УДК 004.451

ОПЕРАЦИОННАЯ СИСТЕМА ДЛЯ ВСТРАИВАЕМЫХ УСТРОЙСТВ

Зайцев С.А., Корниенко С.К.

Запорожский национальный технический университет

Проведен анализ рынка современных операционных систем для встраиваемых устройств. Разработано ядро многозадачной операционной системы, предназначенной для управления встраиваемым устройством. Описаны основные принципы функционирования встраиваемой операционной системы и предложены модели разработки встраиваемых приложений и драйверов.

Проведено аналіз ринку сучасних операційних систем для вбудованих пристроїв. Розроблено ядро багатозадачної вбудованої операційної системи, що призначена для керування вбудованим пристроєм. Описано основні принципи функціонування вбудованої операційної системи та моделі розробки вбудованих додатків та драйверів.

Modern embedded operating system market analysis has been performed. Multitasking operating system kernel for embedded device control has been developed. Major embedded operating system functioning concepts has been described, embedded applications and driver programming model has been suggested.

Введение

Быстрый рост рынка портативных компьютеров, интеллектуальных устройств, сетевых компьютеров и т. д. стимулирует интерес к встраиваемым системам. Встраиваемая система (*embedded system*) — это специализированная компьютерная система, в которой сам компьютер обычно встроен в устройство, которым он управляет. Примерами встраиваемых систем могут служить банкоматы, авионика, карманных персональных компьютеров (КПК), телекоммуникационное оборудование и тому подобные устройства.

Целью данной работы является разработка операционной системы (ОС) для встраиваемых решений, основанных на процессорах семейства Advanced RISC Machines (ARM).

1. ОС встраиваемой системы

Программным обеспечением для встраиваемой системы является микропрограмма (англ. *firmware*). Около 75% встраиваемых решений используют какие-либо ОС, и лишь остальные 25% реализованы в виде самостоятельной программы, управляющей ресурсами системы [1]. Как правило, операционная система в таких случаях не является самостоятельным продуктом, а лишь частью монолитной микропрограммы устройства.

Более половины решений, работающих под управлением ОС — 51% (или 36,3% от общего числа систем) используют коммерческие ОС. В следующей по численности группе используются ОС собственной разработки — 21% или 15,1% от всех систем. Решения на базе условно «бесплатных» ОС с открытым кодом и на коммерчески распространяемых системах с открытым кодом занимают 16% и 11,8% соответственно (11,5% и 8,4% общего числа систем) [2].

Таким образом, если создать бесплатную ОС с открытым кодом, которая обеспечит такую же гибкость, что и ОС собственной разработки, то это удовлетворит преобладающую часть рынка.

Из-за накладываемых ограничений на подобные ОС встраиваемых систем их называют ОС реального времени.

ОС реального времени (далее – realtime OS, RTOS) – это многозадачные ОС, выполняющие задачи в реальном времени[3]. Основные подходы к реализации RTOS базируются либо на реакциях на события (event-driven RTOS), либо на распределении времени выполнения задач (переключении задач по прерыванию от таймера). Второй подход выполняет переключение задач чаще, чем требуется, но при этом задачи выполняются более “гладко” и создаётся впечатление, что задачи выполняются параллельно.

Существующие встраиваемые ОС можно разделить на две группы. Первые представляют собой UNIX-подобные ОС (QNX, eCos, VxWorks), вторые же являются самостоятельными продуктами и представляют собой специфичную реализацию (uC/OS, TNKernel, FreeRTOS). Они обладают полезными особенностями, например ОС uC/OS внедрила поддержку сопроцедур (co-routines) – программ с минимальными требованиями к ресурсам, работающих параллельно с основными задачами ОС.

Практически все встраиваемые ОС поддерживают многозадачность, но лишь небольшое их количество поддерживает возможность выбора алгоритма планировщика (подробнее об алгоритмах планировщика смотрите ниже). В большинство ОС включены примитивы синхронизации – семафоры, очереди сообщений

и т. д. Как правило, минимальный размер таких ОС – от 2 до 4 Кб.

В то же время в некоторых сферах задач существующие ОС обладают и недостатками – многие решения имеют высокую цену (цена лицензии ОС eCos оставляет около 12000$). Большинство ОС ориентированы на разработку только под конкретной гостевой ОС, что ограничивает разработчика (ОС ThreadX ориентирована на разработку только из-под ОС Windows), к тому же в состав многих ОС не входит загрузчик. Из-за этого многие ОС представляют собой только набор базовых функций (FreeRTOS, TNKernel). Сами по себе без пользовательских приложений такие ОС работать не могут. Относительно высокие требования развитых встраиваемых ОС не позволяют применять их в простых встраиваемых системах (uClinux нуждается в 512 Кб RAM [3,4], eCos – в 32 Кб, в то время как дешевые встраиваемые решения обладают примерно 64 Кб RAM, которая расходуется на прикладные задачи в большем количестве, чем на ОС).

Таким образом, рынок встраиваемых ОС представлен широким ассортиментом, но не для всех практических задач (в особенности – для малых встраиваемых систем) можно сделать однозначный выбор ОС.

2. Обзор архитектуры ARM

Хотя в дальнейшем планируется поддержка других архитектур, в данный момент разрабатываемая ОС ориентирована на семейство процессоров ARM. Такой выбор платформы вполне очевиден.

ARM Ltd. (от Advanced RISC Machines) – британская корпорация, являющаяся одним из крупнейших разработчиков и распространителей лицензий современной архитектуры 32-х разрядных RISC-процессоров, специально ориентированных для использования в портативных и мобильных устройствах (таких как мобильные телефоны, персональные органайзеры, пр.).

Технология ARM оказалась весьма успешной и в настоящее время является доминирующей микропроцессорной архитектурой для портативных цифровых устройств. ARM Ltd. утверждает, что общий объём микропроцессоров, произведённых по их лицензии, превышает 2,5 миллиарда штук. Семейство процессоров ARM используется примерно в 75% встраиваемых решений.

Архитектура ARM представляет значительный интерес для разработчиков встраиваемых систем. ARM оперирует 15 рабочими регистрами и работает в 7 режимах – пользовательских (USR/SYS), супервизорном (SVC), режимах прерываний (IRQ/FIQ) , режимах

исключений (ABT/UND).

Ассемблер процессоров ARM во многом схож с Intel 80286 и Motorola 68020, но имеет несколько характерных отличий: двухуровневая система прерываний с банкируемыми регистрами (IRQ, FIQ); мощные режимы адресации; операции сдвига в связке с арифметическими, не снижающие производительность, и пр. К тому же каждая инструкция содержит 4-битный префикс условия, т.е. каждая инструкция может быть условной. Дальнейшее развитие ветки процессоров ARM добавило дополнительные преимущества – ускоренное выполнение Java-инструкций, сокращённый набор инструкций Thumb и пр [5].

3. Разработка ОС

В основе тестовой платформы для разработки ОС использовался процессор AT91SAM7X256. Он обладает 256 Кб flash-памяти и 64 Кб RAM.

Отличительными характеристиками данного чипа является наличие USB и Ethernet, что может оказаться полезным в дальнейшей разработке. Чип содержит два последовательных порта ввода-вывода и порт отладки (синхронный последовательный), поддерживает интерфейс JTAG. ОС разрабатывалась таким образом, чтобы без значительных модификаций использоваться на любом чипе серии AT91SAM7. Планируется поддержка и других чипов.

Разработка ОС велась в среде ОС Linux, при этом использовались компиляторы семейства GCC (gnuarm 3.4.3). Используемый отладчик – GDB 6.4 + OpenOCD [3,6-8].

Исходные тексты ОС доступны для более детального ознакомления в дискуссионной группе http://groups.google.com/group/avos_devel. На использование и распространение исходных текстов ОС никаких ограничений не накладывается.

Проект ОС состоит из следующих частей:

- архитектурно-зависимая часть (arch) включает в себя загрузчик, а также реализацию базовых функций, необходимых для работы ОС и зависящих от ресурсов и архитектуры платформы (функции ввода/вывода, обработка прерываний, функции для работы с таймером и т.д.).
- ядро (kernel) - основные функции ядра системы.
- заголовочные файлы (include) – содержат заголовки основных функций ядра системы.

- библиотечные функции (lib) – с целью минимизации размера ОС стандартная библиотека С была исключена, из-за этого некоторые функции пришлось реализовать в самой ОС.
- пользовательские ресурсы (usr) – здесь должны храниться пользовательские приложения. На данный момент поддерживаются пользовательские приложения и драйвера. Подробнее о написании таких приложений читайте далее.
- вспомогательные утилиты (utils) содержат конфигурационные файлы для программ, использующихся при разработке – отладчика, программатора, терминала, On-Chip отладчика. Также здесь содержатся скрипты оболочки, которые используются для динамической генерации некоторых файлов.

3.1 Загрузчик

Разработка ОС начиналась с загрузчика. Детального описания он не заслуживает, так как является типичным для архитектуры ARM. Загрузчик написан на ассемблере с использованием препроцессора языка С. Микропрограмма начинается с обработки прерывания сброса (reset). Чип стартует в режиме SVC. В этом режиме устанавливается временный стек (от вершины адресного пространства) и вызывается функция lowlevel_init(), отвечающая за инициализацию аппаратных ресурсов. Затем загрузчик устанавливает стеки для каждого режима процессора. Все стеки, кроме пользовательского имеют фиксированный размер и занимают память от вершины адресного пространства. Далее инициализируются секции памяти (копируются из Flash-памяти в ОЗУ чипа) и происходит переход на функцию main(). С этого момента начинается работа ОС.

3.2 Архитектурно-зависимая часть

Некоторые функции являются архитектурно-зависимыми. Чтобы портировать ОС на другие платформы (или даже на другие чипы, отличные от используемого) необходимо будет изменить архитектурно-зависимую часть ОС.

Для функционирования ОС требуется реализация следующих функций:

- lowlevel_init() - должна выполнять действия по инициализации ресурсов системы (установка режимов портов ввода-вывода, установка обработчиков прерываний, если это необходимо и т. д.)
- sys_reset() - функция программной перезагрузки системы.

- serial_init(), serial_tx(unsigned char), serial_rx() - в асинхронном режиме эта функция вызывается из обработчика прерывания соответствующего порта. Полученный байт функция должна передать в функцию ядра serial_interrupt(unsigned char).
- timer_init(), timer_handler() – инициализация таймера и обработчик его прерываний. Последний должен вызывать do_timer(), отвечающий за многозадачность.
- Ram.ld – скрипт линковщика GNU ld.

В данной версии ОС, эти функции реализованы применительно к вышеупомянутому чипу AT91SAM7X256.

3.3 Контекст в архитектуре ARM. Переключение контекста

Чтобы перейти к многозадачности необходимо понимать, что является контекстом задачи. Обычно под контекстом подразумевают виртуальное адресное пространство, в котором выполняется программа, и состояние процессора в данный момент времени.

В данной ОС для архитектуры ARM контекстом считается состояние всех рабочих регистров, включая банкируемые, и текущее содержимое стека задачи. Кроме этого, нужно запоминать остальные данные о задаче в соответствующей структуре (приоритет, состояние, начальная вершина стека и т.д.) [9].

3.4 Алгоритмы многозадачности

Принято использовать следующие типы псевдопараллельной многозадачности:

Кооперативная многозадачность - вид многозадачности, при котором фоновые задачи выполняются только во время простоя основного процесса и только в том случае, если на это получено разрешение основного процесса. Этот вид удобно использовать для тесно связанного между собой небольшого количества задач. При кооперативной многозадачности время работы расходуется наиболее эффективно.

Вытесняющая многозадачность - вид многозадачности, при котором операционная система сама передает управление от одной выполняемой программы другой. Распределение процессорного времени осуществляется планировщиком процессов. Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя. В то же время, квота, выделяемая на каждую задачу, часто может оказаться избыточной.

Планировщик рассматриваемой ОС использует вытесняющую

многозадачность.

В ранних версиях ОС Linux использовался довольно простой, но эффективный алгоритм работы планировщика:

1. Каждой задаче задаётся значение приоритета. Например, есть три задачи с приоритетами $p_1=2$, $p_2=5$, $p_3=8$. Также вводится понятие суммарного приоритета $q_1=q_2=q_3=0$.
2. С каждым переключением задачи накапливается приоритет всех задач.
3. Задача с максимальным приоритетом становится активной, и её суммарный приоритет устанавливается равным нулю.

Рассмотрим несколько итераций такого алгоритма:

1. $q_1=0$ $q_2=0$ $q_3=0$ Нет активных задач, $p_1=2$, $p_2=5$, $p_3=8$
2. $q_1=2$ $q_2=5$ $q_3=8$ Активная задача 3, $q_3=0$
3. $q_1=4$ $q_2=10$ $q_3=8$ Активная задача 2, $q_2=0$
4. $q_1=6$ $q_2=5$ $q_3=16$ Активная задача 3, $q_3=0$
5. $q_1=8$ $q_2=10$ $q_3=8$ Активная задача 2, $q_2=0$
6. $q_1=10$ $q_2=5$ $q_3=16$ Активная задача 3, $q_3=0$
7. $q_1=12$ $q_2=10$ $q_3=8$ Активная задача 1, $q_1=0$
8. $q_1=2$ $q_2=15$ $q_3=16$ Активная задача 3, $q_3=0$

Как видно из рассмотренного примера, квота времени, выделенная на задачу не прямо пропорциональна приоритету, но сохраняется зависимость $q_1<q_2<q_3$ при $p_1<p_2<p_3$, где q_i – квота времени, выделенная на задачу i.

Этот алгоритм реализован и в разрабатываемой ОС в связи со своей простотой и наглядностью. Существует возможность переключения задачи с помощью системного вызова (реализована кооперативная многозадачность наряду с вытесняющей). В ближайшей перспективе планируется опциональное включение различных алгоритмов переключения задач и реализация примитивов синхронизации [9-11].

3.5 Системные вызовы

Некоторые функции ОС не должны прерываться – это может повлиять на их выполнение (например, если во время записи данных в порт задача будет прервана и порт будет захвачен конкурирующей задачей, то данные будут отправлены некорректно).

В ОС реализованы следующие системные вызовы:

- fork() – создание дочернего процесса.
- exit(int status) – завершение текущего процесса.
- sleep(int sec) – приостановка текущего процесса на указанное время.

- setpriority(int new_priority) – установка нового приоритета текущего процесса.
- reset() – программная перезагрузка системы.
- wait(pid_t pid) – ожидание, пока какой-либо процесс не получит сигнал.
- waitpid(pid_t pid, int *status, int options) – ожидание, пока указанный процесс не получит сигнал.
- execve(const char *filename, const char *argv[], const char *envp[]) – выполнение указанной программы (замещает текущий процесс).
- malloc(size_t len) – выделение блока памяти.
- free(void *ptr) – освобождение блока памяти.

Большая часть этих функций относится к управлению задачами, execve() обеспечивает интерфейс для запуска пользовательских программ, а работа последних двух функций касается динамической памяти.

3.6 Менеджер памяти

Стек задачи выделяется при её создании. В стеке хранятся все локальные переменные процедур, а это часто приводит к избыточному использованию стека и преждевременному его переполнению. Чтобы избежать этого, необходимо использовать динамическое выделение памяти. Поскольку встраиваемые устройства ограничены в объёме ОЗУ, то второй способ является более предпочтительным.

Менеджер памяти выполняет следующие действия: обнаруживает свободный участок памяти, помечает его как занятый, возвращает адрес его начала. При освобождении памяти помечает снова блок как незанятый.

В ОС Linux используется следующий принцип выделения памяти. В оперативной памяти резервируется место под блоки различного размера. При выделении памяти ОС находит минимальный блок, подходящий запрошенному размеру. На практике этот способ весьма эффективен, хотя некоторая часть пространства памяти остаётся незанятой из-за избыточности блоков памяти (каждый блок несколько превышает запрошенный размер).

В разрабатываемой ОС используется простейшее последовательное выделение памяти. Т.е. блоки изначально не выделены, а создаются при необходимости в первом найденном свободном участке памяти. При этом каждый блок указывает на начало следующего (работа алгоритма показана на рисунке).

Недостатком этого алгоритма является чрезмерная фрагментация памяти, поэтому использование динамической памяти в данной ОС оправдано только для больших блоков.

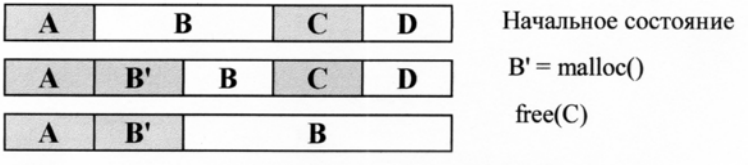

Рисунок – Пример работы менеджера памяти

3.7 Написание приложений для ОС

Разрабатываемая ОС поддерживает такие виды пользовательских задач как приложения и драйвера. В данный момент различие между ними заключается только в интерфейсах обращения к ним (у приложений только одна точка входа – main(), а у драйверов – функции загрузки, выгрузки, реакция на открытие/закрытие устройства и обращение к нему). В дальнейшем планируется выполнение драйверов на системном уровне, в то время как приложения выполняются на пользовательском.

Все программы и драйверы хранятся в папках usr/app и usr/drv соответственно. При сборке проекта происходит этап “bootstrap”. На этом этапе специальный скрипт анализирует, какие программы и драйвера должны быть включены в сборку, и изменяет соответствующим образом заголовочные файлы applist.h и drvlist.h. На содержимом этих файлов основан запуск пользовательских программ и работа с драйверами.

Поскольку ОС находится на стадии разработки, то технология написания приложений и драйверов для данной ОС не является достаточно удобной для конечного пользователя. В дальнейшем возможны существенные изменения в этой области.

Выводы

Подводя итоги проделанной работы, видим, что разработана основа операционной системы и реализованы базовые функции, достаточные для реализации простых встраиваемых систем.

Удалось запустить под управлением ОС простую текстовую оболочку, позволяющую выполнять приложения и проводить

элементарный мониторинг ресурсов системы. ОС обладает достаточной стабильностью. Минимальная сборка ОС занимает менее 3 Кб, что оставляет достаточно памяти для реализации прикладного программного обеспечения.

В то же время в ОС следует реализовать поддержку синхронизации (семафоров, мьютексов, критических секций, очередей сообщений, событий), файловой системы, максимально приблизить ОС к стандартам POSIX, портировать ОС на другие платформы. При этом одним из основных критериев должна оставаться гибкость системы (возможность легко включать/исключать её компоненты) и простота в использовании.

Перечень ссылок

1. Jim Turley. Operating systems on the rise // Embedded Systems Design – 2006, №6, p.21.
2. Таненбаум А.П. Операционные системы: Разработка и реализация.– СПб.: Питер, 2007 – 704 с.
3. Лав Р. Разработка ядра Linux – К.: Диалектика/Вильямс, 2006 – 448 с.
4. eCos vs. uClinux: Which is best for your embedded target? (Электронный ресурс). – Rob Wehrli. – Электрон. дан. – New York, 2002. – Режим доступа: http://www.linuxdevices.com/articles/AT3393503683.html свободный. – Загл. с экрана.
5. Программная платформа для телекоммуникаций: открытые системы (Электронный ресурс). – Александр Трофимов. – Электрон. дан. – М., 2008. – Режим доступа: http://www.osp.ru/os/2008/02/4924564 свободный. – Загл. с экрана.
6. Keith E. Curtis Embedded Multitasking – 2006. – NEWNES – 416 p.
7. Статьи по электронным компонентам: компоненты и технологии (Электронный ресурс). – Электрон. журнал. – М., 2008. – Режим доступа: http://compitech.ru/ свободный. – Загл. с экрана.
8. ARM System Developer's Guide: Designing and Optimizing System Software (The Morgan Kaufmann Series in Computer Architecture and Design) – 2004. – Morgan Kaufmann – 689 p.
9. Michael Barr, Anthony Massa. Programming Embedded Systems: With C and GNU Development Tools, 2nd Edition – 2006. – O'Reilly Media. – 301 p.
10. David Seal. ARM Architecture Reference Manual, Second Edition – 1996. – Addison-Wesley – 807 p.
11. L. Sha, R. Rajkumar, J. Lehoczky. Priority Inheritance Protocols: An

Approach to Real-Time Synchronization // IEEE Transactions on Computers, 1990, Vol.39, No.9. – p.61-64.