе загрузки системы независимо от уровня выполнения (поле runlevels игнорируется). Процессы, помеченные этим словом, запускаются до процессов, помеченных словами boot и bootwait;

boot — процесс будет запущен на этапе загрузки системы независимо от уровня выполнения;

bootwait — процесс будет запущен на этапе загрузки системы независимо от уровня выполнения, и init будет дожидаться его завершения;

initdefault — строка, в которой это слово стоит в поле action, определяет уровень выполнения, на который система переходит по умолчанию. Поле process в этой строке игнорируется. Если уровень выполнения, используемый

по умолчанию, не задан, то процесс init будет ждать, пока пользователь, запускающий систему, не введет его с консоли;

off — игнорировать данный элемент;

powerwait — позволяет процессу init остановить систему, когда пропало питание. Использование этого слова предполагает, что имеется источник бесперебойного питания (UPS) и программное обеспечение, которое отслеживает состояние UPS и информирует init о том, что питание отключилось;

ctrlaltdel — разрешает init перезагрузить систему, когда пользователь нажимает комбинацию клавиш <Ctrl>+<Alt>+<Del> на клавиатуре. Обратите внимание на то, что системный администратор может определить действия по комбинации клавиш <Ctrl>+<Alt>+<Del>, например, игнорировать нажатие этой комбинации (что вполне разумно в системе, где много пользователей).

Этот список не является исчерпывающим. Более подробно о файле inittab можно узнать из man-страниц init (8), inittab (5) и getty (8).

Обработка файла /etc/inittab процессом init начинается в однопользовательском режиме (уровень 1), в котором единственным пользователем является пользователь root, работающий с консоли. Первым делом init находит строку, которая определяет, какой уровень выполнения запускается по умолчанию:

id:3:initdefault:

Это и будет тот уровень, в котором запустится и будет работать система после загрузки, поэтому естественно, что нельзя указывать в строке initdefault уровни 0 и 6.

Далее init выполняет команды, указанные в строке с ключевым словом sysinit. В стандартной конфигурации здесь выполняется скрипт rc.sysinit из каталога /etc/rc.d. После этого процесс init просматривает файл /etc/inittab и выполняет скрипты, соответствующие однопользовательскому уровню (1 во

втором поле строки), всем уровням (строки с пустым вторым полем) и уровню, заданному по умолчанию. В строке, соответствующей уровню по умолчанию, вызывается скрипт rc из каталога /etc/rc.d. Этот скрипт один и тот же для всех уровней (т. е. обязательно вызывается, на какой бы уровень выполнения не загружалась система), только в зависимости от уровня выполнения ему передается соответствующее значение параметра вызова, так что, например, для 3-го уровня вызов скрипта осуществляется строкой типа
l3:3:wait:/etc/rc.d/rc 3
Функции, выполняемые скриптами rc.sysinit и rc мы подробно рассмотрим ниже, в разд. 8.2.2, а сейчас вернемся к краткому обзору действий процесса init. Следующая важная функция, которую выполняет этот процесс (на уровнях со 2 по 5) — запуск шести виртуальных консолей (процессов getty), чтобы предоставить пользователям возможность регистрироваться в системе с терминалов. Для этого init порождает процессы, именуемые getty-процессами (от "get tty" — получить терминал), и следит за тем, какой из процессов открывает какой терминал. Каждый getty-процесс устанавливает свою группу процессов, используя вызов системной функции setpgrp, открывает отдельную терминальную линию и обычно приостанавливается во время выполнения функции open до тех пор, пока машина не получит аппаратную связь с терминалом. Когда функция open возвращает управление, getty-процесс исполняет программу login (регистрации в системе), которая требует от пользователей, чтобы они идентифицировали себя указанием регистрационного имени и пароля. Если пользователь зарегистрировался успешно, программа login, наконец, запускает командный процессор shell и пользователь приступает к работе. Этот вызов shell именуется "login shell" (регистрационный shell, регистрационный интерпретатор команд). Процесс, связанный с shell, имеет тот же идентификатор, что и начальный getty-процесс, поэтому login shell является процессом, возглавляющим группу процессов.

Если пользователь не смог успешно зарегистрироваться, программа регистрации завершается через определенный промежуток времени, закрывая открытую терминальную линию, а процесс init порождает для этой линии следующий getty-процесс, открывающий терминал вместо прекратившего существование.

После завершения загрузки init продолжает работать в фоновом режиме, отслеживая изменения в состоянии системы. Например, если будет подана команда telinit, позволяющая изменить уровень выполнения, процесс init обеспечит выполнение команд, заданных для нового уровня файлом /etc/inittab. Этот файл прочитывается заново и в случае поступления сигнала HUP; эта особенность избавляет от необходимости перезагружать систему для того, чтобы сделать изменения в начальной конфигурации. Таким образом, процесс начальной загрузки init постоянно находится в оперативной памяти и при получении соответствующих сигналов повторно выполняет цикл чтения из файла /etc/inittab инструкций о том, что нужно делать, причем этот набор инструкций различен для разных уровней выполнения.

Когда суперпользователь останавливает систему (командой shutdown), именно init завершает все другие исполняющиеся процессы, размонтирует все файловые системы и останавливает процессор.

В обычной ситуации процесс init помимо запуска процессов getty выполняет 2 основных действия:

запускает скрипт rc.sysinit из каталога /etc/rc.d;

запускает скрипт rc из того же каталога /etc/rc.d с опцией, равной уровню выполнения (обычно rc 5).

Прежде чем рассматривать функции, выполняемые скриптом rc, надо сказать несколько слов о каталоге /etc/rc.d. Этот каталог вообще играет важную роль в процессе загрузки, поскольку он содержит основные скрипты (программы на языке командного процессора shell), служащие для организации процесса загрузки.

Каталог rc.d содержит следующий набор подкаталогов:

rc0.d

rc1.d

rc2.d

rc3.d

rc4.d

rc5.d

rc6.d

init.d

Если вы просмотрите (например, с помощью команды ls -l) содержимое подкаталогов rcX.d, то увидите, что в этих подкаталогах содержатся не файлы, а только ссылки на файлы скриптов, находящиеся в других каталогах, а именно (за редким исключением), в каталоге /etc/rc.d/init.d. Названия этих ссылок имеют имена, начинающиеся либо с буквы K, либо с буквы S. Подкаталог init.d содержит по одному скрипту для каждой из возможных в системе служб (NFS, sendmail, httpd и т. п.).

Теперь вспомним, что процесс init после скрипта rc.sysinit запускает скрипт rc с опцией, равной заданному уровню выполнения. Этот скрипт предназначен в общем случае для перевода системы из одного уровня выполнения на другой. В процессе начальной загрузки этот скрипт переводит систему из однопользовательского режима на уровень, задаваемый по умолчанию. Общий алгоритм работы rc состоит в следующем. При переходе на уровень X сначала просматривается каталог rcX.d и для всех ссылок, которые начинаются на K, вызываются файлы, на которые идет ссылка, с опцией stop, т. е. осуществляется останов соответствующих служб (которые не должны работать на данном уровне выполнения). Затем запускаются службы, которые на данном уровне выполнения должны быть запущены. Это осуществляется путем последовательного просмотра ссылок, которые начинаются с символа S, и запуска соответствующих скриптов с опцией start.

Из сказанного ясно, что буквы (символы) S и K, с которых начинаются имена ссылок в подкаталогах rcX.d, происходят от start и kill, соответственно. Заметим еще, что после S и K в именах ссылок стоят двузначные номера, которые служат для задания порядка запуска скриптов.

Одна из последних ссылок вида SXXname, используемых скриптом rc на уровнях 2–5, является ссылка на скрипт /etc/rc.d/rc.local. Как сказано в самом этом файле, этот скрипт выполняется после всех других скриптов в процессе инициализации системы, поэтому если вы хотите, чтобы в процессе загрузки были выполнены какие-то дополнительные команды или ваши персональные настройки, то их целесообразно поместить именно сюда.

Тот вариант этого скрипта, который устанавливается из дистрибутива, выполняет очень ограниченные задачи: выводит на экран логотип дистрибутива и формирует файлы /etc/issue и /etc/issue.net, содержащие текст сообщений, выдаваемых пользователю при входе в систему.

Процессы, происходящие при регистрации пользователя

Последовательность событий при полной регистрации выглядит так.

Пользователь вводит регистрационное имя по приглашению login: процесса getty.

getty выполняет программу login, используя в качестве аргумента указанное имя.

login запрашивает пароль и сверяет имя и пароль с записанными в файле /etc/passwd.

login выводит на экран из файла /etc/motd "сообщение дня".

login запускает интерпретатор shell, указанный в бюджете пользователя и устанавливает переменную среды TERM.

shell выполняет соответствующие файлы запуска, после чего выводит на экран приглашение и ожидает ввода информации.

О файлах запуска надо сказать несколько слов дополнительно. В домашнем каталоге пользователя находятся несколько личных файлов

конфигурации. Если таких файлов в домашнем каталоге нет, то после входа в систему будут прочитаны глобальные файлы, содержащие значения "по умолчанию". Если в качестве оболочки используется Bourne-shell, выполняется файл .profile, если C-shell — .login и .cshrc, если Korn-shell — .profile и .kshrc (мы в дальнейшем рассматриваем только случай оболочки bash).

Если вы хотите установить для себя переменные среды (PATH или другие), отличающиеся от тех, которые по умолчанию задаются для всех пользователей, или вы хотите изменить сообщение, которое будет выдаваться вам после входа в систему, или хотите, чтобы после того, как вы войдете в систему, автоматически запускалась какая-то программа, вы можете сделать это с помощью следующих файлов:

/home/your_home/.bashrc — устанавливает ваши алиасы (т. е. псевдонимы или альтернативные имена команд, удобные для упрощения ввода часто используемых команд, имеющих значительную длину из-за большого количества опций) и функции;

/home/your_home/.bash_profile или /home/your_home/.profile — устанавливает переменные среды и запускает ваши программы.

Если такие файлы существуют (заметим, что это скрытые файлы), они будут считаны после входа в систему, и команды, записанные в них, будут выполнены.

Если вы хотите, чтобы при входе пользователя в систему выполнялся какой-то скрипт, то можно вызов этого скрипта поместить в файл ~/.profile. Это может сделать и сам пользователь.

Эти команды будут исполняться только при входе пользователя в систему.

Запуск демонов

Запуск системных сервисов осуществляется скриптом /etc/rc.d/rc, который вызывается с параметром, определяющим уровень запуска. В этом скрипте поочередно вызываются на выполнение все программы и скрипты, ссылки на

которые содержатся в особом каталоге /etc/rc.d/rcN.d, где N — номер уровня выполнения. Ссылки в каталоге /etc/rc.d/rcN.d имеют имена KNNname и SNNname, где NN — порядковые номера, определяющие последовательность запуска скриптов, а name — имя соответствующей программы (это имя приводится, скорее всего, просто для удобства пользователей, его отсутствие ничего бы не изменило). Скрипт /etc/rc.d/rc вначале последовательно (в порядке присвоенных номеров NN) вызывает программы, на которые ссылаются линки KNNname. При этом программы вызываются с аргументом stop, т. е. соответствующие службы останавливаются. Затем так же последовательно перебираются ссылки с именами SNNname и соответствующие программы вызываются с параметром start.

Надо сказать, что существует специальная утилита для управления запуском сервисов (демонов) на разных уровнях выполнения. Она называется chkconfig. Если ее запустить с опцией -- list, вы получите полный список доступных сервисов, с указанием того, запускается или нет данный сервис на каждом уровне. Опции — add и -- del служат для создания или удаления соответствующей ссылки в каталоге /etc/rc.d/rcN.d:

[root]# /sbin/chkconfig [--add | --del] name

Формат команды для запуска или остановки сервиса следующий:

[root]# /sbin/chkconfig [-- level levels] name [on | off | reset]

Так что запуск демона apache можно осуществить командой

[root]# /sbin/chkconfig --level 345 httpd on

Компиляция программ. Препроцессор, транслятор, компоновщик.

Компиляцией чаще всего называют процесс трансляции программы с языка программирования высокого уровня на машинно-ориентированный язык и последующую компоновку программы в готовый к использованию программный модуль. Это означает, что на входе у компилятора исходный

текст программы, а на выходе — работающее приложение, написанное в машинных кодах. Этот процесс необходим, поскольку человеку гораздо проще писать и читать программы в виде, близком к человеческой речи. Машины же оперируют только понятиями команд и данных — и те и другие представляют собой числа в двоичной системе счисления.

Как следует из определения, этот процесс состоит из двух важных фаз — **трансляция** и **компоновка**. Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определённом смысле, равносильную первой. Трансляция в большинстве случаев процесс необратимый — например, машинный код не содержит имен переменных. Результатом трансляции программы на языке Си является *объектный код*, файл с которым называют *объектным модулем* (либо объектным файлом). Объектные файлы представляют собой блоки машинного кода и данных, с неопределенными адресами ссылок на данные и процедуры в других объектных модулях, а также список своих процедур и данных. Именно из-за неопределенности адресов объектные модули невозможно сразу выполнить, хоть они и содержат код, понятный процессору. Обычно один объектный файл получается трансляцией одного исходного файла на языке Си.

Для того, чтобы из объектных модулей получить исполняемый файл, используется **компоновка**. Компоновщик (редактор связей, linker) принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль. Для связывания модулей компоновщик использует таблицы имён, созданные транслятором в каждом из объектных модулей. Такие имена могут быть двух типов:

- определённые или экспортируемые имена — функции и переменные, определённые в данном модуле и предоставляемые для использования другим модулям;
- неопределённые или импортируемые имена — функции и переменные, на

которые ссылается модуль, но не определяет их внутри себя

Работа компоновщика заключается в том, чтобы в каждом модуле разрешить ссылки на неопределённые имена. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени заменяется на его адрес. Компоновщик обычно не выполняет проверку типов и количества параметров процедур и функций. Если надо объединить объектные модули программ, написанные на языках со строгой типизацией, то необходимые проверки должны быть выполнены дополнительной утилитой перед запуском редактора связей.

Итак, исходные тексты на всех языках высокого уровня проходят две обязательных фазы в процессе компиляции — трансляцию и компоновку. Исходные тексты на языках Си и Си++, кроме того, подвергаются предварительной обработке — **препроцессингу**. Препроцессор — это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора). О данных на выходе препроцессора говорят, что они находятся в препроцессированной форме, пригодной для обработки последующими программами (компилятор). Подробнее о препроцессоре и компоновщике речь пойдет в следующем разделе.

Для компиляции программ в ОС GNU/Linux обычно используется набор компиляторов GCC - GNU Compiler Collection. GCC является свободным программным обеспечением, распространяется фондом свободного программного обеспечения (FSF) на условиях GNU GPL и GNU LGPL и является ключевым компонентом GNU toolchain. GCC изначально назывался GNU C Compiler, и поддерживал только язык Си. Позднее, GCC был расширен для компиляции исходных кодов на таких языках программирования как C++, Objective-C, Java, Фортран и Ada. GCC почти полностью написан на Си.

Внешний интерфейс GCC является стандартом для компиляторов на

платформе UNIX. Пользователь вызывает управляющую программу, которая называется gcc. Она интерпретирует аргументы командной строки, определяет и запускает для каждого входного файла свои компиляторы нужного языка, запускает, если необходимо, ассемблер и компоновщик.

Компилятор каждого языка является отдельной программой, которая получает исходный текст и порождает вывод на языке ассемблера. Все компиляторы имеют общую внутреннюю структуру: front end, который производит синтаксический разбор и порождает абстрактное синтаксическое дерево, и back end, который конвертирует дерево в Register Transfer Language (RTL), выполняет различные оптимизации, затем порождает программу на языке ассемблера, используя архитектурно-зависимое сопоставление с образцом.

По умолчанию gcc пытается совершить весь процесс компиляции за один запуск. Например, создадим исходный файл hello.c следующего содержания:

```
#include <stdio.h>
int main() {
  printf( "Hello, World!\n" );
  return 0;
};
```

и попробуем скомпилировать его:

```
[user]$ gcc -o hello hello.c
```

Аргументы «-o hello» означают, что полученный исполняемый файл будет сохранен с именем «hello».

Однако часто бывает удобнее разделить фазы трансляции и компоновки. Для этого используется флаг «-c», сообщающий компилятору о том, что необходимо остановиться на фазе трансляции, и не делать компоновку.

```
[user]$ gcc -c hello.c
[user]$ ls
```

hello.c hello.o

По умолчанию gcc создает объектный файл с именем исходного файла, заменив суффикс на «.o».

Компиляция программ. Препроцессор, компилятор, линкер.

Компиляция программ. Три основных фазы компиляции: препроцессинг, компиляция, линковка. Типы входных и выходных данных. GCC, использование для компиляции программ для GNU/Linux. Компилятор. Ключи -c и -o. Предупреждения и ошибки. Ключи -W.

Линковка, препроцессинг.

Линковка. Назначение, типичные проблемы. Статические и разделяемые библиотеки. Ключи -L и -l. Препроцессор. Директивы #include, #define и #ifdef/#else/#endif.

Сборка программ из исходных текстов

Сборка. Использование make для автоматизации процесса компиляции. configure-скрипт, сборка программ из исходных текстов.

Кросс-компиляция

Кросс-компиляция. Набор утилит ELDK. Использование директив препроцессора для портирования приложений. Конфигурирование исходных текстов программ для кросс-компиляции.

ЦСП. Виды ЦСП и их специализация. PNX 8950.

Виды ЦСП и специализация. Параметры: скорость, арифметика, контроллеры, потребляемая мощность, цена. Области применения: мобильные устройства, домашние, технологические устройства.

Особенности PNX 8950. Параметры. Ядра: MIPS и TriMedia. Операционные системы Linux на MIPS и PSOS на TriMedia. Сопроцессоры MBS и QVCP. Взаимодействия между архитектурными компонентами, прокси стабы и

события. Иерархия управления.

Разработка программного обеспечения для ЦСП PNX 8950

Подготовка к работе. Подключение к Ethernet и COM порту. Параметры загрузки, меню bootloader-а. Расположение файлов и настройка рабочего окружения, создание прошивок, возможность изменения меню bootloader-а.

Компиляция кода ANSI C. Настройка переменного окружения. Компиляция отдельно взятого компонента. Создание make-файла проекта. Стандарты по размещению файлов и названию компонент. Готовые компоненты PNX SDK и Elecard SDK.

Использование симулятора. Флаги компилятора. Симулирование. Отслеживание состояний. Профилирование отдельных частей кода.

Программа — образец. Использование стандартных средств отладки: DBG_PRINT, DBG_ASSERT, DBG_ERROR.

Модульная архитектура TSSA.

TSSA. Управление потоками. Аналоги для ОС Windows.

Интерфейс компонентов. Набор функций в зависимости от функциональности компонента. Основные компоненты для работы с файлами, ввод/вывод аудио и видео. Компоненты компании Элекард.

Графы компонент. Последовательность соединённых компонент, предназначенная для решения конкретной задачи обработки мультимедиа данных.

Взаимодействие компонентов в графе. Виды соединений. Управление приоритетами. Установка ресурсов для компоненты.

Использование ассемблера для ЦСП PNX 1500

Регистры. Системные, пользовательские. Рекомендации к использованию

Команды. Синтаксис команд и их время выполнения. Описание команд: арифметические, логические, сдвига, сравнения, перехода, преобразования

данных.

АЛУ. Свойства. Список допустимых операций для каждой. Пример на языке ассемблер.

Конвейерное выполнение команд. Использование. Определение трех стадий конвейера. Разбор примера программы на ассемблере.

Пространство ядра и пространство пользователя

Память в современных операционных системах разделяется на два особых региона: пространство ядра и пространство пользователя. Оба пространства работают с оперативной памятью. Пространство ядра — область, где исполняется и предоставляет свои сервисы ядро, расширения ядра, а также драйвера устройств. Пространство пользователя — область, где исполняются пользовательские приложения (все то, что не ядро). Каждый процесс в пространстве пользователя обычно выполняется в собственной области виртуальной памяти и при отсутствии явной необходимости не может получить доступа к памяти, используемой другими процессами. Такой подход является базисным для обеспечения защиты памяти большинства современных операционных систем и своего рода фундаментом для обеспечения права доступа. В зависимости от привилегий процесс может запросить ядро отобразить часть адресного пространства другого процесса на свое, как, например, это делают отладчики. Программы также могут запрашивать для себя область разделяемой памяти (англ. shared memory) совместно с другими процессами. Одной из задач ядра является управление памятью процессов так, чтобы не было пересечений областей памяти двух разных процессов.

Пространство ядра напрямую отображается в физическую память. Поэтому драйвера имеют прямой доступ к регистрам устройств и регистрам интерфейсов.

Виртуальная память

В основе виртуальной памяти лежит идея, что у каждой программы имеется свое собственное адресное пространство, которое разбивается на участки, называемые страницами. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы присутствие в памяти всех страниц не обязательно. Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету. Когда программа ссылается на часть своего адресного пространства, которое не находится в физической памяти, операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду.

Большинство систем виртуальной памяти использует технологию под названием страничная организация памяти (paging).

Технология виртуальной памяти позволяет программе использовать больше памяти, чем установлено в компьютере, за счет откачки неиспользуемых страниц на вторичное хранилище (жесткий диск).

При использовании виртуальной памяти виртуальные адреса не выставляются напрямую на шине памяти. Вместо этого они поступают в диспетчер памяти (MMU, Memory Management Unit), который отображает виртуальные адреса на адреса физической памяти, как показано на рис.1.

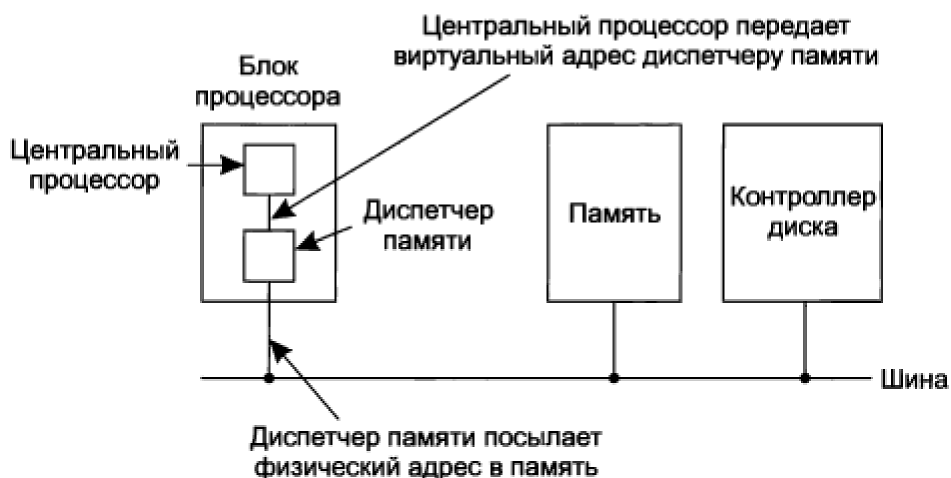

Рис1. Расположение и предназначение диспетчера памяти (MMU).

Диспетчер памяти показан в составе микропроцессора – это неотъемлемая часть системы на кристалле.

Файлы символьных устройств

Имеются два главных пути для общения модуля с процессами. Первый идет через файлы устройства (подобно файлам в каталоге /dev), другой должен использовать файловую систему proc. Рассмотрим файлы устройства.

Первоначальная цель файлов устройства состоит в том, чтобы позволить процессам связываться с драйверами устройства в ядре, и через них с физическими устройствами (модемы, терминалы, и т.д.).

Каждый драйвер устройства, который является ответственным за некоторый тип аппаратных средств, имеет собственный главный номер (majornum). Список драйверов и их главных номеров доступен в /proc/devices. Каждое физическое устройство, управляемое драйвером устройства, имеет малый номер (minornum). Каталог /dev включает специальный файл, названный файлом устройства (нод, node), для каждого из тех устройств, которые реально установлены в системе.

Например, командf ls -l /dev/hd[ab]* выводит список всех IDE разделов жесткого диска, которые могли бы быть связаны с машиной. Обратите внимание, что все они используют тот же самый главный номер, 3, но малые номера у каждого свои.

Файлы устройств создаются командой mknod. Не имеется никакой технической причины, по которой они должны быть в каталоге /dev, это только полезное соглашение.

Устройства разделены на два типа: символьные и блочные. Различие в том, что блочные имеют буфер для запросов, так что они могут выбирать, в каком порядке им отвечать. Это важно в случае устройств памяти, где скорее понадобится читать или писать сектора, которые ближе друг к другу, чем те, которые находятся далеко. Другое различие: блочные устройства могут

принимать ввод и возвращать вывод только в блоках (чей размер может измениться согласно устройству), в то время как символьные устройства могут использовать столько байтов, сколько нужно. Большинство цифровых устройств в мире можно отнести к символьному типу, потому что они не работают с фиксированным размером блока. Узнать, является ли устройство блочным или символьным, можно по первому символу в выводе ls -l. Если это "b", значит, устройство блочное, а если "c", то символьное.

Перемещение данных из пространства ядра в пространство приложений и обратно

Из-за использования пространством приложений виртуального адресного пространства при перемещении данных в пространство ядра (где используются физические адреса) нужно импортировать данные. Для этого используются функции get_user и put_user для перемещения одного байта, и copy_from_user и copy_to_user для перемещения блока данных (asm/uaccss.h).

Модуль символьного устройства

Каждый модуль ядра должен состоять минимум из двух функций: init_module – инициализация модуля, и cleanup_module – деинициализация модуля. Либо воспользоваться макросами module_init() и module_exit().

В nit_module регистрируется устройство module_register_chrdev, чтобы добавить драйвер устройства к символьной таблице драйверов устройств ядра. Этот вызов также возвращает главный номер, который нужно использовать для драйвера. Функция cleanup_module вычеркивает из списка устройство.

```
static struct file_operations fops = {
	.read = device_read,
	.write = device_write,
	.open = device_open,
	.release = device_release
```

```
};
// This function is called when the module is loaded
int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);
        if (Major < 0) {
                printk(KERN_ALERT "Registering char device failed with %d\n", Major);
                return Major;
        }
        printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
        return SUCCESS;
}
// This function is called when the module is unloaded
void cleanup_module(void)
{
        int ret = unregister_chrdev(Major, DEVICE_NAME);
        if (ret < 0)
                printk(KERN_ALERT "Error in unregister_chrdev: %d\n", ret);
}
```

Действия в ядре не выполняются по собственной инициативе, подобно процессам, а вызываются процессами через системные вызовы или аппаратными устройствами через прерывания или другими частями ядра (просто вызывая специфические функции). В результате, когда вы добавляете код к ядру, вы регистрируете его как драйвер для некоторого типа события, а когда удаляете его, то отменяете регистрацию.

Драйвер устройства выполняет четыре действия (функции open, read, write, close), которые вызываются, когда кто-то пробует делать что-либо с

файлом устройства, который имеет главный номер. Ядро знает, что вызвать их надо через структуру file_operations, Fops, который был дан, когда устройство было зарегистрировано, включает указатели на четыре функции, которые данное устройство выполняет.

Нужно помнить, что мы не можем позволять модулю выгружаться командой rmmod всякий раз, когда root захочет его выгрузить. Причина в том, что если файл устройства открыт процессом и мы удаляем модуль, то использование файла вызвало бы обращение к точке памяти, где располагалась соответствующая функция. Если мы удачливы, никакой другой код не был загружен в эту область, то мы получим уродливое сообщение об ошибках. Если мы неудачливы (обычно так и бывает), другой модуль был загружен в то же самое место, это будет означать переход в середину другой функции внутри ядра. Результаты этого невозможно предсказать, но они не могут быть положительны.

Обычно, когда вы не хотите выполнять что-либо, вы возвращаете код ошибки (отрицательное число) из функции, которая делает данное действие. С cleanup_module такой фокус не пройдет: если cleanup_module вызван, модуль завершился. Однако имеется счетчик использований, который считает, сколько других модулей используют этот модуль, названный номером ссылки (последний номер строки в /proc/modules). Если это число не нулевое, rmmod будет терпеть неудачу. Счетчик модульных ссылок доступен в переменной mod_use_count_. Так как имеются макрокоманды, определенные для обработки этой переменной (MOD_INC_USE_COUNT и MOD_DEC_USE_COUNT), лучше использовать их, а не mod_use_count_ непосредственно.

```
static int device_open(struct inode *inode, struct file *file)
{
	if (Device_Open)
		return -EBUSY;
```

```
        Device_Open++;
        try_module_get(THIS_MODULE);
        return SUCCESS;
}
static int device_release(struct inode *inode, struct file *file)
{
        Device_Open--;          /* We're now ready for our next caller */
        module_put(THIS_MODULE);
        return 0;
}
// Called when a process, which already opened the dev file, attempts to read from
it.
static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
                           char *buffer,        /* buffer to fill with data */
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{
        return copy_to_user(buffer, “Hello!!!\n”, 10);
}

// Called when a process writes to dev file: echo "hi" > /dev/hello
static ssize_t device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
        printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
        return -EINVAL;
}
```

Файловая система /proc

procfs — это виртуальная файловая система (process — процесс). Основная концепция этой файловой системы заключается в том, что для каждого процесса системы создается каталог в каталоге /proc. Имя каталога формируется из PID процесса в десятичном формате. Например, /proc/619 — это каталог, соответствующий процессу с PID 619. В этом каталоге находятся файлы, которые хранят информацию о процессе — его командную строку, строки окружения и маски сигналов. В действительности этих файлов на диске нет. Когда они считываются, система получает информацию от реального процесса и возвращает ее в стандартном формате.

Многие расширения, реализованные в операционной системе Linux, относятся к другим файлам и каталогам, расположенным в каталоге /proc. Они содержат информацию о центральном процессоре, дисковых разделах, устройствах, векторах прерывания, счетчиках ядра, файловых системах, подгружаемых модулях и о многом другом. Непривилегированные программы пользователя могут читать большую часть этой информации, что позволяет им узнать о поведении системы (безопасным способом). Некоторые из этих файлов могут записываться в каталог /proc, чтобы изменить параметры системы.

Для создания каталогов и файлов в /proc, в ядре присутствуют функции (include/linux/proc_fs.h):

```
struct proc_dir_entry *proc_mkdir(const char *,struct proc_dir_entry *);
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent);
```

Возвращенному указателю на файл можно задать функции чтения и/или записи.

```
static int proc_sample_read(char *page, char **start, off_t off, int count, int *eof, void *data)
{
	printk(“Proc read func\n”);
```

```
	return len;
}
static int proc_sample_write(struct file *file, const char __user *buffer, unsigned long count, void *data)
{
	printk("Proc write func\n");
	return count;
}
…….
res = create_proc_entry("sampleName", S_IWUSR | S_IRUGO, NULL);
res->read_proc = proc_sample_read;
res->write_proc = proc_sample_write;
```

Линейка цифровых сигнальных процессоров фирмы TI

Компания Texas Instruments (далее TI) производит широкий спектр цифровых сигнальных процессоров серии TMS320. Практически все процессоры характеризуются пониженным энергопотреблением, что обусловлено применением ЦСП во встраиваемых системах. Ниже приведен краткий список процессоров TI:

TMS320C2xxx — 16-и и 32-х битные DSP, оптимизированные для применения в схемах управления.

TMS320C5xxx — 16-битный целочисленный DSP (от 50 до 300 МГц) с пониженным энергопотреблением.

TMS320C6xxx — семейство высокопроизводительных DSP, от 300 до 1200 МГц.

Многоядерные процессоры:

OMAP — микропроцессоры, предназначенные для мультимедийных приложений. Некоторые из них содержат процессорные ядра C55, ARM7, ARM9 или ARM11.

DaVinci — микропроцессоры, содержащие ядро C64x+, ARM9 и специализированные сопроцессоры для обработки видеоданных.

Знакомство с процессорами DaVinci TI

DSP DaVinci — система на кристалле (SoC, System on Chip), включающая в себя два ядра, сопроцессоры и набор периферийных подсистем. Первое ядро представляет из себя процессор общего назначения ARM-архитектуры, на этом процессоре работает ОС Linux и осуществляется взаимодействие и контроль со вторым ядром. Работает на частоте 300МГц. Второе ядро — это DSP процессор TMS320C64x+ (построен на архитектуре VLIW), предназначенный для обработки аудио/видео данных, работает на частоте 600МГц.

Существуют различные реализации процессоров DaVinci: DM644x, DM6467, DM35x, DM36x. Они отличаются друг от друга набором сопроцессоров. DM6443, DM6446 — предназначены для декодирования видео стандартного разрешения (720x576@25), а DM6467 — для кодирования/декодирования видео высокой четкости (1280x720@30).

Обзор SoC **TMS320DM6446**.

На рис. 2 видно, что ARM-ядро может обмениваться сообщениями с DSP-ядром посредством использования общих ресурсов (специальный участок оперативной памяти), но только DSP-ядро может обращаться к сопроцессору VICP. ARM-ядро является управляющим, оно осуществляет запуск и контроль над DSP.

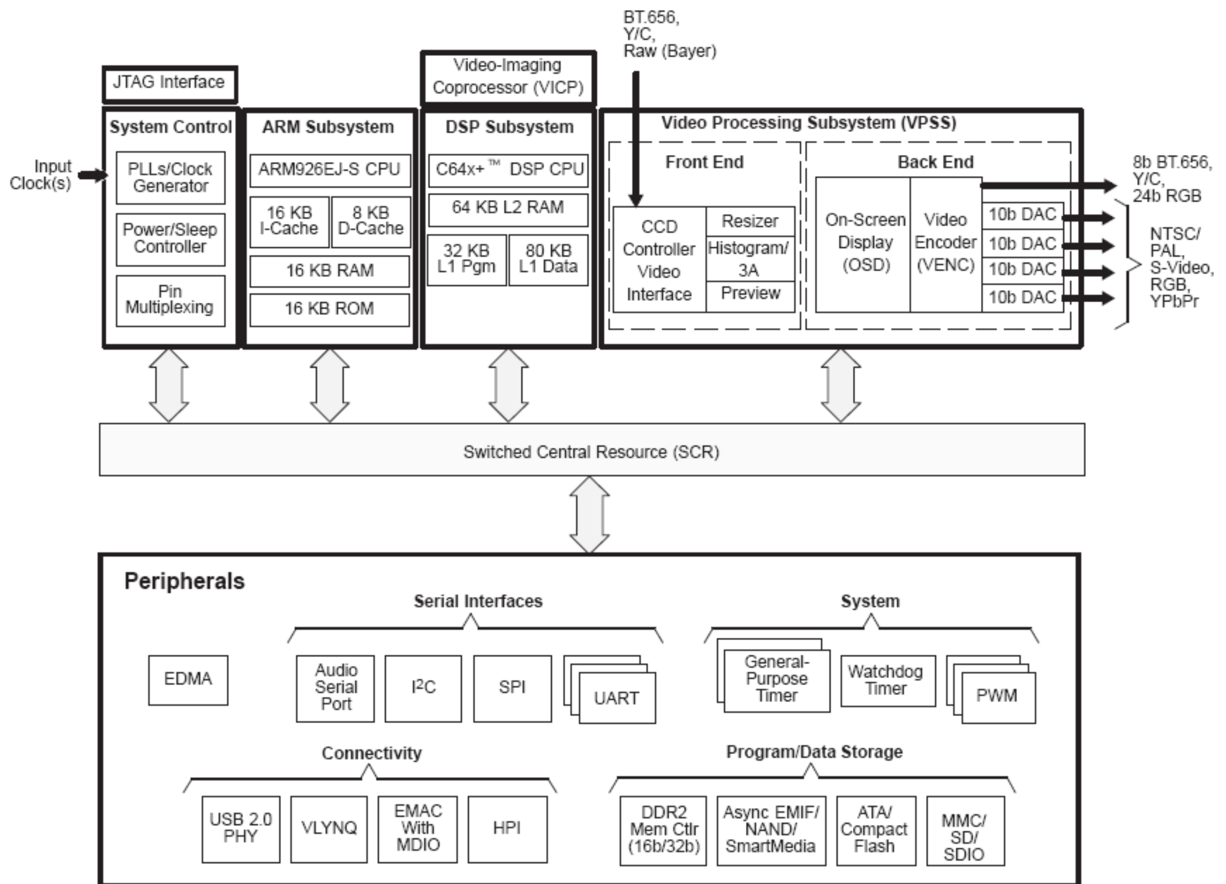

Рис 2 Функциональная блок-диаграмма TMS320DM6446

В SoC также присутствуют практически все необходимые (для цифровой телевизионной приставки) интерфейсы: DDR2, USB2.0, audio и video подсистемы вывода, интерфейсы для различного рода ПЗУ (nand flash, ata) и др.

DSP-ядро TMS320C64x+ - второе поколение высокопроизводительных VLIW процессоров. Состоит из восьми функциональных блоков, двух наборов регистров общего назначения (32 32-х битных регистра в каждом наборе). Регистр может представлять из себя: 4 четыре 8-и битных значения, два 16-и битных значения или же одно 32-х битное. Также возможна работа с 40- и 64-х битными данными, в этом случае используется пара регистров. В ядре присутствуют кэш память первого и второго уровней (L1Data, L1Program, L2). Кэши L1Data и L2 доступны для конфигурации: возможно прямое использование кэш памяти в алгоритмах, как для данных, так и для программы.

Каждый из восьми функциональных блоков (M1, L1, D1, S1, M2, L2, D2, S2) способен выполнить одну инструкцию за цикл. У всех блоков есть инструкции сложения, логические и сдвиговые. Помимо этих инструкций, каждый блок имеет дополнительные возможности. M1 и M2 предназначены для различных операций умножения: одно 32x32 битное, два 16x16 битных или четыре 8x8 битных умножения. Также присутствуют специфические операции, присущие алгоритмам кодирования/декодирования изображений.

L1 и L2 блоки имеют различного рода логические инструкции;

D1 и D2 – основная задача сводится в загрузке-выгрузке данных из оперативной памяти в регистровые файлы;

S1 и S2 – имеют инструкции паковки/распаковки данных.

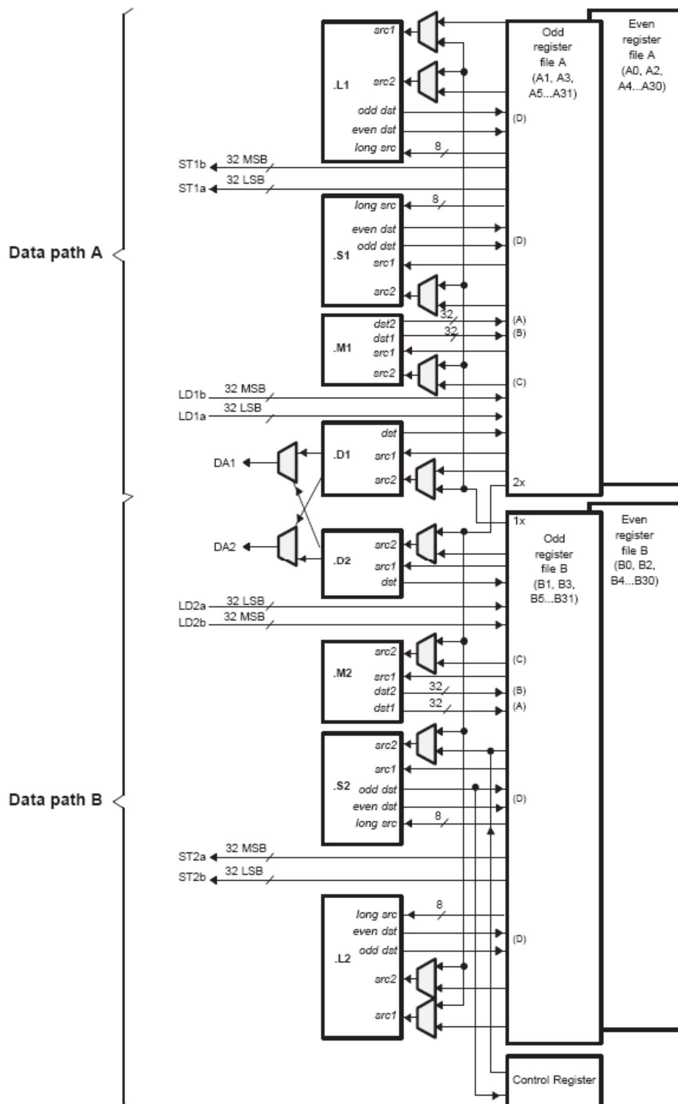

Рис 3 Диаграмма функциональных блоков C64+.

На рис. 3 изображены функциональные блоки, регистровые файлы и шины данных между ними. Из рисунка видно, что L1, S1 и M1 имеют доступ только до регистрового файла A, а L2, S2 и M2 только до регистрового файла B. Перемещением данных между регистровыми файлами занимаются D1 и D2.

Внутри DSP также присутствует свой собственный DMA – IDMA. Он может работать со всеми кэшами DSP и с регистрами, расположенными на SoC.

Во время работы на DSP управляющие функции берет на себя DSP BOIS – базовая операционная система. У DSP и ОС есть возможность обращаться только к оперативной памяти (поэтому в ОС нет поддержка файловых систем и она не может работать с файлами). Межпроцессорное взаимодействие осуществляется через оперативную память, путем выделения специальной области памяти для обмена сообщениями и пулами данных.

Средства разработки программного обеспечения от фирмы TI

Для разработки программного обеспечения существуют кросс-компиляторные системы как для ARM-процессора, так и для DSP.

Средства разработки для ядра ARM.

ARM-архитектура широко распространена, поэтому под эти процессоры существует множество кросс-платформенных средств разработки. Мы будем использовать от фирмы MontaVista. Основные полезные части (для нас) в пакете: кросс-компиляторы, собранные основные Linux утилиты (которые образуют полноценную файловую систему), а также исходные коды загрузчика u-boot и ядра Linux.

Примечание. На практических занятиях проводится разбор создания программы «hello world» на стороне ARM.

Средства разработки для ядра DSP.

Для C64+ Ti поставляет набор программ DVSDK, в который входят:

DSP BIOS – операционная система;

cg6x – кросс-компиляторы для ядра C64+;
codec engine – набор библиотек и утилит для работы с DSP BIOS, сопроцессорами, DMA, cmem и другими частями SoC;
dsplink – набор библиотек для инициализации, загрузки и контроля ОС DSP BIOS, со стороны Linux;
xdctools – система сборки приложений.

Примечание. На практических занятиях проводится разбор создания программы «hello world» на стороне DSP.

Модульная архитектура XDAIS

eXpressDSP – стандарт написания алгоритмов для DSP фирмы TI (также известный как XDAIS). Стандарт придуман для более легкой миграции алгоритмов между процессорами одного семейства.

В соответствии со стандартом, алгоритмы должны реализовываться с IALG интерфейсом. IALG-интерфейс – абстрактный интерфейс сервисов, объявленный в ialg.h. В дополнение к IALG-интерфейсу, алгоритмы не должны напрямую обращаться к периферии. IALG - интерфейс определяется таблицей IALG_Fxns. Каждый алгоритм должен объявить и инициализировать эту таблицу. В таблице назначаются указатели на базовые функции, часть из которых необязательны. Обязательные функции: algAlloc(), algInit() и algFree(). Эти функции должны быть определены в реализуемом алгоритме.

algAlloc() – функция запроса памяти. По стандарту требуется, чтобы все области памяти, используемые в алгоритме, были объявлены в этой функции и занесены в соответствующую таблицу. В таблице возможно указать местоположение выделяемой области памяти (в случае, если в запрашиваемой области нет свободной памяти достаточного размера, память будет выделена в DDR).

Int MyALG_Elc_algAlloc(const IALG_Params *algParams, IALG_Fxns **pf,

```
IALG_MemRec
memTab[])
{
/* Request memory for MyAlgorythm instance object */
memTab[0].size = sizeof(MyALG_Elc_Obj);
memTab[0].alignment = 0;
memTab[0].space = IALG_DARAM0;
memTab[0].attrs = IALG_PERSIST;
return(1) /* return number of memory blocks requested */
}
```

algInit() – функция инициализации. Вызывается после algAlloc(). В этой функции происходит начальная инициализация данных.

```
Int MyALG_Elc_algInit(IALG_Handle handle, const IALG_MemRec memTab[],
IALG_Handle
p, const IALG_Params *algParams)
{
MyALG_Elc_Obj *enc = (Void *)handle;
const MyALG_Elc_Params *params = (Void *)algParams;
if (params == NULL) {
params = &MYALG_PARAMS; /* set default parameters */
}
/* Copy creation params into the object */
enc->workingRate = params->rate;
return(IALG_EOK);
}
```

algFree() – функция вызывающаяся по завершении выполнения алгоритма. В этой функции необходимо заполнить таблицу выделенной памяти, аналогично algAlloc().

```
Int MyALG_Elc_algFree(IALG_Handle handle, IALG_MemRec memTab[])
```

```
{
MyALG_Elc_Obj *enc = (Void *)handle;
algAlloc(NULL, NULL, memTab); /* Fill the memTab struct */
memTab[0].base = (Void *)&enc;
return(1);
}
```

Стандарт eXpressDSP позволяет избежать использования языка C++ там, где необходим упор на производительность алгоритмов, вместе с тем позволяет достичь корректного использования ресурсов. В результатеудается запускать несколько экземпляров алгоритма одновременно.

Видео подсистема. Framebuffer. DirectFB

Кадровый буфер (framebuffer) — реальное или виртуальное электронное устройство, или область памяти для кратковременного хранения одного или нескольких кадров в цифровом виде перед его отправкой на устройство видеовывода. В системах на кристалле используется для изображения интерфейса пользователя. Framebuffer может поддерживать функцию double buffering. Что позволяет, рисуя графический интерфейс на одном из кадров, отображать другой (достигается более четкая смена кадров интерфейса при переключении страниц).

Framebuffer предоставляет низкоуровневые функции, такие как: задания цветовой схемы (RGB, ARGB, BGR и т.д.), глубины цвета (8, 16, 24 или 32 бит), а также предоставляет область памяти для рисования. Рисование производится копированием в память значений пикселей.

DirectFB - это графическая библиотека, которая была разработана с учетом особенностей встроенных систем. Библиотека предоставляет программистам, работающим с аппаратными графическими ускорителями, средства управления устройствами ввода на абстрактном уровне, а также многослойный вывод поверх аппаратного видеобуфера (framebuffer) под

Linux. DirectFB представляет собой абстрактное представление аппаратного обеспечения с программной реализацией всех графических операций, не поддерживаемых нижележащим «железом».

Видео подсистема DaVinci 644x позволяет работать с двумя окнами видео (для реализации функции картинка в картинке) и с двумя графическими видеобуферами (framebuffer). На рис.4 показан приоритет наложения видео и графических слоев видео подсистемы. Встроенный resizer позволяет позиционировать окна относительно друг друга. Также для каждого из окон устанавливается общая прозрачность.

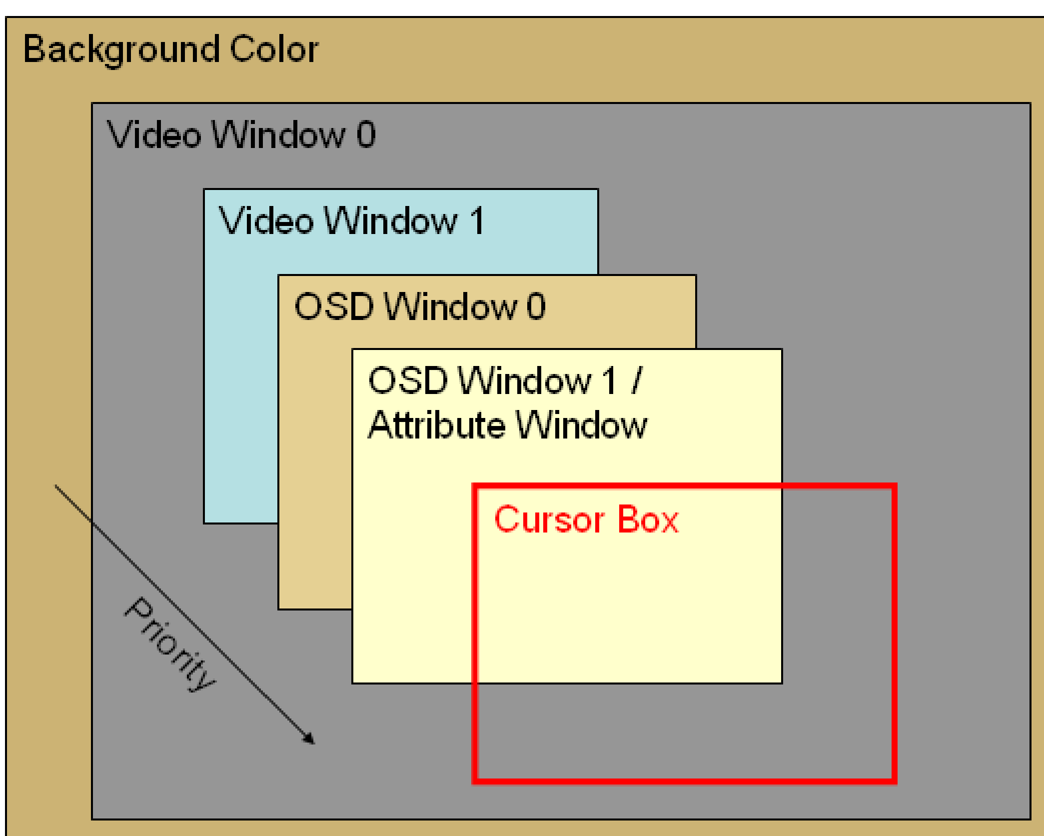

Рис.4. Видео микшер DaVinci 644x.

Оптимизация C кода и директивы компилятора при разработке для ЦСП

Кросс-компилятор для TI DSP C64x+ позволяет выполнять оптимизацию кода.

Флаг компилятору --gen_profile_info позволяет создать профилировочную информацию для компилированного кода. В профилировку входит: время выполнения функций и использование памяти. После компиляции программы ее нужно запустить с рабочими данными (приближенными к реальности). Полученные данные нужно смотреть с помощью утилиты pdd6x.

Прагма MUST_ITERATE (min, max, multiple). Ставится перед началом циклов. Говорит о том, что цикл будет выполнен минимум min раз и максимум max раз. Multiple – кратность количества выполнения циклов, если больше 1, компилятор разворачивает цикл. Значения могут быть пропущены.

Прагма UNROLL(n). Указывает компилятору на принудительное разворачивание цикла, перед которым стоит прагма, в указанное число n раз.

Прагма DATA_SECTION (symbol , " section name "). Указывает линкеру, что symbol надо поместить в указанную секцию. Это полезно, если необходимо поместить часто используемые данные в один из кэшей DSP. Тогда будет достигнут прирост производительности.

При чтении и записи следует использовать выровненные данные, поскольку доступ к невыровненным данным заменяется на последовательность чтения/записи данных, а также наложение масок и сдвиги.

Интринсиксы – это аналоги ассемблерных команд, доступные в C. В интринсинксах реализовани арифметические операции с 16-ти- и 32-х-разрядными данными, а также SIMD (ingle instruction, multiple data) операции.

Использование инлайн функций иногда может быть полезным, поскольку вызываемой функции может и не быть в program cache, а включение функции инлайном подставляет ее в текущую функцию. Но это может повлечь к увеличению кода программы. Компилятор будет подставлять инлайн функции только при уровне оптимизации O2 или O3.

Структура ассемблера для С64х. Использование специализированных инструкций ЦСП

Структура ядра. Возможности АЛУ. Связи внутри ядра, набор команд.

Синтаксис ассемблера, директивы, написание параллельного кода, компиляция, отладка.

Этапы трансляции С кода в параллельный ассемблер.

Трудности профилирования. Расчет трудоемкости. Построение графа зависимостей для описания движения данных внутри алгоритма.

Использование ассемблера для ЦСП С64х

Разбор примера использования построения правильного программного конвейера на задаче вычисления КИХ и БИХ фильтра.

Расчет входа и выхода в тело цикла. Определение минимального интервала итерации в тактах.

Литература

1. В. В. Корнеев, А. В. Киселев. Современные микропроцессоры: учебное пособие/ - 3-е изд., перераб. и доп. - СПб. : БХВ-Петербург, 2003. - 440 с.
2. Костромин, Виктор. Самоучитель Linux для пользователей : учебное пособие. - СПб. : БХВ-Петербург, 2003. - 648[10] с.

1. PNX 15xx/952x Series Data Book. Connected Media Processor. NXP. 2007. p. 830.
2. TM3260 Architecture Databook. Trimedia VLIW Core. Philips. 2004. p. 480.
3. The Architecture of TSSA1. NDK SR 6.0. NXP. 2009. p. 190.
4. TSSA1 – Classic APIs. NDK SR 5.4. NXP. 2006. p. 476
5. Getting Started TCS 5.2. NXP. 2008. p. 44.
6. CookBook TCS 5.2. NXP. 2008. p. 218.
7. Compilation Tools TCS 5.2. NXP. 2008. p. 364.

8. Trimedia Simulator TCS 5.2. NXP. 2008. p. 330.
9. Murat Karaorman, Vincent Wan Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6X1X. Texas Instruments, SantaBarbara. 2004. p. 36.
10. Эндрю Таненбаум. Современные операционные системы. ISBN 5-318-00299-4
11. TMS320C6000 Programmer's Guide. Texas Instruments, Dallas. 2002. p. 382.
12. TMS320DM6443 Digital Media System-on-Chip. Texas Instruments, Dallas. 2010. p. 218
13. TMS320DM6446 Digital Media System-on-Chip. Texas Instruments, Dallas. 2010. p. 227
14. TMS320DM6441 Digital Media System-on-Chip. Texas Instruments, Dallas. 2010. p. 235
15. TMS320C6000 Optimizing Compiler v 6.0 User's Guide. Texas Instruments, Dallas. 2005. p. 284
16. TMS320C6000 Optimizing C Compiler Tutorial. Texas Instruments, Dallas. 2002. p. 71
17. TMS320C6000 DSP Cache User's Guide. Texas Instruments, Dallas. 2003. p. 147
18. TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide. Texas Instruments, Dallas. 2006. p. 883
19. Дж. Фон Нейман Теория самовоспроизводящихся автоматов. М.: Мир, 1971. – 382 с.
20. Мур Г. Ничто не бесконечно, но предел можно отодвинуть! // Chip News. 2003. № 2.
21. Волин В., Рудометов В., Столярский Е. Организация подкачки кода в VLIW-процессоре // Информационные технологии и вычислительные системы. 1999. № 1, с. 58-64.

22.PNX17xx Series Data Book. Connected Media Processor. NXP. 2007. p. 832.
23.TM5250 User Manual. Trimedia VLIW Core. NXP. 2005. p. 490.
24.TM3282 User Manual. Trimedia VLIW Core. NXP. 2009. p. 453
25.TC_STR0 User Manual. Trimedia Streaming Coprocessor. NXP. 2008. p. 37
26.PNX1005 Media Processor. NXP. 2009. p. 66.