ЛЕКЦИИ ПО ДИСЦИПЛИНЕ «СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ»

ТЕМА 1. АППАРАТНО-ПРОГРАММНЫЕ СРЕДСТВА И КОМПЛЕКСЫ РЕАЛЬНОГО ВРЕМЕНИ

Лекция 1.1. Определение и основные особенности систем реального времени

1. Определение систем реального времени.
2. Требования, предъявляемые к системам реального времени.
3. Основные области применения систем реального времени.
4. Аппаратурная среда систем реального времени.

1. Определение систем реального времени

Существует несколько определений систем реального времени (СРВ) (real time operating systems (RTOS)), большинство из которых противоречит друг другу. Приведем некоторые из них, чтобы продемонстрировать различные взгляды на назначение и основные задачи СРВ:

1. Системой реального времени называется система, в которой успешность работы любой программы зависит не только от ее логической правильности, но и от времени, за которое она получила результат. Если временные ограничения не удовлетворены, то фиксируется сбой в работе систем.

Таким образом, временные ограничения должны быть гарантированно удовлетворены. Это требует от системы быть предсказуемой, то есть вне зависимости от своего текущего состояния и загруженности выдавать нужный результат за требуемое время. При этом желательно, чтобы система обеспечивала как можно больший процент использования имеющихся ресурсов.

Примером задачи, где требуется СРВ, является управление роботом, берущим деталь с ленты конвейера. Деталь движется, и робот имеет лишь небольшое временное окно, когда он может ее взять. Если он опоздает, то деталь уже не будет на нужном участке конвейера, и, следовательно, работа не будет сделана, несмотря на то, что робот находится в правильном месте. Если он позиционируется раньше, то деталь еще не успеет подъехать, и он заблокирует ей путь.

Другим примером может быть космический аппарат, находящийся на автопилоте. Сенсорные серводатчики должны постоянно передавать в управляющий компьютер результаты измерений. Если результат какого-либо измерения будет пропущен, то это может привести к недопустимому несоответствию между реальным состоянием систем космического аппарата и информацией о нем в управляющей программе.

Различают сильное (hard) и слабое (soft) требование реального времени. Если запаздывание программы приводит к полному нарушению работы управляемой системы, то говорят о сильном реальном времени (жесткие СРВ). Если же запаздывание ведет только к потере производительности, то говорят о слабом реальном времени (мягкие СРВ). Большинство программного обеспечения ориентировано на слабое реальное время, а задача хорошей

СРВ - обеспечить уровень безопасного функционирования системы, даже если управляющая программа никогда не закончит своей работы.

2. Стандарт POSIX 1003.1 определяет СРВ следующим образом: «Реальное время в операционных системах - это способность операционной системы обеспечить требуемый уровень сервиса в заданный промежуток времени».

3. Иногда системами реального времени называют системы постоянной готовности (on-line системы), или «интерактивные системы с достаточным временем реакции». Обычно это делают фирмы-производители по маркетинговым соображениям. Если интерактивную программу называют работающей в реальном времени, то это означает, что она успевает обрабатывать запросы от человека, для которого задержка в сотни миллисекунд даже незаметна.

4. Часто понятие «система реального времени» отождествляют с понятием «быстрая система». Это не всегда правильно. Время задержки реакции СРВ на событие не так уж важно (оно может достигать нескольких секунд). Главное, чтобы это время было достаточно для рассматриваемого приложения и гарантированно. Часто алгоритм с гарантированным временем работы менее эффективен, чем алгоритм, таким свойством не обладающий. Например, алгоритм «быстрой» сортировки (quicksort) в среднем работает значительно быстрее многих других алгоритмов сортировки, но его гарантированная оценка сложности значительно хуже.

5. Во многих важных сферах приложения СРВ вводятся свои понятия «реального времени». Так, процесс цифровой обработки сигнала называют идущим в «реальном времени», если анализ (при вводе) и/или генерация (при выводе) данных может быть проведен за то же время, что и анализ и/или генерация тех же данных без цифровой обработки сигнала.

Например, если при обработке аудио данных требуется 2,01 секунды для анализа 2,00 секунды звука, то это не процесс реального времени. Если же требуется 1,99 секунды, то это процесс реального времени. Исходя из выше сказанного, дадим определение системы реального времени в следующей интерпретации.

Определение. Система реального времени реагирует в предсказуемое время на непредсказуемое появление внешних событий.

Это определение предъявляет к системе вполне определенные **базовые требования.** Рассмотрим требования, предъявляемые к системам реального времени.

2. Требования, предъявляемые к системам реального времени Своевременная реакция. После того как произошло событие, реакция должна последовать не позднее, чем через требуемое время. Превышение этого времени рассматривается как серьезная ошибка.

Одновременная обработка информации, которая характеризует изменение процесса нескольких событий. Даже если одновременно происходит несколько событий, реакция ни на одно из них не должна запаздывать. Это означает, что система реального времени должна иметь встроенный

параллелизм. Параллелизм достигается использованием нескольких процессоров в системе и/или многозадачным подходом.

Рассмотрим основные признаки систем жесткого и мягкого реального времени.

Признаки систем жесткого реального времени:

. недопустимость никаких задержек, ни при каких условиях;

. бесполезность результатов при опоздании;

. катастрофа при задержке реакции;

. цена опоздания бесконечно велика.

Пример системы жесткого реального времени - бортовая система управления самолетом.

Признаки систем мягкого реального времени:

. за опоздание результатов приходится платить;

. снижение производительности системы, вызванное запаздыванием реакции на происходящие события.

Пример - автомат розничной торговли и подсистема сетевого интерфейса. В последнем случае можно восстановить пропущенный пакет, используя сетевой протокол, повторяющий передачу пропущенных пакетов. При этом, конечно, произойдет снижение производительности системы.

Таким образом, различие между системами жесткого и мягкого реального времени определяется следующими требованиями: **система называется системой жесткого реального времени, если она "не имеет права опаздывать", и мягкого реального времени - если ей "не следует опаздывать".**

Введем понятие операционной системы (ОС). Операционная система - это комплекс программ для управления и координации работы всех устройств системы, управления процессом выполнения прикладных программ и обеспечения диалога с пользователем.

Не существует операционных систем жесткого или мягкого реального времени. Понятия системы реального времени и операционной системы реального времени (ОСРВ) часто смешиваются.

Система реального времени - это конкретная система, связанная с реальным объектом. Она включает в себя необходимые аппаратные средства, операционную систему и прикладное программное обеспечение.

Операционная система реального времени – это только инструмент, помогающий построить конкретную систему реального времени. Поэтому бессмысленно говорить об операционных системах жесткого или мягкого реального времени. Можно говорить только о том, можно ли с помощью данной операционной системы построить систему реального времени. Конкретная ОСРВ может только предоставить возможность создать систему жесткого реального времени. Но обладание такой ОСРВ вовсе не делает систему "жесткой". Для создания системы жесткого реального времени необходимо сочетание подходящих аппаратных средств, адекватной операционной системы и правильного проектирования прикладного программного обеспечения.

Если, например, принято решение построить систему реального времени, обслуживающую TCP/IP-соединение через Ethernet, то система никогда не будет системой жесткого реального времени, поскольку сам Ethernet непредсказуем. В данном случае, основное ограничение на создание СРВ оказывает метод случайного доступа CSMA/CD.

Если, с другой стороны, вы создаете приложение над такой ОС, как "Windows 3.11", то ваша система никогда не будет системой жесткого реального времени, поскольку непредсказуемо поведение операционной системы.

Согласно определению, СРВ должна «обеспечить требуемый уровень сервиса в заданный промежуток времени». Этот промежуток времени обычно задается периодичностью и скоростью процессов, которыми управляет система. Приведем типичные времена реакции на внешние события в процессах, управляемых СРВ:

математическое моделирование - несколько микросекунд;
радиолокация - несколько миллисекунд;
складской учет - несколько секунд;
торговые операции - несколько минут;
управление производством - несколько минут;
химические реакции - несколько часов.

Видно, что времена сильно разнятся и накладывают различные требования на вычислительную установку, на которой работает СРВ. Различная предметная область использования СРВ, предъявляет к системам в каждом конкретном случае различные временные требования.

Интервал между поступлениями сообщений в ЭВМ может быть случайным и определяться внешними факторами, такими, как нажатие клавиши оператором, или он может быть циклическим и управляться от часов или от сканирующего механизма в ЭВМ. Так же, как и время ответа, этот интервал может изменяться от доли миллисекунд до получаса и более.

Определим операционную систему реального времени как операционную систему, с помощью которой можно построить систему жесткого реального времени. Обязательные требования к ОСРВ:

Требование 1: ОСРВ должна быть многонитиевой или многозадачной и поддерживать диспетчеризацию с вытеснением.

Поведение ОСРВ должно быть предсказуемым. Это не означает, что ОСРВ должна быть быстрой, но означает, что максимальный промежуток времени для выполнения любой операции должен быть известен заранее и должен быть согласован с требованиями приложения. Например, Windows 3.11 - даже на процессоре Pentium Pro с тактовой частотой 200 МГц - неприменима для построения систем реального времени, поскольку одно приложение может навсегда захватить управление и заблокировать все остальные приложения.

Первое требование состоит в том, чтобы такая ОС была многонитиевой или многозадачной и, кроме того, планировщик должен иметь возможность вытеснять любую нить (задачу) и передавать управление той нити (задаче), которая больше всего в этом нуждается. Для обеспечения вытеснения

на уровне прерываний структура обслуживания прерываний (в том числе и аппаратная архитектура) должна быть многоуровневой.

Требование 2: Должно существовать понятие приоритета нити (задачи). Как найти нить (задачу), которая нуждается в ресурсах больше всего? В идеальном случае ОСРВ предоставляет ресурсы той задаче или драйверу, у которых осталось меньше всего времени до истечения срока реакции на событие (назовем такую ОС – ОС управляющей критическими сроками). Однако для реализации этого механизма нужно уметь прогнозировать, сколько времени понадобится задаче для завершения своей работы и сколько времени понадобится другим задачам для того, чтобы они успели к своим критическим срокам. Подобная ОСРВ пока еще не создана из-за сложности реализации. Поэтому разработчики ОС используют другой метод: они вводят концепцию приоритетов для нитей (задач).

При построении конкретной системы реального времени разработчик должен выстроить приоритеты задач таким образом, чтобы каждая из них успела с реакцией к своему критическому сроку, то есть он должен трансформировать базовое требование реального времени "успеть с реакцией к нужному моменту" в комбинацию приоритетов и в сценарий их динамического изменения. Очевидно, что при этой трансформации возможны ошибки, приводящие к неправильной работе системы. Для решения этого вопроса используют различные теории, такие как, теорию монотонного планирования или различные методы и средства моделирования. Однако, эти методы оказываются не всегда эффективными. Как бы то ни было, во всех современных ОСРВ приходится использовать механизм приоритетов как один из инструментов предсказуемости поведения системы. На сегодяшний день не имеется другого решения, понятие приоритета потока для систем реального времени неизбежно.

Требование 3: ОС должна поддерживать предсказуемые механизмы синхронизации нитей (задач). Все нити (задачи) разделяют данные (ресурсы) и должны обмениваться между собой информацией, поэтому необходимы механизмы межзадачного (межнитиевого) взаимодействия.

Требование 4: Должен существовать механизм наследования приоритетов (система должна быть защищена от инверсии приоритетов). Под инверсией приоритетов будем понимать изменение их обычного порядка. На самом деле именно эти механизмы синхронизации и тот факт, что разные нити выполняются в одном и том же пространстве памяти, и определяют различие между нитями и процессами. Процессы не разделяют одно и то же пространство памяти.

Комбинации приоритетов нитей и разделение между ними ресурсов приводит к классической проблеме инверсии приоритетов. Для создания условия инверсии приоритетов должно быть задействовано как минимум три нити. Если нить с самым низким приоритетом заблокировала ресурс (который она делит с самой высокоприоритетной нитью), в то время как работает нить с промежуточным приоритетом, возникает следующий эффект: нить с наивысшим приоритетом ожидает освобождения ресурса; нить с промежу-

точным приоритетом вытесняет низкоприоритетную нить и работает, пока не завершится; управление получает низкоприоритетная нить, которая освобождает ресурс, и только после этого нить с высоким приоритетом может продолжить свою работу. В этом случае время, необходимое для завершения нити с наивысшим приоритетом, зависит от времени работы нити с более низким приоритетом – это и есть инверсия приоритетов. Очевидно, что в такой ситуации высокоприоритетная нить может "прозевать" критическое событие.

Чтобы избежать таких ситуаций, ОСРВ должна быть снабжена механизмом наследования приоритетов, то есть блокирующая нить должна наследовать приоритет нити, которую она блокирует (конечно, только, в том случае, если заблокированная нить имеет более высокий приоритет). Поведение ОС должно быть предсказуемо. Наследование означает, что блокирующий ресурс тред наследует приоритет треда, который он блокирует (это справедливо лишь в том случае, если блокируемый тред имеет более высокий приоритет).

Здесь есть еще одна проблема: количество возможных приоритетов очень мало. Большинство современных ОСРВ допускают использование как минимум 256 приоритетов. В чем суть проблемы? Ответ очевиден: чем больше приоритетов в распоряжении проектировщика, тем более предсказуемую систему можно создать. При оптимальном проектировании системы различным нитям присваиваются различные приоритеты.

Рассмотрим временные требования к операционным системам. Разработчик должен знать все времена выполнения системных вызовов и уметь предсказывать поведение системы в любых ситуациях. Поэтому производитель ОСРВ обязательно должен давать информацию о следующих временных характеристиках системы:

. задержке прерывания (interrupt latency) - то есть время от момента появления запроса на прерывание до начала его обработки;

. максимальном времени исполнения каждого системного вызова. Оно должно быть предсказуемым и не зависеть от количества объектов в системе;

. максимальном времени, на которое ОС и драйверы могут блокировать прерывания.

Разработчик также должен знать и учитывать следующее:

. уровни системных прерываний;

. уровни прерываний устройств, максимальное время, которое занимают программы обработки прерываний, и т.д.

Если все перечисленные выше времена известны, то имеются все предпосылки для создания системы жесткого реального времени. При этом требования к производительности разрабатываемой системы должны быть согласованы с характеристиками выбранной ОСРВ и аппаратуры.

Те места в программах, в которых происходит обращение к критическим ресурсам, называются критическими секциями. Решение этой проблемы заключается в организации такого доступа к критическому ресурсу, когда только одному процессу разрешается входить в критическую секцию.

Ресурсы, которые не допускают одновременного использования несколькими процессами, называются критическими. Если нескольким вычислительным ресурсам необходимо пользоваться критическим ресурсом в режиме разделения, им следует синхронизировать свои действия таким образом, чтобы ресурс всегда находился в распоряжении не более чем одного из процессов.

Любая система реального времени взаимодействует с внешним миром через аппаратуру компьютера. Внешние события преобразуются в прерывания и обрабатываются драйвером устройства.

Доступ к аппаратуре имеют только драйверы. Поскольку приложения реального времени часто работают со специфическими внешними устройствами, требующими и специфического управления, разработчик системы реального времени должен уметь разрабатывать драйверы устройств.

В ОСРВ разработчик в первую очередь узнает, на каких приоритетах работают драйверы других устройств. Здесь обычно существует свободное пространство для прерываний с приоритетами, которые выше приоритетов стандартных драйверов.

Требование 5: Политика управления памятью в ОСРВ. При проектировании системы реального времени необходимо рассмотреть и другой важный вопрос: как строится политика управления памятью в ОСРВ? От решения этой проблемы во многом зависит быстродействие проектируемой системы.

Требования, накладываемые на вычислительную установку реального времени, формулируются следующим образом:

1. В зависимости от сложности программы управления, требование «реального времени» накладывает различные условия на вычислительную мощность процессора для СРВ.

2. Внешние события становятся известны системе посредством прерываний (interrupt requests (IRQ)) (т.е. запросов на обслуживание со стороны внешних устройств). Поэтому часто для ОСРВ более важна не мощность процессора, а характеристики компьютера, связанные с подсистемой прерываний. Желательными являются:

- наличие как можно большего количества уровней прерываний (IRQ levels) (т.е. аппаратного или/и программного декодирования источника запроса);

- как можно меньшее время реакции на прерывание (т.е. как можно меньшее время между поступлением запроса на обслуживание и началом выполнения обслуживающей программы).

3. СРВ часто сама является инициатором периодических процессов, которыми управляет (например, движением космического аппарата или луча радара). Поэтому необходимо иметь в наличии один или несколько таймеров (аппаратных устройств, выдающих прерывание через заданные промежутки времени), которые могут работать в периодическом или ждущем режиме.

4. Ввиду того, что СРВ часто управляет ответственными промышленными процессами, данное обстоятельство выдвигает очень жесткие требования к надежности используемого оборудования.

В течение длительного времени основными потребителями СРВ были военная и космическая области. Сейчас ситуация кардинально изменилась и

СРВ можно встретить даже в товарах широкого потребления. Рассмотрим основные области применения СРВ.

3. Основные области применения систем реального времени Военная и космическая области:

- бортовое и встраиваемое оборудование;
- системы измерения и управления, радары;
- цифровые видеосистемы, симуляторы;
- ракеты, системы определения положения и привязки к местности. Промышленность:
- автоматические системы управления производством (АСУП), автоматические системы управления технологическим процессом (АСУТП);
- автомобилестроение: симуляторы, системы управления двигателем, автоматическое сцепление, системы антиблокировки колес и т.д.;
- энергетика: сбор информации, управление данными и оборудованием;
- телекоммуникации: коммуникационное оборудование, сетевые коммутаторы, телефонные станции и т.д.;
- банковское оборудование (например, во многих банкоматах работает СРВ QNX).

Товары широкого потребления:

- мобильные телефоны (например, в телефонах стандарта GSM работает СРВ pSOS);
- цифровые телевизионные декодеры;
- цифровое телевидение (мультимедиа, видеосерверы);
- компьютерное и офисное оборудование (принтеры, копиры), например, в факсах применяется СРВ VxWorks, в устройствах чтения компакт-дисков – СРВ VRTX32.

4. Аппаратурная среда систем реального времени

Систему реального времени можно разделить как бы на три слоя:

1. **Ядро** - содержит только строгий минимум, необходимый для работы системы: управление задачами, их синхронизация и взаимодействие, управление памятью и устройствами ввода/вывода; размер ядра очень ограничен: часто несколько килобайт.

2. **Система управления** - содержит ядро и ряд дополнительных сервисов, расширяющих его возможности: расширенное управление памятью, вводом/выводом, задачами, файлами и т.д., обеспечивает также взаимодействие системы и управляющего/управляемого оборудования.

3. **Система реального времени** - содержит систему управления и набор утилит: средства разработки (компиляторы, отладчики и т.д.), средства визуализации (взаимодействия человека и операционной системы).

Вычислительные установки, на которых применяются СРВ, можно условно разделить на три группы.

1. «Обычные» компьютеры. По логическому устройству совпадают с настольными системами. Аппаратное устройство несколько отличается. Для обеспечения минимального времени простоя в случае технической неполадки процессор, память и т.д. размещены на съемной плате, вставляемой в специальный разъем так называемой «пассивной» основной платы. В другие разъемы этой платы вставляются платы периферийных контроллеров и другое оборудование. Сам компьютер помещается в специальный корпус, обеспечивающий защиту от пыли и механических повреждений. В качестве мониторов часто используются жидкокристаллические дисплеи, иногда с сенсочувствительным покрытием.

По экономическим причинам среди процессоров этих компьютеров доминирует семейство Intel 80x86.

Подобные вычислительные системы обычно не используются для непосредственного управления промышленным оборудованием. Они, в основном, служат как терминалы для взаимодействия с промышленными компьютерами и встроенными контроллерами, для визуализации состояния оборудования и технологического процесса. На таких компьютерах в качестве операционных систем часто используются «обычные» операционные системы с дополнительными программными комплексами, адаптирующими их к требованиям «реального времени».

2. Промышленные компьютеры. Состоят из одной платы, на которой размещены: процессор, контроллер памяти, память 4-х видов:

- ПЗУ, постоянное запоминающее устройство (ROM, read-only memory), где обычно размещена сама операционная система реального времени; типичная емкость – 0,5-1 Мб;
- ОЗУ, оперативное запоминающее устройство (RAM, random access memory), куда загружается код и данные ОСРВ; обычно организована на базе динамической памяти (dynamic RAM, DRAM); типичная емкость – 16-128 Мб;
- статическое ОЗУ (static RAM, SRAM) (то же, что и ОЗУ, но питается от имеющейся на плате батарейки), где размещаются критически важные данные, которые не должны пропадать при выключении питания; типичная емкость - 2Мб; типичное время сохранения данных - 5 лет;
- флеш-память (flash RAM) (электрически программируемое ПЗУ), которая играет роль диска для ОСРВ; типичная емкость - 4Мб.

Контроллеры периферийных устройств: SCSI (Small Computer System Interface), Ethernet, COM портов, параллельного порта, несколько программируемых таймеров. На плате находится также контроллер и разъем шины, через которую компьютер управляет внешними устройствами. В качестве шины в подавляющем большинстве случаев используется шина VME, которую в последнее время стала теснить шина Compact PCI.

Несмотря на наличие контроллера SCSI, обычно ОСРВ работает без дисковых накопителей, поскольку они не удовлетворяют предъявляемым к

системам реального времени требованиям по надежности, устойчивости к вибрации, габаритам и времени готовности после включения питания.

Плата помещается в специальный корпус (крейт), в котором разведены разъемы шины и установлен блок питания. Корпус обеспечивает надлежащий температурный режим, защиту от пыли и механических повреждений. В этот же корпус вставляются платы аналого-цифровых и/или цифро-аналоговых преобразователей (АЦП и/или ЦАП), через которые осуществляется ввод/вывод управляющей информации, платы управления электромоторами. В тот же корпус могут вставляться другие такие же (или иные) промышленные компьютеры, образуя многопроцессорную систему.

Среди процессоров промышленных компьютеров доминируют процессоры семейств Power PC (Motorola IBM) и Motorola 68xxx (Motorola). Также присутствуют процессоры семейств SPARC (SUN), Intel 80x86 (Intel), ARM (ARM), Intel 80960x (Intel). При выборе процессора определяющими факторами являются получение требуемой производительности при наименьшей тактовой частоте, а, значит, и наименьшей рассеиваемой мощности, а также наименьшее время переключения задач и реакции на прерывания. Подчеркнем важность малой рассеиваемой мощности процессора с точки зрения получения высокой отказоустойчивости системы в целом, поскольку малый нагрев процессора позволяет обойтись без охлаждающего вентилятора, который является достаточно ненадежным механическим устройством.

Промышленные компьютеры используются для непосредственного управления промышленным или иным оборудованием. Они часто не имеют монитора и клавиатуры, и для взаимодействия с ними служат «обычные» компьютеры, соединенные с ними через последовательный порт (COM порт) или Ethernet.

3. Встраиваемые системы. Устанавливаются внутрь оборудования, которым они управляют. Для крупного оборудования (например, ракета или космический аппарат) могут по исполнению совпадать с промышленными компьютерами. Для оборудования поменьше (например, принтер) могут представлять собой процессор с сопутствующими элементами, размещенный на одной плате с другими электронными компонентами этого оборудования. Для миниатюрного оборудования (например, мобильный телефон) процессор с сопутствующими элементами может быть частью одной из больших интегральных схем этого оборудования.

В дальнейшем под компьютером для ОСРВ будем понимать промышленный компьютер. Отметим основные особенности ОСРВ, диктуемые необходимостью ее работы на промышленном компьютере.

Система часто должна работать на бездисковом компьютере и осуществлять начальную загрузку из ПЗУ. В силу этого:

- критически важным является размер системы;
- для экономии места в ПЗУ часть системы может храниться в сжатом виде и загружаться в ОЗУ по мере необходимости;
- система часто позволяет исполнять код как в ОЗУ, так и в ПЗУ;

- при наличии свободного места в ОЗУ система часто копирует себя из медленного ПЗУ в более быстрое ОЗУ;

- сама система компилируется, линкуется и превращается в загрузочный модуль на другом, «обычном» компьютере, связанном с промышленным компьютером через последовательный порт или Ethernet; это требует специального кроссплатформенного инструментария разработчика, поскольку типы процессоров и/или операционных систем на этих двух компьютерах не совпадают.

Система должна поддерживать как можно более широкий ряд процессоров, что дает возможность потребителю выбрать процессор подходящей мощности, а также поддерживать как можно более широкий ряд специального оборудования (периферийные контроллеры, таймеры и т.д.), которые могут стоять на плате компьютера и платах, которыми он управляет через общую шину.

Очевидно, что для получения законченной системы управления недостаточно промышленного компьютера, АЦП и/или ЦАП платы, крейта и ОСРВ. Нужно еще написать программу, которая будет непосредственно управлять конкретным промышленным оборудованием. Для этого необходим (кроссплатформенный) инструментарий разработчика, цена которого может превосходить цену перечисленных выше компонент, вместе взятых. Правда, этот инструментарий нужен только разработчику, а полученная программа может работать на многих компьютерах.

Критически важным параметром для СРВ является время ее реакции на прерывания (которое складывается из аппаратного времени задержки и программных задержек), а также предсказуемость этого времени.

Таким образом, понятие «системы реального времени» является новым и в полной мере не устоявшимся. Однако основной мыслью существующих определений является наложение жестких или мягких ограничений на время отклика системы при поступлении на нее внешнего воздействия. В связи с этим нельзя считать системами реального времени информационно-управляющие системы, не предусматривающие в алгоритмах своего функционирования возможность выдачи откликов в условиях лимита времени. Поэтому организация работы информационно-управляющих систем, функционирующих в режиме реального времени, существенно отличается от работы традиционных систем управления.

Лекция 1.2. Классы систем реального времени

1. Основные понятия систем реального времени.
2. Типы задач систем реального времени.
3. Классы систем реального времени.

1. Основные понятия систем реального времени

Рассмотрим суть основных понятий, которые используются в системах реального времени. Одним из основных понятий данной дисциплины является понятие процесса.

Процесс - это динамическая сущность программы, ее код в процессе своего выполнения.

В данном случае под программой понимается описание на некотором формализованном языке алгоритма, решающего поставленную задачу. Программа является статической единицей, то есть неизменяемой с точки зрения операционной системы, ее выполняющей.

Процесс имеет:

- собственные области памяти, где осуществляется хранение кода и данных;
- собственный стек;
- собственное отображение виртуальной памяти на физическую (в системах с виртуальной памятью);
- собственное состояние.

Процесс может находиться в одном из следующих типичных состояний (точное количество и свойства того или иного состояния зависят от операционной системы):

1) «остановлен» - процесс остановлен и не использует процессор; например, в таком состоянии процесс находится сразу после создания;

2) «терминирован» - процесс терминирован и не использует процессор; например, процесс закончился, но еще не удален операционной системой;

3) «ждет» - процесс ждет некоторого события (которым может быть аппаратное или программное прерывание, сигнал или другая форма межпроцессорного взаимодействия);

4) «готов» - процесс не остановлен, не терминирован, не ожидает, не удален, но и не работает; например, процесс может не получать доступа к процессору, если в данный момент выполняется другой, более приоритетный процесс;

5) «выполняется» - процесс выполняется и использует процессор; в ОСРВ это обычно означает, что этот процесс является самым приоритетным среди всех процессов, находящихся в состоянии «готов».

Понятие **вычислительного процесса** (или просто «процесса») было введено для реализации идей мультипрограммирования и мультизадачности. Как понятие процесс является определенным видом абстракции. Последовательный процесс (иногда называемый «задачей») – это выполнение отдельной программы с ее данными на последовательном процессоре.

В концепции, получившей широкое распространение в 70 – е годы, под **задачей** понимали совокупность связанных между собой и образующих единое целое программных модулей и данных, требующих ресурсов вычислительной системы для своей реализации. В последующие годы задачей стали называть единицу работы, для выполнения которой предоставляется центральный процессор. Вычислительный процесс может включать в себя несколько задач.

Концептуально процессор рассматривается в двух аспектах: во – первых, он является носителем данных и, во – вторых, он одновременно выполняет операции, связанные с их обработкой. Определение концепции процесса преследует цель выработать механизмы распределения и управления

ресурсами. Понятие **ресурса**, так же как и понятие процесса, является одним из основных при рассмотрении операционных систем реального времени. Термин ресурс обычно применяется по отношению к повторно используемым, относительно стабильным и часто недостающим объектам, которые запрашиваются, используются и освобождаются процессами в период их активности. Другими словами, ресурсом называется всякий объект, который может распределяться внутри системы. **Ресурс** - это объект, необходимый для работы процессу или задаче. Ресурсы могут быть разделяемыми, когда несколько процессов могут их использовать одновременно (в один и тот же момент времени) или параллельно (в течение некоторого интервала времени процессы используют ресурс попеременно), а могут быть и неделимыми (рис. 1).

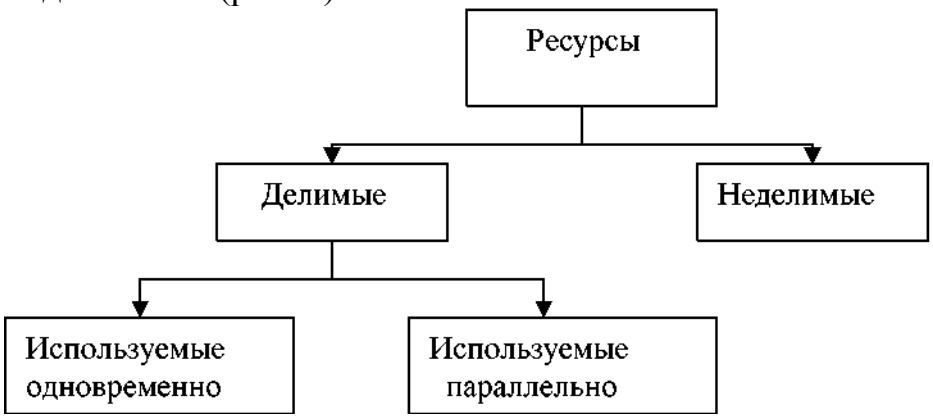


Рисунок 1. - Классификация ресурсов

При разработке первых систем ресурсами считались процессорное время, память, каналы ввода/вывода и периферийные устройства. Однако, с течением времени понятие ресурса стало гораздо более универсальным и общим. Различного рода программные и информационные ресурсы также могут быть определены для системы как объекты, которые могут разделяться и распределяться, и доступ к которым, необходимо соответствующим образом контролировать. В настоящее время понятие ресурса превратилось в абстрактную структуру с целым рядом атрибутов, характеризующих способы доступа к этой структуре и ее физическое представление в системе.

В первых вычислительных системах любая программа могла выполняться только после полного завершения предыдущей программы. Поскольку первые вычислительные системы были построены в соответствии с принципами Неймана, все подсистемы и устройства компьютера управлялись исключительно центральным процессором. Центральный процессор осуществлял и выполнение вычислений, и управление операциями ввода/вывода данных. Соответственно, пока осуществлялся обмен данными между оперативной памятью и внешними устройствами, процессор не мог выполнять вычисления. Введение в состав вычислительной машины специальных контроллеров позволило совместить во времени (распараллелить) операции вывода полученных данных и последующие вычисления на центральном процессоре. Однако, процессор продолжал часто и долго простаивать, дожидаясь завер-

шения очередной операции ввода/вывода. Поэтому было предложено организовать мультипрограммный (мультизадачный) режим работы вычислительной системы. Суть его заключается в том, что пока одна программа (один вычислительный процесс или задача) ожидает завершения очередной операции ввода/вывода, другая программа (а точнее, другая задача) может быть поставлена на решение.

Если в операционной системе могут одновременно существовать несколько процессов или/и задач, находящихся в состоянии «выполняется», то говорят, что это многозадачная система, а эти процессы называют параллельными.

Если процессор один, то в каждый момент времени на самом деле реально выполняется только один процесс или задача. Система разделяет время между такими «выполняющимися» процессами, давая каждому, квант времени, пропорциональный его приоритету. Этот квант времени часто не зависит от специфики решаемой задачи реального времени, поэтому такой подход обычно не используется в СРВ. Обычно в СРВ в состоянии выполнения может быть только один процесс. В «хорошей» СРВ это можно изменить программным путем.

Благодаря совмещению во времени выполнения двух программ общее время выполнения двух задач получается меньше, чем, если бы мы выполняли их по очереди (запуск только одной задачи после полного завершения другой). Но время выполнения каждой задачи в общем случае становится больше, чем, если бы мы выполняли каждую из них как единственную.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем, если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса).

Система поддерживает мультипрограммирование и старается эффективно использовать ресурсы путем организации к ним очередей запросов, составляемых тем или иным способом. Это требование достигается поддержанием в памяти более одного процесса, ожидающего процессор, и более одного процесса, готового использовать другие ресурсы, как только последние станут доступными. Общая схема выделения ресурсов такова. При необходимости использовать какой-либо ресурс (оперативную память, устройство ввода/вывода, массив данных и т.п.), задача обращается к супервизору операционной системы – ее центральному управляющему модулю, который может состоять из нескольких модулей, например: супервизор ввода/вывода, супервизор прерываний, супервизор программ, диспетчер задач и т.д. – посредством специальных вызовов (команд, директив) и сообщает о своем требовании. При этом указывается вид ресурса и, если надо, его объем. Директива обращения к операционной системе передает ей управление, переводя процессор в привилегированный режим работы, который обязательно существует в СРВ.

Ресурс может быть выделен задаче, обратившейся к супервизору с соответствующим запросом, если:

- он свободен и в системе нет запросов от задач более высокого приоритета к этому же ресурсу;

- текущий запрос и ранее выданные запросы допускают совместное использование ресурсов;

- ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемые ресурсы).

Получив запрос, система либо удовлетворяет его и возвращает управление задаче, выдавшей данный запрос, либо, если ресурс занят, ставит задачу в очередь к ресурсу, переводя ее в состояние ожидания (блокируя).

После окончания работы с ресурсом задача опять с помощью специального вызова супервизора сообщает операционной системе об отказе от ресурса, или операционная система забирает ресурс сама, если управление возвращается супервизору после выполнения какой-либо системной функции. Супервизор операционной системы, получив управление по этому обращению, освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу. Если очередь есть – в зависимости от принятой дисциплины обслуживания и приоритетов заявок он выводит из состояния ожидания задачу, ждущую ресурс, и переводит ее в состояние готовности к выполнению. После этого управление либо передается данной задаче, либо возвращается той, которая только что освободила ресурс.

В общем случае при организации управления ресурсами в СРВ всегда требуется принять решение о том, что в данной ситуации выгоднее: быстро обслуживать отдельные наиболее важные запросы, предоставлять всем процессам равные возможности, либо обслуживать максимально возможное количество процессов и наиболее полно использовать ресурсы.

Стек (stack) - это область памяти, в которой размещаются локальные переменные, аргументы и возвращаемые значения функций. Вместе с областью статических данных полностью задает текущее состояние процесса.

Виртуальная память - это «память», в адресном пространстве которой работает процесс. Виртуальная память:

1) позволяет увеличить объем памяти, доступной процессам за счет дисковой памяти;

2) обеспечивает выделение каждому из процессов виртуально непрерывного блока памяти, начинающегося (виртуально) с одного и того же адреса;

3) обеспечивает изоляцию одного процесса от другого.

Трансляцией виртуального адреса в физический адрес занимается операционная система. Для ускорения этого процесса многие компьютерные системы имеют поддержку со стороны аппаратуры, которая может быть либо прямо в процессоре, либо в специальном устройстве управления памятью. Среди механизмов трансляции виртуального адреса преобладает страничный, при котором виртуальная и физическая память разбиваются на части равного размера, называемые страницами (типичный размер - 4Кб), между

страницами виртуальной и физической памяти устанавливается взаимно однозначное (для каждого процесса) отображение. Отметим, что ОСРВ стремятся получить максимальную производительность на имеющемся оборудовании, поэтому некоторые ОСРВ не используют механизм виртуальной памяти из-за задержек, вносимых при трансляции адреса.

Межпроцессное взаимодействие - это тот или иной способ передачи информации из одного процесса в другой. Наиболее распространенными формами взаимодействия процессов являются (не все системы поддерживают перечисленные ниже возможности):

1) разделяемая память - два (или более) процесса имеют доступ к одному и тому же блоку памяти. В системах с виртуальной памятью организация такого вида взаимодействия требует поддержки со стороны операционной системы, поскольку необходимо отобразить соответствующие блоки виртуальной памяти процессов на один и тот же блок физической памяти); семафоры - два (или более) процесса имеют доступ к одной переменной, принимающей значение 0 или 1. Сама переменная часто находится в области данных операционной системы и доступ к ней организуется посредством специальных функций;

3) сигналы - это сообщения, доставляемые посредством операционной системы процессу. Процесс должен зарегистрировать обработчик этого сообщения у операционной системы, чтобы получить возможность реагировать на него. Часто операционная система извещает процесс сигналом о наступлении какого-либо сбоя, например, делении на 0, или о каком-либо аппаратном прерывании, например, прерывании таймера;

4) почтовые ящики - это очередь сообщений (обычно тех или иных структур данных), которые помещаются в почтовый ящик процессами и/или операционной системой. Несколько процессов могут ждать поступления сообщения в почтовый ящик и активизироваться после его поступления. Требует поддержки со стороны операционной системы.

Событие - это оповещение процесса со стороны операционной системы о той или иной форме межпроцессного взаимодействия, например, о принятии семафором нужного значения, о наличии сигнала, о поступлении сообщения в почтовый ящик.

Создание, обеспечение взаимодействия, разделение процессорного времени требует от операционной системы значительных вычислительных затрат, особенно в системах с виртуальной памятью. Это связано, прежде всего, с тем, что каждый процесс имеет свое отображение виртуальной памяти на физическую, которое надо менять при переключении процессов и при обеспечении их доступа к объектам взаимодействия (общей памяти, семафорам, почтовым ящикам). Часто бывает так, что требуется запустить несколько копий одной и той же программы, например, для управления несколькими единицами одного и того же оборудования. В этом случае мы несем двойные накладные расходы: держим в оперативной памяти несколько копий кода одной программы и тратим дополнительное время на обеспечение их взаимодействия.

Задача (или поток, или нить, thread) - это как бы одна из ветвей исполнения процесса:

- разделяет с процессом область памяти под код и данные;
- имеет собственный стек;
- разделяет с процессом отображение виртуальной памяти на физическую (в системах с виртуальной памятью);
- имеет собственное состояние.

Таким образом, у двух задач в одном процессе вся память является разделяемой и дополнительные расходы, связанные с разным отображением виртуальной памяти на физическую, сведены к нулю. Для задач так же, как для процессов, определяются понятия состояния задачи и межзадачного взаимодействия. Отметим, что для двух процессов обычно требуется организовать что-то общее (память, канал и т.д.) для их взаимодействия, в то время как для двух потоков часто требуется организовать что-то общее (например, область памяти), имеющее свое значение в каждом из них.

Приоритет - это число, приписанное операционной системой каждому процессу и задаче. Чем больше это число, тем важнее этот процесс или задача и тем больше процессорного времени он или она получит. При неправильном планировании приоритетов в ОСРВ, задача с меньшим приоритетом может вообще не получить управления при наличии в состоянии готовности задачи с большим приоритетом.

Связывание (линковка, linkage) - это процесс превращения скомпилированного кода (объектных модулей) в загрузочный модуль (то есть то, что может исполняться процессором при поддержке операционной системы). Различают:

- статическое связывание, когда код необходимых для работы программы библиотечных функций физически добавляется к коду объектных модулей для получения загрузочного модуля;
- динамическое связывание, когда в результирующем загрузочном модуле проставляются лишь ссылки на код необходимых библиотечных функций; сам код будет реально добавлен к загрузочному модулю только при его исполнении.

При статическом связывании загрузочные модули получаются очень большого размера. Поэтому подавляющее большинство современных операционных систем использует динамическое связывание, несмотря на то, что при этом начальная загрузка процесса на исполнение медленнее, чем при статическом связывании из-за необходимости поиска и загрузки кода нужных библиотечных функций (часто только тех из них, которые не были загружены для других процессов). При этом для избежания недетерминированной задержки на загрузку программы на исполнение все необходимые процессы реального времени запускают при старте системы (заранее, а не по требованию).

2. Типы задач систем реального времени Всякий процесс содержит одну или несколько задач. Операционная система позволяет задаче

порождать новые задачи. Задачи, по своей манере действовать, можно разделить на 3 категории:

1. Циклические задачи. Характерны для процессов управления и интерактивных процессов.

2. Периодические задачи. Характерны для многих технологических процессов и задач синхронизации.

3. Импульсные задачи. Характерны для задач сигнализации и асинхронных технологических процессов.

Чтобы система могла управлять задачами, она должна располагать всей необходимой для этого информацией. С этой целью на каждую задачу (процесс) заводится специальная информационная структура, называемая дескриптором процесса (описателем задачи). В общем случае дескриптор процесса содержит следующую информацию:

- идентификатор процесса (так называемый PID - process identificator);
- тип (или класс) процесса, который определяет для супервизора некоторые правила предоставления ресурсов. Управление вводом/выводом осуществляется операционной системой, компонентом, который называют супервизором ввода/вывода;
- приоритет процесса, в соответствии с которым супервизор предоставляет ресурсы. В рамках одного класса процессов в первую очередь обслуживаются более приоритетные процессы;
- переменную состояния, которая определяет, в каком состоянии находится процесс (готов к работе, в состоянии выполнения, ожидание устройства ввода/вывода и т.д.);
- защищенную область памяти (или адрес такой зоны), в которой хранятся текущие значения регистров процесса, если процесс прерывается, не закончив работы. Эта информация называется контекстом задачи;
- информацию о ресурсах, которыми процесс владеет и/или имеет право пользоваться (указатели на открытые файлы, информация о незавершенных операциях ввода/вывода и т.д.);
- место (или его адрес) для организации общения с другими процессами;
- параметры времени запуска (момент времени, когда процесс должен активизироваться, и периодичность этой процедуры);
- в случае отсутствия системы управления файлами – адрес задачи на диске в ее исходном состоянии и адрес на диске, куда информация выгружается из оперативной памяти, если ее вытесняет другая задача.

Когда говорят о процессах, то тем самым хотят отметить, что система поддерживает их обособленность: у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы – файлы, окна, семафоры и т.д. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы вычислительной системы, конкурируют друг с другом. В общем случае процессы (задачи) никак не связаны между собой и могут принадлежать даже разным пользователям, разделяющим одну вычислительную сис-

тему. Другими словами, в случае процессов система считает их совершенно не связанными и не зависимыми. При этом именно система берет на себя роль арбитра в конкуренции между процессами по поводу ресурсов.

Желательно иметь возможность задействовать внутренний параллелизм, который может быть в самих процессах. Такой внутренний параллелизм встречается достаточно часто и его использование позволяет ускорить их решение. В однопроцессорной системе задачи разделяют между собой процессорное время так же, как это делают обычные процессы, а в мультипроцессорной системе могут выполняться одновременно, если не встречают конкуренции из-за обращения к иным ресурсам.

Таким образом, главное, что обеспечивает многопоточность, - это возможность параллельно выполнять несколько видов операций в одной прикладной программе. Параллельные вычисления (а, следовательно, и более эффективное использование ресурсов центрального процессора, и меньшее суммарное время выполнения задач) теперь уже часто реализуются на уровне задач, и программа, оформленная в виде нескольких задач (потоков, нитей) в рамках одного процесса, может быть выполнена быстрее за счет параллельного выполнения ее отдельных частей.

3. Классы систем реального времени Количество операционных систем реального времени, несмотря на их специфику, очень велико. Сама специфика применения операционных систем реального времени требует гарантий надежности, причем гарантий, в том числе и юридических - этим, видимо, можно объяснить тот факт, что среди некоммерческих систем реального времени нет сколько-нибудь популярных.

Среди коммерческих систем реального времени можно выделить группу ведущих систем - по объемам продаж и по популярности. Это системы: VxWorks, OS-9, pSOS, LynxOS, QNX, VRTX. Различают следующие классы СРВ:

- исполнительные системы реального времени;
- ядра реального времени;
- UNIX'ы реального времени.

Исполнительные системы реального времени. Признаки систем этого типа - различные платформы для систем разработки и исполнения. Приложение реального времени разрабатывается на host- компьютере (компьютере системы разработки), затем компонуется с ядром и загружается в целевую систему для исполнения. Как правило, приложение реального времени -это одна задача и параллелизм здесь достигается с помощью нитей (threads).

Системы этого типа обладают рядом достоинств, среди которых основным достоинством является высокая скорость и реактивность системы.

Главная причина высокой реактивности систем этого типа - наличие только нитей (потоков) и, следовательно, малое время переключения контекста между ними (в отличие от процессов). С этим главным достоинством связан и ряд недостатков:

зависание всей системы при зависании нити;

проблемы с динамической загрузкой новых приложений.

Кроме того, системы разработки для продуктов этого класса традиционно дороги (порядка $20000). Однако, необходимо отметить, что качество и функциональность систем разработки в этом классе традиционно являются хорошими.

Наиболее ярким представителем систем этого класса является операционная система VxWorks. Область применения - компактные системы реального времени с хорошими временами реакций.

Ядра реального времени. В этот класс входят системы с монолитным ядром, где и содержится реализация всех механизмов реального времени этих операционных систем. Исторически системы этого типа были хорошо спроектированы. В отличие от систем других классов, разработчики систем этого класса имели время для разработки систем именно реального времени и не были изначально ограничены в выборе средств (например фирма "Microware" имела в своем распоряжении три года для разработки первого варианта OS-9). Системы этого класса, как правило, модульны, хорошо структурированы, имеют наиболее развитый набор специфических механизмов реального времени, компактны и предсказуемы. Наиболее популярные системы этого класса: OS-9, QNX.

Одна из особенностей систем этого класса - высокая степень масштабируемости. На базе этих ОС можно построить как компактные системы реального времени, так и большие системы серверного класса.

UNIX'ы реального времени. Исторически системы реального времени создавались в эпоху расцвета и бума UNIX'а и поэтому многие из них содержат те или иные заимствования из этой красивой концепции операционный системы (пользовательский интерфейс, концепция процессов).

Часть разработчиков операционных систем реального времени попыталась просто переписать ядро UNIX, сохранив при этом интерфейс пользовательских процессов с системой, насколько это было возможно. Реализация этой идеи не была слишком сложной, поскольку не было препятствия в доступе к исходным текстам ядра, а результат оказался замечательным. Получили и реальное время, и весь набор пользовательских приложений - компиляторы, пакеты, различные инструментальные системы.

В этом смысле создателям систем первых двух классов пришлось потрудиться не только при создании ядра реального времени, но и продвинутых систем разработки.

Однако Unix'ы реального времени имеют следующие недостатки: системы реального времени получаются достаточно большими и реактивность их ниже, чем реактивность систем первых двух классов.

Наиболее популярным представителем систем этого класса является операционная система реального времени Lynx OS.

Расширения реального времени для Windows NT. После появления Windows NT, сразу несколько фирм объявили о создании расширений реального времени для Windows NT. Этот означает, что подобные продукты были востребованы, что и подтверждает динамика их рыночного развития. Появ-

ление в свое время UNIX'ов реального времени означало ни что иное, как попытку применить господствующую программную технологию для создания приложений реального времени. Появление расширений реального времени для Windows NT имеет те же корни, ту же мотивацию. Огромный набор прикладных программ под Windows, мощный программный интерфейс

WIN32, большое количество специалистов, знающих эту систему. Соблазнительно было получить в системе реального времени все эти возможности.

Несмотря на то, что Windows NT создавалась как сетевая операционная система, и сочетание слов "Windows NT" и "реальное время" многими воспринимается как нонсенс, в нее при создании были заложены элементы реального времени. Она имеет - двухуровневую систему обработки прерываний (ISR и DPC), классы реального времени (процессы с приоритетами 16-32 планируются в соответствии с правилами реального времени). Причина появления этих элементов кроется в том, что у разработчиков Windows NT за плечами был опыт создания классической для своего времени операционной системы реального времени RSX11M (для компьютеров фирмы DEC).

Анализ возможностей Windows NT показывает, что эта система не годится для построения систем жесткого реального времени (система непредсказуема - время выполнения системных вызовов и время реакции на прерывания сильно зависит от загрузки системы; система велика; нет механизмов защиты от зависаний). Поэтому даже в системах мягкого реального времени Windows NT может быть использована, только при выполнении целого ряда рекомендаций и ограничений.

Разработчики расширений пошли двумя путями. Они использовали ядра классических операционных систем реального времени в качестве дополнения к ядру Windows NT. Таковы решения фирм "LP Eleknroniks" и "Radisys". В первом случае параллельно с Windows NT (на одном компьютере) работает операционная система VxWorks, во-втором случае - InTime. Кроме того, предоставляется набор функций для связи приложений реального времени и приложений Windows NT. Вот как, например, это выглядит у LP Elektroniks: вначале стандартным образом загружается Windows NT, затем с помощью специального загрузчика загружается операционная система VxWorks, распределяя под себя необходимую память Windows (что в дальнейшем позволяет избежать конфликтов памяти между двумя ОС). После этого полной "хозяйкой" на компьютере уже становится VxWorks, отдавая процессор ядру Windows NT только в случаях, когда в нем нет надобности для приложений VxWorks. В качестве канала для синхронизации и обмена данными между Windows NT и VxWorks служат псевдодрайверы TCP/IP в обеих системах. Технология использования двух систем на одном компьютере следующая - работу с объектом выполняет приложение реального времени, передавая затем результаты приложениям Windows NT для обработки, передачи в сеть, архивирования.

Второй вариант расширений реального времени фирмы VenturCom выглядит иначе: здесь сделана попытка "интегрировать" реальное время в

Windows NT путем исследования причин задержек и зависаний и устранения этих причин с помощью подсистемы реального времени. Решения фирмы "VenturCom" (RTX 4.2) базируются на модификациях уровня аппаратных абстракций Windows NT (HAL - Hardware Abstraction Layer) - программного слоя, через который драйверы взаимодействуют с аппаратурой. Модифицированный HAL и дополнтельные функции (RTAPI) отвечают также за стабильность и надежность системы, обеспечивая отслеживание краха Windows NT, зависания приложений или блокировку прерываний. В состав RTX входит также подсистема реального времени RTSS, с помощью которой Windows NT расширяется дополнительным набором объектов (аналогичным стандартным, но с атрибутами реального времени). Среди новых объектов - нити (потоки, процессы) реального времени, которые управляются специальным планировщиком реального времени (256 фиксированных приоритетов, алгоритм - приоритетный с вытеснением). Побочным результатом RTX является возможность простого создания программ управления устройствами, так как среди функций RTAPI есть и функции работы с портами ввода-вывода и физической памятью. Решения VenturCom характерны еще и тем, что они предоставляют для NT возможность конфигурирования Windows NT и создания встроенных конфигураций (без дисков, клавиатуры и монитора, интегратор компонентов - CI).

Несмотря на всю неоднозначность отношения традиционных пользователей систем реального времени ко всему, что связано с "Microsoft", необходимо констатировать факт: появился новый класс операционных систем реального времени - а именно расширения реального времени для Windows NT. Результаты независимых тестирований этих продуктов показывают, что они могут быть в перспективе использованы для построения систем жесткого реального времени после соответствующей доработки. Область применения расширений реального времени - большие системы реального времени, где требуется визуализация, работа с базами данных, доступ в Интернет и пр.

ТЕМА 2. УСТРОЙСТВА СВЯЗИ С ОБЪЕКТОМ

Лекция 2.1. Методы и средства обработки асинхронных событий

1. Обобщенная функциональная структура информационного тракта СРВ и устройства связи с объектом.
2. Средства обработки асинхронных событий.
3. Принципы функционирования интерфейса.
4. Программное обеспечение интерфейса.
5. Аппаратные средства интерфейса.

1. Обобщенная функциональная структура информационного тракта СРВ и устройства связи с объектом

Из всего состава функциональных устройств СРВ, образующих информационный тракт системы, рассмотрим только те, которые осуществляют функции сбора, предварительной обработки, представления, передачи и обработки информации. Блок-схема обобщенной

функциональной структуры информационного тракта и устройства связи с объектом представлены на рис. 1.

На вход системы поступает в общем случае аналоговый сигнал S(t), сформированный информационным устройством (или датчиком), являющимся источником данных. Сигнал S(t) рассматривается как реализация случайного процесса. Цепь преобразования данных одного устройства (или датчика) в многоканальной системе образует измерительный канал.

В блоке подготовки сигнал подвергается предварительной аналоговой обработке – согласованию, усилению (приведение амплитуды к динамическому диапазону устройством выборки и хранения – УВХ), полосовой фильтрации (ограничение полосы частот сигналов для корректной оцифровки).

Поскольку подсистема обработки является цифровой системой, то каждый сигнал подвергается процедуре аналого-цифрового преобразования в модуле АЦП. Последовательность отсчетов от различных измерительных каналов объединяется в общий поток для последующего ввода в компьютер или передачи по каналу связи. В ряде случаев могут применяться устройства сжатия данных (либо сжатие осуществляется после ввода данных в компьютер – программные методы сжатия). Состав и последовательность расположения функциональных устройств в различных СРВ может отличаться от приведенной в блок-схеме. Но, характерным является наличие данных устройств, как типовых в системах различного назначения и технического воплощения.

Подсистема передачи включает кодер и декодер канала связи, передающее и приемное устройства и собственно канал связи (среда с антенными устройствами). Кодер и декодер осуществляют помехоустойчивое кодирование и декодирование сигналов с целью дополни дополнительной защиты передаваемых сообщений от помех в канале связи и могут отсутствовать при наличии качественного канала.

Восстановление исходного аналогового сообщения по цифровым отсчетам с допустимой погрешностью производится на приемной стороне. В современных системах восстановление непрерывного сообщения, как правило, не выполняется, поскольку регистрация, хранение и обработка информации выполняются в цифровом виде, но принципиальная возможность восстановления предусматривается.

Одна из задач подсистемы цифровой обработки, которая выполняется с использованием ресурсов компьютера и специализированных процессоров цифровой обработки – сортировка информации и отбраковка аномальных результатов наблюдений. Отбраковка является частным случаем более общей задачи – фильтрации сигналов от помех или использования методов распознавания образов. Другими задачами подсистемы обработки являются:

предварительная обработка данных (сглаживание, удаление тренда);

статистическая обработка сигналов (применяются различные алгоритмы в зависимости от назначения СРВ);

спектральная обработка;

формирование моделей процессов и явлений;

представление результатов предварительной обработки или анализа;

хранение данных.

Исходная информация для последующего анализа исследуемого явления (или объекта) формируется с помощью средств проведения эксперимента, представляющих собой совокупность средств измерений различных типов (измерительных устройств, преобразователей, датчиков и принадлежностей к ним), каналов передачи информации и вспомогательных устройств для обеспечения условий проведения эксперимента. В различных предметных областях совокупность средств для проведения эксперимента может называться по-разному (например, экспериментальная установка, информационно-измерительная система, измерительная система). В дальнейшем будем пользоваться термином "измерительная система" (ИС). В зависимости от целей эксперимента иногда различают измерительные информационные (исследование), измерительные контролирующие (контроль, испытание) и измерительные управляющие (управление, оптимизация) системы, которые различаются в общем случае как составом оборудования, так и сложностью обработки экспериментальных данных.

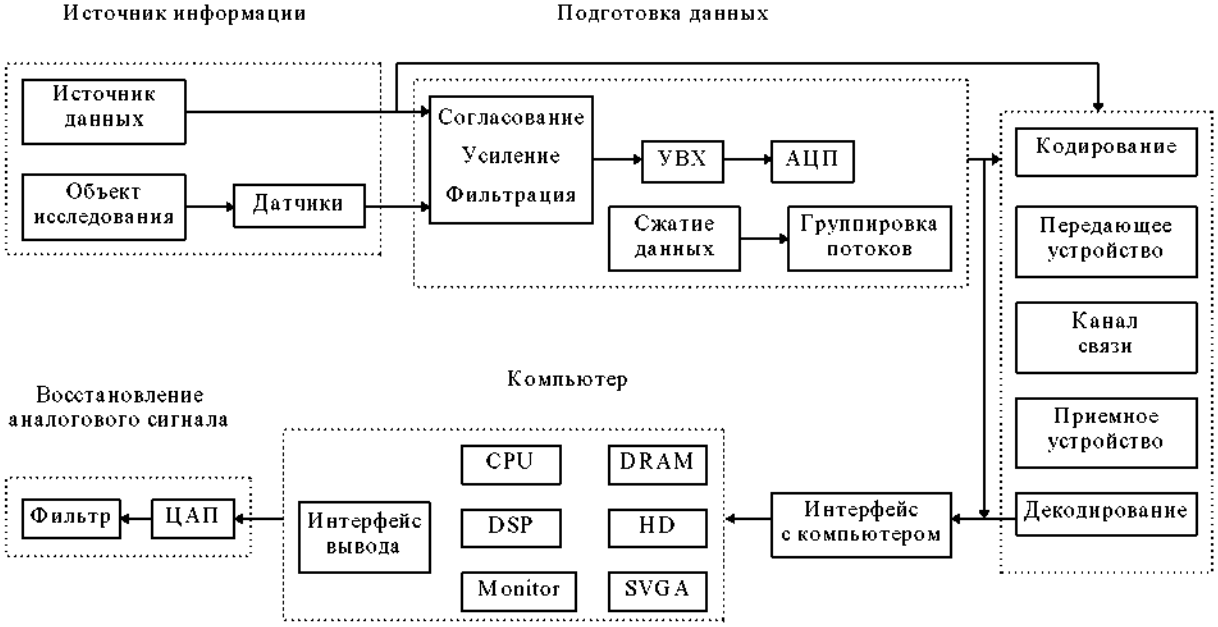


Рисунок 1. - Обобщенная блок-схема функциональной структуры информационного тракта и устройство связи с объектом

2. Средства обработки асинхронных событий

Состав средств измерений, входящих в измерительную систему и выполняющих функции датчиков сигналов, формирователей воздействий на исследуемый объект, в существенной степени определяется задачами эксперимента, которые ставятся при его планировании. То же самое можно сказать и о предварительном выборе методов обработки экспериментальных данных, которые могут в дальнейшем уточняться по мере получения экспериментальной информации об объекте исследования и условиях проведения эксперимента.

В связи с возрастанием сложности экспериментальных исследований (это проявляется в увеличении числа измеряемых величин, большом количестве информационных каналов, повышении требований к качеству регистрируемой информации и оперативности ее получения) в состав современных измерительных систем включаются вычислительные средства различных классов. Эти средства (мини-ЭВМ, персональные компьютеры, специализированные вычислители и контроллеры) не только выполняют функции сбора и обработки экспериментальной информации, но и решают задачи управления ходом эксперимента, автоматизации функционирования измерительной системы, хранения измерительных данных и результатов анализа, графической поддержки режимов контроля, представления и анализа.

Таким образом, современные средства проведения эксперимента представляют собой измерительно-вычислительные системы или комплексы, снабженные развитыми вычислительными средствами (в последнее время все чаще многопроцессорные). При обосновании структуры и состава ИС необходимо решить следующие основные задачи:

определить состав измерительного оборудования (датчики, устройства согласования, усиления, фильтрации, калибровки);

выбрать тип и характеристики компьютера, входящего в состав ИС (сейчас, как правило, персональный компьютер);

выбрать тип оборудования, выполняющего сбор данных и цифровую обработку сигналов;

адаптировать каналы связи между компьютером, оборудованием сбора данных (интерфейс), измерительными устройствами и потребителем информации;

разработать программное обеспечение ИС.

При выборе компьютера необходимо учитывать требования по оперативности получения результатов экспериментов, сложность алгоритмов обработки экспериментальных данных и объем получаемой информации. Это позволит оценить требуемую производительность процессора, емкость и характеристики ОЗУ и жестких дисков, характеристики видеосистемы.

Известно два подхода к обеспечению ввода аналоговых измерительных сигналов для последующей обработки с использованием цифровых методов. Первый подход основан на применении специализированных комплексных систем, в состав которых входит аппаратура аналого-цифрового преобразования, микропроцессорные средства цифровой обработки и устройства отображения информации. Второй подход основан на применении интерфейсных устройств сбора данных и универсальных компьютерных систем.

Примером специализированной системы является многоканальный анализатор сигналов SA 3550 фирмы Brüel & Kjær. Данный прибор выполняет следующие функции:

анализ сигналов и систем (механических, электрических, электромеханических);

структурные и модальные испытания с несколькими входами и выходами с помощью случайных сигналов и испытания с учетом собственных мод колебаний;

отыскание неисправностей механических систем и их компонент с возможностью изменения форм операционных деформаций;

анализ сервомеханизмов и сервосистем;

анализ и испытания в программах контроля качества;

анализ акустических и электроакустических систем;

измерения и анализ интенсивности звука;

исследования в целях борьбы с шумом.

Другим примером специализированной системы является многоканальный спектральный анализатор SI 1220 фирмы Schlumberger Technologies. Данный прибор позволяет выполнять многоканальный мониторинг конструкций, исследование резонансных явлений, структурный анализ, тестирование и балансировку машинного оборудования, частотный анализ сигналов и нелинейных цепей, исследование речи.

К недостаткам такого подхода построения измерительных систем можно отнести: ограничения на количество входных сигналов и их характеристики; жесткая структура алгоритмов обработки, не допускающая разработку программ анализа под конкретную задачу; ограниченные возможности графического представления результатов; высокая стоимость измерительных систем.

Второй подход основан на применении дополнительных интерфейсных модулей и цифровых процессоров сигналов в составе персонального компьютера. Существенными преимуществами второго подхода являются: гибкость измерительной системы при реализации различных алгоритмов обработки; функциональная полнота системы (решаются задачи ввода данных, обработки, управления, анализа, хранения измерительных данных и результатов анализа); хорошие метрологические характеристики и возможность тиражирования разработанных систем.

Перспективной является тенденция построения ИС на базе типовых микропроцессорных средств, что обеспечивает массовость их применения. Стратегия создания таких систем состоит в объединении регистрирующих датчиков, аппаратуры сбора данных и цифровой обработки сигналов, а также средств программного обеспечения в единую информационную систему.

Большое значение для рассматриваемых ИС имеют обеспечение функциональной гибкости в части управления, выбора метода исследования и развитый пользовательский интерфейс. Для реализации таких свойств разрабатывается мощная полиэкранная графическая поддержка с использованием популярных в последнее время объектной метафоры и комбинированных методов представления информации (текст, графика, звук, видео).

Целевое назначение рассматриваемых ИС связано с регистрацией, обработкой и анализом данных физических и инженерных измерений, а также

созданием баз экспериментальных данных для исследования методов информационного обеспечения измерительных задач.

В качестве базового элемента ИС может быть выбран ПК с процессором i486 или Pentium с шиной стандарта ISA (или ISA/PCI). К дополнительному оборудованию ИС следует отнести (рис.2.):

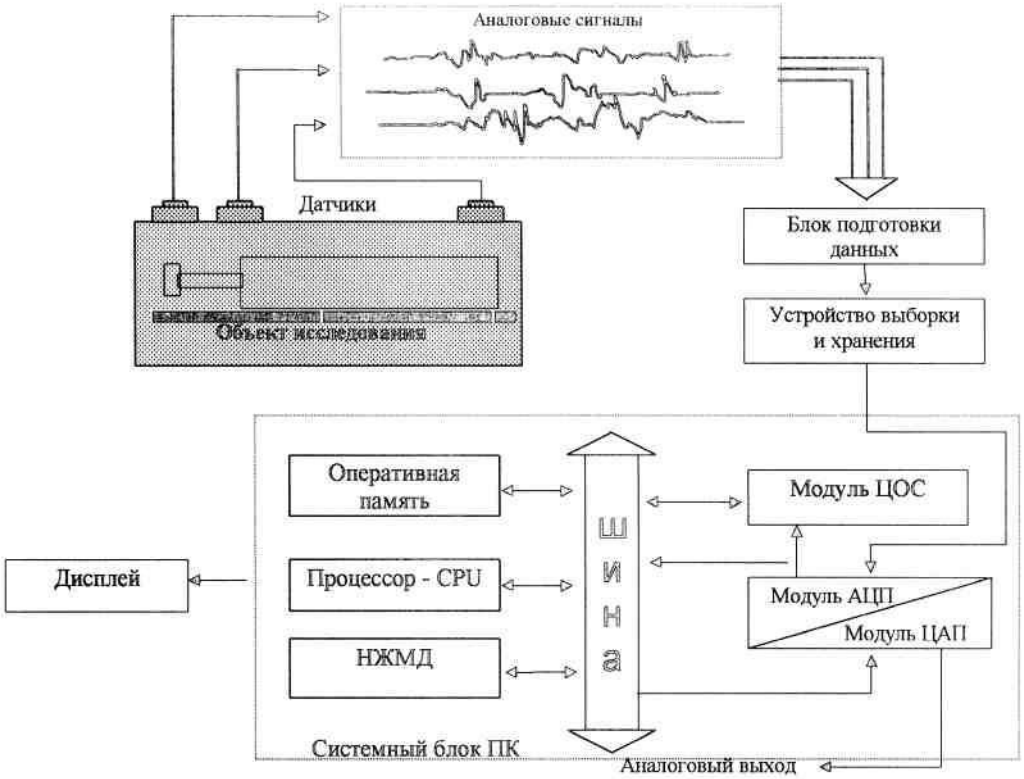


Рисунок 2. - Блок-схема измерительной системы

датчики физических параметров;

блок подготовки аналоговых сигналов (усиление, полосовая фильтрация);

интерфейсные средства ввода-вывода аналоговых сигналов (модули АЦП и ЦАП);

модуль цифрового процессора сигналов.

Элементы системы связаны между собой на физическом и (или) логико-функциональном уровне.

Ввод данных в ИС реализуется аппаратными средствами подсистемы сбора данных, а управляет процессом сбора пользователь, используя экранные формы интерфейса.

Структура ИС, приведенная на рис. 2, обеспечивает выполнение следующих основных задач:

автоматизированный синхронный ввод в ПК сигналов, регистрируемых группой датчиков;

вывод аналоговых сигналов в соответствии с аналитической моделью (например для калибровки);

обработка записанных на жесткий диск данных с помощью методов цифровой обработки сигналов (ЦОС) для изучения состояния физических объектов и исследования протекающих процессов;

графическое представление регистрируемой информации и результатов анализа;

хранение экспериментальных данных и результатов обработки.

Частотный диапазон сигналов, количество параллельных информационных каналов и динамический диапазон сигналов на входе определяют технические требования к системе. Технические требования являются основными исходными данными при выборе структуры измерительной системы (ИС) и разработке алгоритмов ввода многоканальных аналоговых сигналов в персональный компьютер. Типовые требования к ИС:

количество синхронных входных каналов 16;

частотный диапазон входных сигналов 10-30000 Гц;

разрядность АЦП/ЦАП 12-16 бит;

время преобразования АЦП 2.5-10 мкс;

порт ввода – вывода 8 бит ТТЛ;

динамический диапазон по входу 60-80 ДБ.

Программное обеспечение должно выполнять следующие функции:

настройка параметров и запуск процедуры сбора данных;

запись собираемых данных в оперативную память или на жесткий диск с отображением характера регистрируемых сигналов и временного изменения параметров на экране дисплея;

графический пользовательский интерфейс со средствами функциональной помощи;

реализация вычислительных алгоритмов цифровой обработки сигналов с отображением результатов комбинированными средствами представления информации;

выполнение калибровки передаточных характеристик физико-информационных преобразователей и аналоговых цепей;

поддержка базы экспериментальных данных о характеристиках объектов испытаний.

При разработке программного обеспечения используются следующие принципы: модульность, использование объектной метафоры в управлении, унификация связей, разделение программ управления, графической поддержки, обработки и доступа к базе данных.

3. Принципы функционирования интерфейса

Существует несколько методов реализации интерфейса АЦП – процессор ПК.

Схема “самых последних данных”. В этом методе реализации интерфейса АЦП работает непрерывно. В конце каждого цикла преобразования он обновляет данные в выходном буферном регистре и затем автоматически начинает новый цикл преобразования. Микропроцессор просто считывает содержимое этого буфера, когда ему нужны самые последние данные. Этот ме-

тод подходит для тех применений, где необходимость в обновлении данных возникает лишь от случая к случаю.

Схема “запуска-ожидания”. Микропроцессор инициирует выполнение преобразования каждый раз, когда ему нужны новые данные, и затем непрерывно тестирует состояние АЦП, чтобы узнать, закончилось ли преобразование. Зафиксировав конец преобразования, он считывает выходное слово преобразователя. В возможной модификации этого метода микропроцессор находится в состоянии ожидания в течение интервала времени, превышающего предполагаемое время преобразования, и затем считывает выходные данные. Этот метод несколько проще в реализации, но при этом микропроцессор отвлекается от выполнения всех других программ на время преобразования.

Использование прерывания микропроцессора. Этот метод основан на возможности использования системы прерываний микропроцессора. Как и в предыдущей схеме, процессор или таймер запускают преобразователь, но затем микропроцессор может продолжать выполнение других заданий. Когда преобразование завершено, АЦП вызывает прерывание микропроцессора. Микропроцессор прекращает выполнение текущей программы и сохраняет всю необходимую информацию для последующего восстановления этой программы. Затем он осуществляет поиск и использование ряда команд (обслуживающая программа – обработчик прерывания), предназначенных для выборки данных от АЦП. После того как обслуживающая программа выполнена, микропроцессор возвращается к выполнению исходной программы.

Задача поиска обслуживающей программы иногда решается путем выполнения другой программы (программы или процедуры последовательного опроса – поллинга), которая определяет источник прерывания путем последовательной проверки всех возможных источников. Гораздо эффективнее подход, связанный с использованием векторных прерываний. Этот подход основан на хранении адресов отдельных обслуживающих программ в заранее определенной области памяти, называемой векторной таблицей. В ответ на сигнал прерывания микропроцессор теперь обращается к определенной ячейке памяти, в которую пользователем занесен адрес соответствующей обслуживающей программы. Реальная эффективность этого метода проявляется в системах с большим числом источников прерываний, как в случае IBM PC. В таких системах, как правило, используется специальное устройство, называемое контроллером прерываний. Контроллер прерываний, например Intel 8259A (другие семейства микропроцессоров имеют эквивалентные устройства), организует различные приходящие сигналы прерываний в приоритетные очереди (выстраивает в порядке их значимости), посылает сигнал прерывания в микропроцессор и указывает ему на нужную ячейку в векторной таблице.

4. Программное обеспечение интерфейса

Передача данных между АЦП и микропроцессором на программном уровне может быть организована тремя способами.

Передача через пространство основной памяти. При распределении памяти АЦП присваивается некоторый адрес в пространстве основной памяти, не используемый для фактического хранения данных и программ. Передача данных между АЦП и микропроцессором осуществляется путем обращения к АЦП просто как к ячейке памяти с данным адресом. Однако помимо уменьшения полезного пространства памяти такой подход может привести к усложнению управления памятью и, как правило, требует использования дополнительных аппаратных средств дешифрации адреса, поскольку при минимуме этих средств, слишком расточительно используется память.

Передача через пространство подсистемы ввода – вывода (ВВ). В некоторых системах создается отдельный набор адресов для подсистемы ВВ (пространство ВВ), которые могут совпадать по численным значениям с адресами ячеек основной памяти, но отличаются от них с помощью использования специальных управляющих сигналов (IOR и IOW), выдаваемых на системную шину PC. Отделение пространства памяти от пространства ВВ улучшает характеристики системы. Как правило, это позволяет довольно просто осуществлять дешифрацию адреса с использованием минимального количества аппаратных средств, поскольку “приносится в жертву” пространство ВВ, а не очень ценное пространство основной памяти.

Прямой доступ к памяти (ПДП). Если возникает необходимость только в простой передаче данных между памятью и каким-либо периферийным устройством, включение в интерфейс регистра - аккумулятора микропроцессора неоправданно уменьшает скорость передачи данных. Используя дополнительные аппаратные средства, обычно в виде специального устройства, называемого контроллером ПДП, можно осуществлять непосредственную передачу данных с гораздо большей скоростью. Большинство микропроцессоров допускает реализацию ПДП путем передачи управления системной шиной на определенный промежуток времени контроллеру ПДП. Контроллер ПДП в течение этого промежутка времени управляет работой шины (захватывает шину) и обеспечивает передачу данных путем генерации соответствующих адресов и управляющих сигналов. Затем управление системной шиной передается обратно микропроцессору. Для передачи всех данных может потребоваться несколько таких ПДП-циклов. ПДП эффективен в тех применениях, где нужно обеспечить высокую скорость передачи данных или нужно передавать большие объемы данных. Применение этого метода в системах сбора данных в принципе возможно, но характерно только для систем с высокими рабочими параметрами. На системной плате PC имеется восьмика-нальный контроллер ПДП, который выполняет некоторые системные функции, включая регенерацию памяти и обмен информацией с диском.

5. Аппаратные средства интерфейса

Характер использования аппаратных средств в сильной степени зависит от того, в какой форме представляются данные – в последовательной, или в параллельной.

Параллельная форма представления данных. Аппаратные средства параллельного интерфейса почти всегда включают буфер с тремя состояниями (тристабильный буфер), через который АЦП подключается к шине данных микропроцессора. Дешифрованный адрес и вырабатываемый микропроцессором управляющий сигнал (строб) чтения используются для отпирания этого буфера и передачи данных от АЦП к микропроцессору. Тот же самый адрес и вырабатываемый микропроцессором управляющий сигнал записи используются для запуска преобразователя. В общем случае, наличие отдельных управляющих сигналов чтения и записи необязательно, но такой подход позволяет использовать один и тот же адрес при передаче команд к АЦП и считывании данных с выхода АЦП.

В большинстве АЦП нового поколения тристабильные буферы вместе со своими управляющими схемами находятся на самом кристалле. Такие АЦП можно непосредственно подключать к шине данных микропроцессора. Для сопряжения этих устройств с процессором пользователь должен только обеспечить дешифрованный адрес и ввести несколько логических элементов для согласования управляющих сигналов.

Последовательная форма представления данных. Последовательная форма представления данных естественна для систем, в которых используется последовательная передача данных на большие расстояния к станциям контроля (диспетчерским станциям). По экономическим показателям исключительно эффективным средством реализации такой передачи данных является асинхронная последовательная передача с использованием специализированных или телефонных линий с модемами на каждом конце линии. Аппаратные средства интерфейса со стороны микропроцессора, обычно находящегося на станции контроля, чаще всего представлены в виде специального устройства, называемого универсальным асинхронным приемопередатчиком (УАПП). УАПП принимает и передает данные в последовательной форме, но обменивается этими данными с микропроцессором через параллельный интерфейс. Для каждого микропроцессора имеется, по меньшей мере, один совместимый с ним УАПП. Интерфейс на том конце линии передачи, где находится АЦП, в сильной степени зависит от выбора АЦП, и его лучше всего рассматривать отдельно в каждом конкретном случае. Наблюдается тенденция к размещению большинства обеспечивающих интерфейс схем на самом кристалле АЦП.

Сопряжение 10- или 12-разрядного АЦП с 8-разрядной шиной данных довольно просто решается путем передачи данных порциями по 8 бит (1 байт) одна за другой. Этот способ пригоден как для параллельного, так и для последовательного интерфейсов.

Лекция 2.2. Управление задачами

1. Переключение контекста.
2. Прерывания.

1. Переключение контекста

Рассмотрим сущность понятия «переключение контекста».

Контекст задачи - это набор данных, задающих состояние процессора при выполнении задачи. Он обычно совпадает с набором регистров, доступных для изменения прикладной задачи. В системах с виртуальной памятью может включать регистры, отвечающие за трансляцию виртуального адреса в физический (обычно доступны на запись только операционной системе).

Переключение задач - это переход процессора от исполнения одной задачи к другой. Может быть инициировано:

1. Планировщиком задач (например, освободился ресурс и в очередь готовых задач попала ожидавшая его приоритетная задача),

2. Прерыванием (аппаратным прерыванием, например, запрос на обслуживание от внешнего устройства),

3. Исключением (программным прерыванием, например, системный вызов).

Поскольку контекст полностью определяет, какая задача будет выполняться, то часто термины «переключение задач» и «переключение контекста» употребляют как синонимы.

Диспетчер (dispatcher) - это модуль (программа), отвечающий за переключение контекста.

При переключении задач диспетчеру необходимо: 1) корректно остановить работающую задачу, для этого необходимо: а) выполнить инструкции текущей задачи, уже загруженные в процессор, но еще не выполненные (современные процессоры имеют внутри себя конвейеры инструкций, куда могут загружаться более 10 инструкций, некоторые из которых могут быть сложными, например, записать в память 32 регистра), обычно это делается аппаратно;

б) сохранить в оперативной памяти регистры текущей задачи;

2) найти, подготовить и загрузить затребованную задачу (обработчик прерываний - в этом случае требуется еще установить источник прерывания);

3) запустить новую задачу, для этого:

а) восстановить из оперативной памяти регистры новой задачи (сохраненные ранее, если она до этого уже работала);

б) загрузить в процессор инструкции новой задачи (современные процессоры начинают выполнять инструкции только после загрузки конвейера), эта фаза делается аппаратно.

Каждая из этих стадий вносит свой вклад в задержку при переключении контекста. Поскольку любое приложение реального времени должно обеспечить выдачу результата в заданное время, то эта задержка должна быть мала, детерминирована и известна. Это число является одной из важнейших характеристик ОСРВ.

2. Прерывания

Прерывания являются основным источником сообщения внешним устройствам о готовности данных или необходимости передачи данных. По

самому назначению систем реального времени, прерывания являются одним из основных объектов в ОСРВ.

Время реакции на прерывание - это время переключения контекста от текущей задачи к процедуре обработки прерывания. В многозадачных системах время ожидания прерывания (события) может быть использовано другой задачей. Прерывание может произойти во время обработки системного вызова и во время критической секции.

Рассмотрим суть понятия прерывания.

Прерывания представляют собой механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора. Таким образом, прерывание - это принудительная передача управления от выполняемой программы к системе (а через нее - к соответствующей программе обработки прерывания), происходящая при возникновении определенного события. Идея прерываний была предложена в середине 50-х годов и можно без преувеличения сказать, что она внесла наиболее весомый вклад в развитие вычислительной техники. Основная цель введения прерываний - реализация асинхронного режима работы и распараллеливание работы отдельных устройств вычислительного комплМекесхаа.низм прерываний реализуется аппаратно-программными средствами. Структуры систем прерывания (в зависимости от аппаратной архитектуры) могут быть самыми разными, но все они имеют одну общую особенность - прерывание непременно влечет за собой изменение порядка выполнения команд процессором.

Механизм обработки прерываний независимо от архитектуры вычислительной системы включает следующие элементы:

1. Установление факта прерывания (прием сигнала на прерывание) и идентификация прерывания.

2. Запоминание состояния прерванного процесса. Состояние процесса определяется, прежде всего, значением счетчика команд (адресом следующей команды, который, например, в i80x86 определяется регистрами CS и IP – указателем команды, содержимым регистров процессора и может включать также спецификацию режима, например, режим пользовательский или привилегированный) и другую информацию.

3. Управление аппаратно передается подпрограмме обработки прерывания. В простейшем случае в счетчик команд заносится начальный адрес

подпрограммы обработки прерываний, а в соответствующие регистры - информация из слова состояния. В более развитых процессорах осуществляется достаточно сложная процедура определения начального адреса соответствующей подпрограммы обработки прерывания, и не менее сложная процедура инициализации рабочих регистров процессора.

4. Сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью действий аппаратуры. В некоторых вычислительных системах предусматривается запоминание довольно большого объема информации о состоянии прерванного процесса.

5. Обработка прерывания. Эта работа может быть выполнена той же подпрограммой, которой было передано управление на шаге 3, но в ОС чаще всего она реализуется путем последующего вызова соответствующей подпрограммы.

6. Восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4).

7. Возврат в прерванную программу.

Шаги 1 - 3 реализуются аппаратно, а шаги 4 - 7 - программно.

На рис. 1 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается программе обработки возникшего прерывания. При этом средствами аппаратуры сохраняется (как правило, с помощью механизмов стековой памяти) адрес той команды, с которой следует продолжить выполнение прерванной программы. После выполнения программы обработки прерывания управление возвращается прерванной ранее программе посредством занесения в указатель команд сохраненного адреса команды. Однако такая схема используется только в самых простых программных средах. В мультипрограммных системах обработка прерываний происходит по более сложным схемам.

Главными функциями механизма прерываний являются:

- распознавание или классификация прерываний;
- передача управления соответственно обработчику прерываний;
- корректное возвращение к прерванной программе.

Рисунок 1. - Обработка прерывания

Переход от прерываемой программы к обработчику и обратно должен выполняться как можно быстрее. Одним из быстрых методов является использование таблицы, содержащей перечень всех допустимых прерываний и адреса соответствующих обработчиков. Для корректного возвращения к прерванной программе перед передачей управления обработчику прерываний содержимое регистров процессора запоминается либо в памяти с прямым доступом, либо в системном стеке. Прерывания, возникающие при работе вычислительной системы, разделяются на два основных класса: внешние (асинхронные) и внутренние (синхронные).

Внешние прерывания вызываются асинхронными событиями, которые происходят вне прерываемого процесса, например:

прерывания от таймера;

прерывания от внешних устройств (прерывания по вводу/выводу);

прерывания по нарушению питания;

прерывания с пульта оператора вычислительной системы;

прерывания от другого процессора или другой вычислительной системы.

Внутренние прерывания вызываются событиями, которые связаны с работой процессора и являются синхронными с его операциями. Примерами являются следующие запросы на прерывания:

при нарушении адресации (в адресной части выполняемой команды указан запрещенный или несуществующий адрес, обращение к отсутствующему сегменту или странице при организации механизмов виртуальной памяти);

при наличии в поле кода операции незадействованной двоичной комбинации;

при делении на нуль;

при переполнении или исчезновении порядка;

при обнаружении ошибок четности, ошибок в работе различных устройств аппаратуры средствами контроля.

Могут существовать **прерывания при обращении к супервизору ОС** -в некоторых компьютерах часть команд может использовать только ОС, а не пользователи. Соответственно в аппаратуре предусмотрены различные режимы работы, и пользовательские программы выполняются в режиме, в котором эти привилегированные команды не исполняются. При попытке использовать команду, запрещенную в данном режиме, происходит внутреннее прерывание, и управление передается супервизору ОС. К привилегированным командам относятся и команды переключения режима работа центрального процессора.

Наконец, существуют собственно **программные прерывания**.Эти прерывания происходят по соответствующей команде прерывания, то есть по этой команде процессор осуществляет практически те же действия, что и при обычных внутренних прерываниях. Данный механизм был специально введен для того, чтобы переключение на системные программные модули про-

исходило не просто как переход в подпрограмму, а точно таким же образом, как и обычное прерывание. Этим обеспечивается автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд.

Сигналы, вызывающие прерывания, формируются вне процессора или в самом процессоре; они могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому типу прерывания. Очевидно, что прерывания от схем контроля процессора должны обладать наивысшим приоритетом (если аппаратура работает неправильно, то не имеет смысла продолжать обработку информации). На рис. 2 изображен обычный порядок обработки прерываний в зависимости от типа прерываний. Учет приоритета может быть встроен в технические средства, а также определяться операционной системой, то есть кроме аппаратно реализованных приоритетов прерывания большинство вычислительных машин и комплексов допускают программно-аппаратное управление порядком обработки сигналов прерывания. Второй способ, дополняя первый, позволяет применять различные **дисциплины обслуживания прерываний.**

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы. Процессор может обладать средствами защиты от прерываний: отключение системы прерываний, маскирование (запрет) отдельных сигналов прерывания. Программное управление этими средствами (существуют специальные команда для управления работой системы прерываний) позволяет операционной системе регулировать обработку сигналов прерывания, заставляя процессор обрабатывать их сразу по приходу, откладывать их обработку на некоторое время или полностью игнорировать. Обычно операция прерывания выполняется только после завершения выполнения текущей команды. Поскольку сигналы прерывания возникают в произвольные моменты времени, то на момент прерывания может существовать несколько сигналов прерывания, которые могут быть обработаны только последовательно. Чтобы обработать сигналы прерывания в разумном порядке им присваиваются приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается.

Рисунок 2. - Распределение прерываний по уровням приоритета

Программное управление специальными регистрами маски (маскирование сигналов прерывания) позволяет реализовать различные дисциплины обслуживания:

с относительными приоритетами, то есть обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса обслуживается запрос с наивысшим приоритетом. Для организации такой дисциплины необходимо в программе обслуживания данного запроса наложить маски на все остальные сигналы прерывания или просто отключить систему прерываний;

с **абсолютными приоритетами**, то есть всегда обслуживается прерывание с наивысшим приоритетом. Для реализации этого режима необходимо на время обработки прерывания замаскировать все запросы с более низким приоритетом. При этом возможно многоуровневое прерывание, то есть прерывание программ обработки прерываний. Число уровней прерывания в этом режиме изменяется и зависит от приоритета запроса;

по **принципу стека**, или, как иногда говорят, **по дисциплине LCFS** (last come first served - последним пришел - первым обслужен), то есть запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Дли этого необходимо не накладывать маски ни на один сигнал прерывания и не выключать систему прерываний.

Следует особо отметить, что для правильной реализации последних двух дисциплин нужно обеспечить полное маскирование системы прерываний при выполнении шагов 1-4 и 6-7. Это необходимо для того, чтобы не потерять запрос и правильно его обслужить. Многоуровневое прерывание должно происходить на этапе собственно обработки прерывания, а не на этапе перехода с одного процесса на другой.

Управление ходом выполнения задач со стороны ОС заключается в организации реакций на прерывания, в организации обмена информацией (данными и программами), предоставлении необходимых ресурсов, в дина-

мике выполнения задачи и в организации сервиса. Причины прерываний определяет ОС (модуль, который называют супервизором прерываний), она же и выполняет действия, необходимые при данном прерывании и в данной ситуации. Поэтому в состав любой ОС реального времени, прежде всего, входят программы управления системой прерываний, контроля состояний задач и событий, синхронизации задач, средства распределения памяти и управления ею, а уже потом средства организации данных (с помощью файловых систем и т. д.). Современная ОС реального времени должна вносить в аппаратно-программный комплекс нечто большее, нежели просто обеспечение быстрой реакции на прерывания.

При появлении запроса на прерывание система прерываний идентифицирует сигнал и, если прерывания разрешены, управление передается на соответствующую подпрограмму обработки. Из рис.1 видно, что в подпрограмме обработки прерывания имеются две служебные секции. В первой секции осуществляется сохранение контекста прерванной задачи, который не смог быть сохранен на 2-м шаге, и последняя, заключительная секция, в которой, наоборот, осуществляется восстановление контекста. Чтобы система прерываний не среагировала повторно на сигнал запроса на прерывание, она обычно автоматически «закрывает» (отключает) прерывания, поэтому необходимо потом в подпрограмме обработки прерываний вновь включать систему прерываний. Установка рассмотренных режимов обработки прерываний осуществляется в конце первой секции подпрограммы обработки. Таким образом, на время выполнения центральной секции прерывания разрешены. На время работы заключительной секции подпрограммы обработки система прерываний должна быть отключена, и после восстановления контекста вновь включена. Поскольку эти действия необходимо выполнять практически в каждой подпрограмме обработки прерываний, во многих операционных системах первые секции подпрограмм обработки прерываний выделяются в специальный системный программный модуль, называемый **супервизором прерываний.**

Супервизор прерываний, прежде всего, сохраняет в дескрипторе текущей задачи рабочие регистры процессора, определяющие контекст прерываемого вычислительного процесса. Далее он определяет ту подпрограмму, которая должна выполнить действия, связанные с обслуживанием текущего запроса на прерывание. Перед тем как передать управление этой подпрограмме, супервизор прерываний устанавливает необходимый режим обработки прерывания. После выполнения подпрограммы обработки прерывания управление вновь передается супервизору, на этот раз уже на тот модуль, который занимается диспетчеризацией задач. И уже диспетчер задач, в свою очередь, в соответствии с принятым режимом распределения процессорного времени восстановит контекст той задачи, которой будет решено выделить процессор. В данном случае нет непосредственного возврата в прерванную ранее программу прямо из самой подпрограммы обработки прерывания. Для прямого непосредственного возврата достаточно адрес возврата сохранить в стеке, что и делает аппаратура процессора. При

этом стек легко обеспечивает возможность возврата в случае вложенных прерываний, поскольку он всегда реализует дисциплину (последним пришел – первым обслужен).

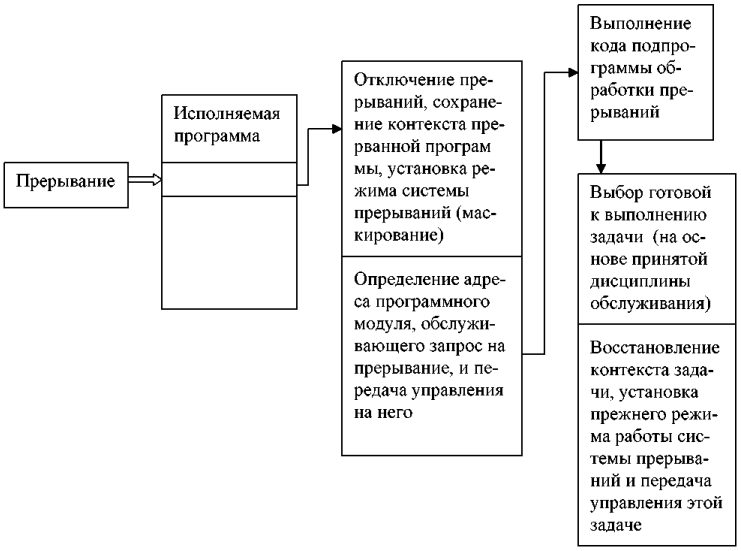


Рисунок 3. - Обработка прерывания с использованием супервизора прерываний

Однако если бы контекст процессов сохранялся в стеке, как это обычно реализуется аппаратурой, а не в описанных выше дескрипторах задач, то не было бы возможности гибко подходить к выбору той задачи, которой нужно передать процессор после завершения работы подпрограммы обработки прерывания. Это только общий принцип. В конкретных процессорах и в конкретных ОС могут существовать некоторые отступления от рассмотренной схемы и/или дополнения к ней. Например, в современных процессорах часто имеются специальные аппаратные возможности для сохранения контекста прерываний.

Лекция 2.3. Управление системными ресурсами

1. Однопроцессорная и распределенная архитектуры.
2. Функции операционных систем в среде реального времени.
3. Управление процессором и состояния процесса.
4. Стратегии выбора процесса.

1. Однопроцессорная и распределенная архитектуры Рассмотрим **распределенные системы** (distributed systems), которые по своей природе больше подходят для управления сложными процессами. К основным преимуществам распределенных систем относятся:

- экономичность;
- надежность (при отказе нескольких процессоров остальные продолжают работать);
- возможность подобрать аппаратные средства в соответствии с конкретными требованиями.

Говоря о распределенной системе, необходимо иметь в виду, каким способом достигается распределение ресурсов. Одна крайность - когда единственным общим ресурсом является сеть, соединяющая ЭВМ, каждая из которых работает независимо и лишь обменивается сообщениями с остальными. Другая крайность - реально распределенная сетевая операционная система, предоставляющая пользователю гомогенную среду, не зависящую от аппаратной платформы. Пользователь может вводить произвольные команды, а операционная система находит наиболее подходящий способ и место их выполнения.

Распределенные системы используются в управлении процессами, поскольку эти приложения являются принципиально распределенными и такая архитектура обеспечивает более полное соответствие между аппаратными и программными средствами и управляемым объектом. Сложный технологический процесс можно разбить на несколько уровней, на каждом из которых собираются и обобщаются (агрегируются) данные, передающиеся на более высокие уровни. Такой тип распределенной системы более экономичен, чем централизованная система с одним процессором, она надежна в том смысле, что отказ одного из компонентов не нарушает работу других (при условии, что система хорошо структурирована), и ее можно построить таким образом, чтобы она в максимальной степени соответствовала управляемому процессу.

Однако чисто аппаратный подход к надежности не решает всех проблем. В распределенной системе процессы, исполняющиеся на разном оборудовании, зависят и друг от друга, и от коммуникаций. Если процесс или оборудование в одном узле перестанет работать или возникнут проблемы с коммуникациями, то остановится исполнение не только конкретного процесса, но и процессов, зависящих от него, потому, например, что они ждут ответа на свои вопросы.

По сравнению с централизованными - распределенные системы требуют принципиально иных программных средств, поскольку такие системы тесно связаны сетью. Сетевая операционная система должна управлять как ресурсами отдельных ЭВМ, так и всей сети. Поэтому функции операционной системы нельзя отделять от функциональных свойств сети, а работа сети оказывает заметное влияние на работу распределенной системы. Фактически сетевые операционные системы имеют многоуровневую структуру, аналогично стеку коммуникационных протоколов.

Главным различием между однопроцессорной и распределенной архитектурами является способ обмена информацией между процессами. Эта процедура наиболее важна при мультипрограммировании и программировании в реальном времени. В однопроцессорной конфигурации обмен данными между процессами происходит через общую локальную память, очередность доступа к которой регулируется многозадачной операционной системой.

В отличие от этого, в распределенной системе нет общей памяти как таковой, и процессы обмениваются информацией с помощью сообщений. Если один процесс должен передать информацию другому, то он формирует

сообщение и обращается к услугам операционной системы для передачи его по назначению.

Этот принцип взаимодействия лежит в основе одной из наиболее важных концепций распределенных операционных систем - модели "клиент-сервер". В этой модели процесс либо запрашивает услуги - клиент, либо предоставляет их - сервер. Очевидно, что один и тот же процесс может быть как клиентом, так и сервером. "Услуга" - это некоторая законченная (замкнутая) операция, в частности выполнение расчетов, прием внешних данных, операция с устройством, например, вывод изображения на экран. В определенном смысле модель "клиент-сервер" можно рассматривать как расширенный вариант обращения к подпрограмме, при котором сервер играет роль подпрограммы или системной процедуры.

Модель "клиент-сервер" основана на обмене сообщениями между программами. Если клиент и сервер исполняются на разных ЭВМ, а сообщения передаются через сеть, то система является распределенной.

Чем больше вычислительные ресурсы процедур клиента и сервера и чем больше сложных функций они могут выполнять независимо, тем меньше число сообщений и, соответственно, нагрузка на сеть. Фактически важным преимуществом распределенных систем является то, что ресурсоемкие вычисления можно выполнять локально и в результате уменьшить объем трафика, поскольку передается только информация, относящаяся к более высокому абстрактному уровню, чем локальные вычисления, то есть некоторый итог локальных операций. Иными словами, в хорошо спроектированной системе сообщения содержат информацию о цели ("установить опорное значение x = 78.2"), а не о том, какие шаги следует для этого предпринять ("каково значение x в данный момент?", "x = 63", "увеличить на 16", "каково x сейчас?", "x = 79", уменьшить на 1", и т. д.). Промежуточные шаги выполняются локально при условии, что программное обеспечение спроектировано соответствующим образом.

2. Функции операционных систем в среде реального времени

Операционная система (ОС, Operating System - OS) - это сложный программный продукт, предназначенный для управления аппаратными и программными ресурсами вычислительной системы. Она предоставляет каждому процессу виртуальную (логическую) среду, включающую в себя время процессора и память. "Виртуальная среда" - это концептуальное понятие. Ее характеристики могут, как совпадать, так и не совпадать с параметрами реального оборудования.

Многозадачность сейчас доступна почти на всех типах ЭВМ, и ее поддержка является одной из основных характеристик таких операционных систем, как UNIX и Windows NT и выше. В первую очередь многозадачность должна обеспечивать распределение и защиту ресурсов. Первоначальной целью создания многозадачных систем, или систем разделения времени (time-sharing systems), было желание обеспечить одновременный доступ нескольких пользователей к дорогим вычислительным ресурсам и, соответственно, разделить между ними эксплуатационные расходы, то есть повысить эконо-

мическую эффективность оборудования. В системах реального времени целью многозадачного режима является изоляция друг от друга разных операций и распределение рабочей нагрузки между отдельными программными модулями. Единственным "пользователем" в этом случае является управляемая система.

В системах разделения времени, или многопользовательских системах, большое внимание уделяется защите и изоляции пользователей друг от друга с помощью паролей, управления доступом, учета использования ресурсов и т. д. Системы реального времени в этом смысле имеют меньше ограничений, поскольку разработчик всегда знает, что делает каждый модуль. В ситуациях, когда ценится каждая миллисекунда машинного времени, его нельзя тратить на дополнительные расходы по контролю доступа, поэтому файловые системы и механизмы защиты не являются важными компонентами операционных систем реального времени. Многопользовательские системы должны быть, в определенном смысле, "справедливыми", поскольку даже в режиме большой нагрузки нельзя допускать дискриминации ни одного пользователя. Наоборот, в приоритетных системах реального времени процессы четко разграничены с точки зрения права доступа к ресурсам процессора.

В распределенной среде операционная система дополнительно выполняет функции сопряжения программ с сетью и управления обменом данными и сообщениями между ЭВМ. В сетевых операционных системах каждая ЭВМ имеет высокую степень автономности. Общесистемные требования к обмену информацией позволяют взаимодействовать процессам, даже если они работают под управлением разных операционных систем, при условии, что каждая из них обладает необходимыми сетевыми возможностями.

3. Управление процессором и состояния процесса

Основными объектами в многозадачной среде являются процессы или задачи, описываемые своим контекстом. На одном процессоре в любой момент времени может исполняться только одна задача. Контекст исполняемой задачи всегда можно "заморозить", сохранив содержимое регистров процессора. При остановке текущей задачи процессор продолжает исполнение других задач. Таким образом, процессор есть ограниченный ресурс, который распределяется между всеми задачами.

На одном процессоре для организации многозадачного режима выполнение каждой задачи разбивается на несколько коротких интервалов (рис.1). Процессор выполняет часть первой задачи, затем второй, третьей и т. д. Временной интервал, выделенный для каждой задачи, составляет, например, 10 миллисекунд.

Внешний эффект разделения процессорного времени между задачами состоит в параллельном выполнении n задач. Когда n задач выполняются в системе параллельно каждая из них в среднем монопольно "располагает" процессором с производительностью 1/n, т. е. работает (развивается) на виртуальном процессоре, производительность которого в n раз меньше, чем у реального физического процессора. Если вместо одного используется не-

сколько процессоров, то это просто другая реализация того же самого логического принципа. В первом случае процессы разделены во времени, во втором – в пространстве. Если исключить накладные расходы на планирование и межзадачное взаимодействие, то при выполнении n процессов на k одинаковых процессорах каждому процессу в среднем выделяется виртуальный процессор с производительностью, равной k/n части от производительности одного физического процессора.

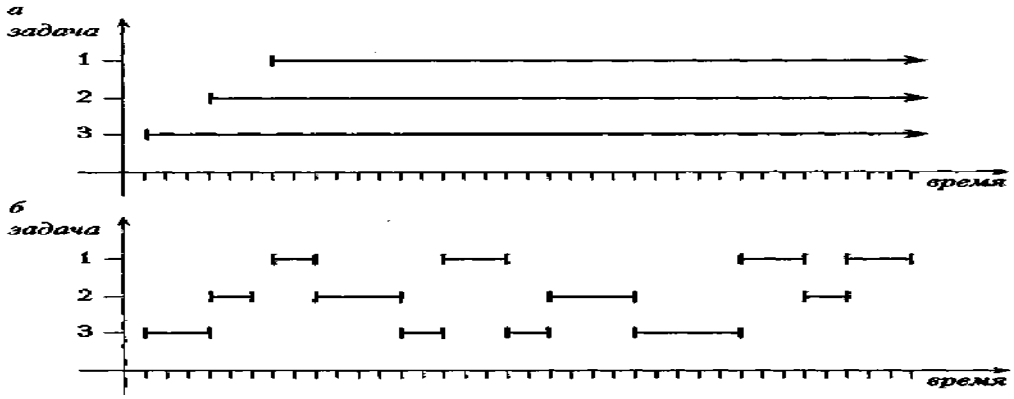


Рисунок 1. - Принцип организации многозадачного режима: а - внешний эффект; б - распределение времени процессора

Простейшая многозадачная однопроцессорная система состоит из процедуры, сохраняющей контекст текущего процесса в стеке или в определенной области памяти и восстанавливающей контекст другого процесса, исполнение которого возобновляется с того места, где он был прерван. Системная программа, выбирающая один из готовых для исполнения процессов, называется планировщиком (scheduler). Стратегии выбора достаточно разнообразны и меняются от одной операционной системы к другой, однако чаще всего используется какой-либо механизм на основе приоритетов. Планировщик работает как один из процессов, который автоматически получает управление после каждого прерывания текущего процесса.

Операции по переключению процессов критичны по времени и должны осуществляться с максимальной эффективностью. На процессорах, первоначально не разработанных для многозадачного режима, процедура сохранения и восстановления контекста - переключение процессов - реализуется длинной последовательностью стандартных инструкций процессора. В набор команд процессора, спроектированного для работы в многозадачном режиме, входят специальные инструкции для сохранения и восстановления контекста. Переменные процесса не входят в состав контекста, и сохранять их специально нет необходимости, поскольку они хранятся в памяти, выделенной процессу и защищенной операционной системой от доступа других процессов.

Планировщик вызывается обычно после обработки каждого прерывания. Если используется стратегия переключения процессов на основе квантования времени (рис. 1), необходимо иметь внешний по отношению к процессору интервальный таймер, вырабатывающий прерывания по истечении

кванта времени (time slice), выделенного процессу. При возникновении прерывания исполнение текущего процесса приостанавливается и проверяется, должен ли быть прерван текущий процесс и загружен новый. Принудительная приостановка текущего процесса для передачи управления другому процессу называется вытеснением (preemption).

Продолжительность кванта времени влияет на производительность системы. При коротком кванте улучшается общее время обслуживания коротких процессов. Если величина кванта сопоставима с временем, необходимым для переключения процессов, то большая часть ресурсов процессора будет уходить на планирование и переключение. Если величина кванта слишком большая, увеличивается время ожидания процессов и, соответственно, время реакции.

Процесс исполняется до тех пор, пока не произойдет одно из следующих событий:

- истек выделенный ему квант времени;
- процесс заблокирован, например, ждет завершения операции ввода/вывода;
- процесс завершился;
- вытеснен другим процессом, имеющим более высокий приоритет, например обработчиком прерываний.

В многозадачной среде процесс может находиться в одном из трех состояний (рис. 2).

• Готов (ready). Процесс может начать исполнение, как только освободится процессор.

• Исполнение (running, executing). Процесс исполняется в данный момент, т. е. процессор исполняет его код.

• Ожидание, заблокирован (waiting, locked). Для продолжения работы процессу не хватает какого-либо ресурса, за исключением ЦП, либо он ждет наступления внешнего события.

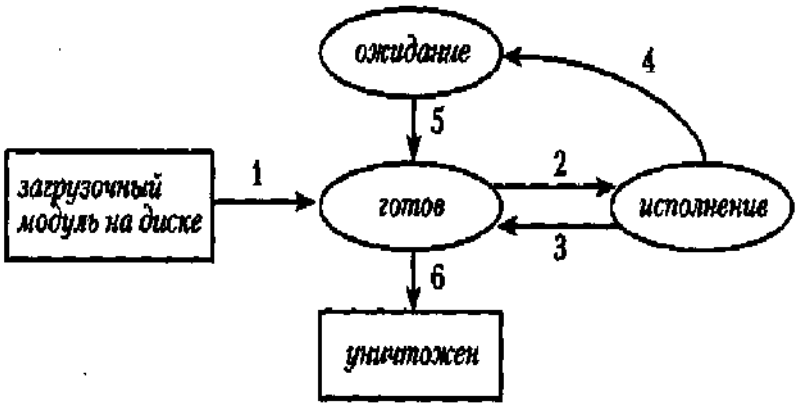


Рисунок 2. - Состояния процесса

На рис. 2 также показаны возможные переходы из одного состояния в другое:

1. От "Загрузочный модуль на диске" к "Готов". Программа загружается (load) в оперативную память, настраиваются относительные

адреса (relocation), выдёляются рабочие области для данных, кучи и стека с соответствующими указателями и создается контекст процесса.

2. От "Готов" к "Исполнение". Планировщик выбирает первый в очереди готовых процессов и передает ему управление.

3. От "Исполнение" к "Готов". Процесс либо исчерпал свой квант времени, либо появился готовый для исполнения процесс с более высоким приоритетом.

4. От "Исполнение" к "Ожидание". Для дальнейшего развития процесс должен ждать наступления какого-либо внешнего события (завершения операции ввода/вывода или освобождения ресурса, например доступа к памяти, заблокированной другим процессом, или сигнала от другого процесса и т. п.). Иногда процесс переводится в состояние ожидания до истечения некоторого интервала времени с помощью явной инструкции в его программе.

5. От "Ожидание" к "Готов". Когда ожидаемое событие произошло или истекло заданное время, процесс переводится в состояние "Готов" и помещается в очередь готовых процессов, откуда затем выбирается планировщиком.

6. После выполнения последней инструкции программы операционная система удаляет процесс из памяти и освобождает все выделенные ему ресурсы, включая память.

4. Стратегии выбора процесса

Существует несколько возможных стратегий выбора готовых процессов из очереди. Для определения той или иной стратегии необходимо принимать во внимание несколько противоречащих друг другу факторов - общее время, необходимое для решения задачи, ограничение на время реакции, важность и т. п. Рассмотрим две стратегии аналогичные тем, которые применяются при арбитраже шины.

Наиболее простой стратегией выбора является циклический (round-robin) метод - процессы выбираются последовательно один за другим в фиксированном порядке и через равные интервалы времени. Основное достоинство метода - простота, однако, поскольку процессам с различными требованиями выделяются равные ресурсы процессора, некоторые из них обслуживаются неадекватно своим потребностям.

Более сложный принцип выбора основан на приоритетах (priorities). При каждом переключении планировщик передает управление готовому процессу с наивысшим приоритетом. Приоритет присваивается процессу в момент его создания и остается постоянным в течение всего времени - статический приоритет (static priority). Такой приоритет, как правило, определяется на основе информации, предоставленной пользователем.

Планирование на основе статических приоритетов может привести к неприятным ситуациям. Процесс с наивысшим приоритетом, если он не находится в состояния ожидания, будет всегда выбираться для исполнения и практически полностью занимать процессор. Нетривиальным является также выбор между процессами с одинаковым приоритетом. Для исключения подобной ситуации применяется какой-либо алгоритм динамического назначе-

ния приоритетов (dynamic priority allocation). Например, планировщик снижает приоритет исполняемого процесса на фиксированную величину. В результате его приоритет будет ниже, чем у другого готового процесса, который затем и выбирается для исполнения. Таким образом, обеспечивается выполнение всех процессов. Через некоторое время ожидающим процессам возвращаются номинальные значения их приоритетов. Этот метод обеспечивает исполнение процессов даже с низким приоритетом и гарантирует, что процесс с высоким начальным приоритетом не будет непрерывно занимать процессор.

Разница в первоначально назначенных приоритетах приводит к тому, что процессы с более высокими приоритетами будут получать управление чаще, чем другие. Процессы, обращение к которым происходит более интенсивно и/или время реакции которых ограничено, получают в начальный момент более высокие приоритеты; менее важным процессам, для которых допустима отложенная реакция, присваиваются более низкие приоритеты.

Планирование процессов, основанное на приоритетах, работает хорошо, только если разные процессы имеют неодинаковые приоритеты. Присвоение наивысших приоритетов всем процессам не повышает скорость исполнения, так как это не увеличивает быстродействие процессора, - каждый процесс будет ждать в очереди до тех пор, пока все остальные не будут выполнены. Система, в которой всем процессам присвоены одинаковые приоритеты, работает по циклическому принципу. Наилучшие результаты достигаются в системе реального времени, если относительные приоритеты тщательно выбраны и сбалансированы.

Лекция 2.4. Управление оперативной памятью

1. Отображение адресного пространства программы на основную память.

2. Функции операционной системы по управлению памятью.

1. Отображение адресного пространства программы на основную память.

Наиболее важным ресурсом после процессора является оперативная память. В отличие от памяти жесткого диска, которую называют внешней памятью, оперативной памяти для сохранения информации требуется постоянное электропитание. Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Особая роль памяти объясняется тем, что процессор может выполнять инструкции программы только в том случае, если они находятся в памяти. Память распределяется как между модулями прикладных программ, так и между модулями самой операционной системы.

В ранних ОС управление памятью сводилось к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в память. С появлением мультипрограммирования перед ОС были поставлены новые задачи, связанные с распределением имеющейся памяти между несколькими

одновременно выполняющимися программами. Функциями ОС по управлению памятью в мультипрограммной системе являются:

отслеживание свободной и занятой памяти;

выделение памяти процессам и освобождение памяти по завершении процессов;

вытеснение кодов и данных процессов из оперативной памяти на диск (полное или частичное), когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;

настройка адресов программы на конкретную область физической памяти.

Алгоритмы распределения, использования, освобождения ресурсов и представления к ним доступа предназначены для наиболее эффективной организации работы всего комплекса устройств ЭВМ. Рассмотрим их на примере управления основной памятью.

Для выполнения программы при ее загрузке в основную память ей выделяется часть машинных ресурсов - они необходимы для размещения команд, данных, управляющих таблиц и областей ввода-вывода, то есть, производится трансляция адресного пространства откомпилированной программы в местоположение в реальной памяти.

Выделение ресурсов может быть осуществлено самим программистом (если он работает на языке, близком машинному), но может производиться и операционной системой.

Если выделение ресурсов производится перед выполнением программы, такой процесс называется **статическим перемещением,** в результате которого программа «привязывается» к определенному месту в памяти вычислительной машины. Если же ресурсы выделяются в процессе выполнении программы, это называется динамическим перемещением, и в этом случае программа не привязана к определенному месту в реальной памяти. Динамический режим можно реализовать только с помощью операционной системы.

При статическом перемещении могут встретиться два случая:

1. Реальная память больше требуемого адресного пространства программы. В этом случае загрузка программы в реальную память производится, начиная с 0-го адреса.

Загружаемая программа А является абсолютной программой, так как никакого изменения адресов в адресном пространстве, подготовлен-

ном компилятором, при загрузке в основную память не происходит - программа располагается с 0-го адреса реальной памяти;

2. Реальная память меньше требуемого адресного пространства программы. В этом случае программист (или операционная система) вынужден решать проблему, как организовать выполнение программы. Методов решения проблемы существует несколько: можно создать оверлейную структуру (то есть разбить программу на части, вызываемые в ОП по мере необходимости), сделать модули программы реентерабельными (то есть

допускающими одновременную работу модуля по нескольким обращениям из разных частей программы или из различных программ).

В некоторых операционных системах адреса откомпилированной (с 0-го адреса) программы могут быть преобразованы в адреса реальной памяти, отличные от 0. При этом создается абсолютный модуль, который требует размещения его в памяти всегда с одного и того же адреса.

При мультипрограммном режиме, если имеем программы А, В и С, для которых известно, что программа А выполняется при размещении в памяти с адреса 60 Кбайт до 90 Кбайт, В - с 60 Кбайт до 90 Кбайт, С - с 50 Кбайт до 120 Кбайт, организовать их совместное выполнение невозможно, так как им необходим один и тот же участок реальной памяти. Эти программы будут ждать друг друга либо их нужно заново редактировать с другого адреса.

В системах с динамическим перемещением программ перемещающий загрузчик размещает программу в свободной части памяти и допускает использование ее несмежных участков. В этом случае имеется больше возможностей для организации мультипрограммной работы, а, следовательно, и для более эффективного использования временных ресурсов ЭВМ.

2. Функции операционной системы по управлению памятью Помимо первоначального выделения памяти процессам при их создании ОС должна также заниматься динамическим распределением памяти, то есть выполнять запросы приложений на выделение им дополнительной памяти во время выполнения. После того как приложение перестает нуждаться в дополнительной памяти, оно может возвратить ее системе. Выделение памяти случайной длины в случайные моменты времени из общего пула памяти приводит к фрагментации и, вследствие этого, к неэффективному ее использованию. Оно характерно для систем со статическим перемещением. Дефрагментация памяти тоже является функцией операционной системы.

Во время работы операционной системы ей часто приходится создавать новые служебные информационные структуры, такие как описатели процессов и потоков, различные таблицы распределения ресурсов, буферы, используемые процессами для обмена данными, синхронизирующие объекты. Все эти системные объекты требуют памяти. В некоторых ОС заранее (во время установки) резервируется некоторый фиксированный объем памяти для системных нужд. В других же ОС используется более гибкий подход, при котором память для системных целей выделяется динамически. В таком случае разные подсистемы ОС при создании своих таблиц, объектов, структур обращаются к подсистеме управления памятью с запросами.

Защита памяти - это еще одна важная задача операционной системы, которая состоит в том, чтобы не позволить выполняемому процессу записывать или читать данные из памяти, назначенной другому процессу. Эта функция, как правило, реализуется программными модулями ОС в тесном взаимодействии с аппаратными средствами.

Для идентификации переменных и команд на разных этапах жизненного цикла программы используются символьные имена (метки), виртуальные адреса и физические адреса.

Символьные имена присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.

Виртуальные адреса, называемые иногда математическими, или логическими адресами, вырабатывает транслятор, переводящий программу на машинный язык. Поскольку во время трансляции в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что начальным адресом программы будет нулевой адрес.

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды.

Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Диапазон возможных адресов виртуального пространства у всех процессов является одним и тем же.

Совпадение виртуальных адресов переменных и команд различных процессов не приводит к конфликтам, так как в том случае, когда эти переменные одновременно присутствуют в памяти, операционная система отображает их на разные физические адреса. В том случае, когда необходимо, чтобы несколько процессов разделяли общие данные или коды, операционная система отображает соответствующие участки виртуального адресного пространства этих процессов на один и тот же участок физической памяти.

В разных операционных системах используются разные способы структуризации виртуального адресного пространства. В одних ОС виртуальное адресное пространство процесса подобно физической памяти представлено в виде непрерывной линейной последовательности виртуальных адресов. Такую структуру адресного пространства называют плоской. При этом виртуальным адресом является единственное число, представляющее собой смещение относительно начала (обычно это значение 000...000) виртуального адресного пространства. Адрес такого типа называют линейным виртуальным адресом.

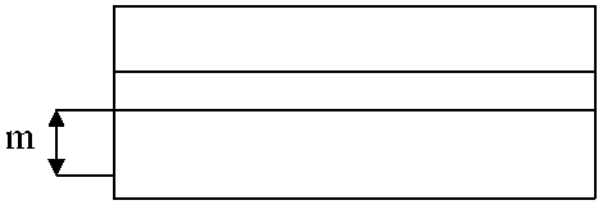


Рисунок 1. – Интерпретация виртуального адреса

В других ОС виртуальное адресное пространство делится на части, называемые сегментами (или секциями, или областями). Б этом случае

помимо линейного адреса может быть использован виртуальный адрес (рис.1), представляющий собой *пару* чисел *(n, m),* где *n* определяет сегмент, а m - смещение внутри сегмента.

Существуют и более сложные способы структуризации виртуального адресного пространства, когда виртуальный адрес образуется тремя или даже более числами.

Задачей операционной системы является отображение индивидуальных виртуальных адресных пространств, всех одновременно выполняющихся процессов, на общую физическую память. При этом ОС отображает либо все виртуальное адресное пространство, либо только определенную его часть. Процедура преобразования виртуальных адресов в физические должна быть максимально прозрачна для пользователя и программиста.

Существуют два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические. В первом случае замена виртуальных адресов на физические выполняется один раз для каждого процесса во время начальной загрузки программы в память. Специальная системная программа -перемещающий загрузчик-на основании имеющихся у нее исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, а также информации, предоставленной транслятором об адресно-зависимых элементах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, то есть операнды инструкций, и адреса переходов имеют те значения, которые выработал транслятор. В наиболее простом случае, когда виртуальная и физическая память процесса представляют собой единые непрерывные области адресов, операционная система выполняет преобразование виртуальных адресов в физические по следующей схеме. При загрузке операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический.

Последний способ является более гибким: в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти, динамическое преобразование виртуальных адресов позволяет перемещать программный код процесса в течение всего периода его выполнения. Но использование перемещающего загрузчика более экономично, так как в этом случае преобразование каждого виртуального адреса происходит только один раз во время загрузки, а при динамическом преобразовании - при каждом обращении по данному адресу.

В СРВ, когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

Необходимо различать максимально возможное виртуальное адресное пространство процесса и назначенное (выделенное) процессу виртуальное адресное пространство. В первом случае речь идет о максимальном размере виртуального адресного пространства, определяемом архитектурой компьютера, на котором работает ОС, и, в частности, разрядностью его схем адресации (32-битная, 64-битная и т. п.). Например, при работе на компьютерах с 32-разряднымии процессорами Intel Pentium операционная система может предоставить каждому процессу виртуальное адресное пространство до 4 Гбайт (2^{32}). Однако это значение представляет собой только потенциально возможный размер виртуального адресного пространства, который редко на практике необходим процессу. Процесс использует только часть доступного ему виртуального адресного пространства.

Назначенное виртуальное адресное пространство представляет собой набор виртуальных адресов, действительно нужных процессу для работы. Эти адреса первоначально назначает программе транслятор на основании текста программы, когда создает кодовый (текстовый) сегмент, а также сегмент или сегменты данных, с которыми программа работает. Затем при создании процесса ОС фиксирует назначенное виртуальное адресное пространство в своих системных таблицах. В ходе своего выполнения процесс может увеличить размер первоначального назначенного ему виртуального адресного пространства, запросив у ОС создания дополнительных сегментов или увеличения размера существующих. В любом случае операционная система следит за корректностью использования процессом виртуальных адресов - процессу не разрешается оперировать с виртуальным адресом, выходящим за пределы назначенных ему сегментов.

Максимальный размер виртуального адресного пространства ограничивается только разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

Необходимо подчеркнуть, что виртуальное адресное пространство и виртуальная память - это различные механизмы, и они не обязательно реализуются в операционной системе одновременно. Можно представить себе ОС, в которой поддерживаются виртуальные адресные пространства для процессов, но отсутствует механизм виртуальной памяти. Это возможно только в том случае, если размер виртуального адресного пространства каждого процесса меньше объема физической памяти.

Содержимое назначенного процессу виртуального адресного пространства, то есть коды команд, исходные и промежуточные данные, а также результаты вычислений, представляет собой *образ процесса.*

Во время работы процесса постоянно выполняются переходы от прикладных кодов к кодам ОС, которые либо явно вызываются из прикладных процессов как системные функции, либо вызываются как реакция па внешние события или на исключительные ситуации, возникающие при некорректном поведении прикладных кодов. Для того чтобы упростить передачу управления от прикладного кода к коду ОС, а также для легкого доступа мо-

дулей ОС к прикладным данным (например, для вывода их на внешнее устройство), в большинстве ОС ее сегменты разделяют виртуальное адресное пространство с прикладными сегментами активного процесса. То есть, сегменты ОС и сегменты активного процесса, образуют единое виртуальное адресное пространство.

Обычно виртуальное адресное пространство процесса делится на две непрерывные части; системную и пользовательскую. В некоторых ОС (например, Windows NT, OS/2) эти части имеют одинаковый размер - по 2 Гбайт, хотя в принципе деление может быть и другим, например 1 Гбайт - для ОС, и 2 Гбайт - для прикладных программ. Часть виртуального адресного пространства каждого процесса, отводимая под сегменты ОС, является идентичной для всех процессов. Поэтому при смене активного процесса заменяется только вторая часть виртуального адресного пространства, содержащая его индивидуальные сегменты, как правило, - коды и данные прикладной программы. Архитектура современных процессоров отражает эту особенность структуры виртуального адресного пространства. Например, в процессорах Intel Pentium существует два типа системных таблиц: одна - для описания сегментов, общих для всех процессов, а другая - для описания индивидуальных сегментов данного процесса. При смене процесса первая таблица остается неизменной, а вторая заменяется новой.

Описанное выше назначение двух частей виртуального адресного пространства - для сегментов ОС и для сегментов прикладной программы - является типичным, но не абсолютным. Имеются и исключения из общего правила. В некоторых ОС существуют системные процессы, порожденные для решения внутренних задач ОС. В этих процессах отсутствуют сегменты прикладной программы части, обычно предназначенной для прикладных сегментов. И, наоборот, в общей, системной части виртуального адресного пространства размещаются сегменты прикладного кода, предназначенные для совместного использования несколькими прикладными процессами. Механизм страничной памяти в большинстве универсальных операционных систем применяется ко всем сегментам пользовательской части виртуального адресного пространства процесса. Исключения составляют ОС реального времени, в которых некоторые сегменты жестко фиксируются в оперативной памяти и соответственно никогда не выгружаются на диск - это обеспечивает быструю реакцию определенных приложений на внешние события. Системная часть виртуальной памяти в ОС любого типа включает область, подвергаемую страничному вытеснению, и область, на которую страничное вытеснение не распространяется. В не вытесняемой области размещаются модули ОС, требующие быстрой реакции и/или постоянного присутствия в памяти, например диспетчер потоков или код, который управляет заменой страниц памяти. Остальные модули ОС подвергаются страничному вытеснению, как и пользовательские сегменты. Обычно аппаратура накладывает свои ограничения на порядок использования виртуального адресного пространства. Некоторые процессоры (например, MIPS) предусматривают для определенной области системной части адресного

пространства особые правила отображения на физическую память. При этом виртуальный адрес прямо отображается на физический адрес (последний либо полностью соответствует виртуальному адресу, либо равен его части). Такая особая область памяти не подвергается страничному вытеснению, и поскольку достаточно трудоемкая процедура преобразования адресов исключается, то доступ к располагаемым здесь кодам и данным осуществляется очень быстро.

Методы управления, используемые в системах реального времени, обычно проще, чем в многопользовательских системах с разделением времени. В крупных вычислительных системах с множеством пользователей большинство программ и данных хранятся во вторичной (внешней) памяти - на жестком диске - и загружаются в оперативную память только при необходимости. Это приемлемо для систем разделения времени и пакетной обработки, в которых несущественно, начнется задание минутой раньше или позже. Однако в системах реального времени задержек исполнения быть не должно, поэтому все необходимые модули предварительно загружаются в оперативную память. Тем не менее, в системах реального времени может возникнуть необходимость в выгрузке содержимого части оперативной памяти на диск.

Работа виртуальной памяти основана на предположении, что объем памяти, требуемый для процессов, превосходит размер доступной оперативной памяти. Устройства массовой памяти, например жесткий диск, используемые для реализации этого механизма, должны обладать как достаточной емкостью, так и значительным быстродействием. Операционная система копирует с диска в оперативную память только те части процесса и области его данных, называемые страницами (pages), которые непосредственно используются в данный момент, оставляя остальную часть во внешней памяти. Для загрузки наиболее часто используемых страниц и для уменьшения числа обращений к диску применяются различные стратегии оптимизации. Механизм виртуальной памяти позволяет процессу иметь адресное пространство больше, чем размер выделенной ему реальной оперативной памяти. С другой стороны, применение виртуальной памяти существенно увеличивает накладные расходы и замедляет работу системы из-за многократных обращений к диску.

Применение виртуальной памяти в системах реального времени вызвано в основном экономическими причинами. Стоимость хранения единицы информации в оперативной памяти выше, чем во вторичной памяти. Еще одной важной причиной является надежность работы. В случае системного сбоя можно восстановить работу процесса. Если сбой или перерыв в электропитании происходит, когда вся система находится только в оперативной памяти, все процессы и их данные будут потеряны, и восстановить их будет невозможно.

В системах реального времени представляет интерес только быстрая и эффективная виртуальная память. Чтобы быстро реагировать на внешние сигналы, соответствующие служебные процедуры должны постоянно хра-

ниться в оперативной памяти. Другим важным соображением, относящимся к использованию вторичной памяти в задачах реального времени, является ее работоспособность в производственной среде - жесткие диски и дискеты нельзя использовать в условиях сильных вибраций, ударов или интенсивных магнитных полей.

Одно из существенных различий между многопользовательскими операционными системами и системами реального времени касается управления файлами. В многопользовательских системах наиболее важными проблемами является структура каталогов и защита файлов. Управление и защита каталогов с соответствующим контролем прав доступа при каждом обращении требуют таких накладных расходов, которые обычно неприемлемы для систем реального времени. Однако, как правило, в системах реального времени эти меры не нужны, поскольку дисковая память используется в основном для протоколов и отчетов, а все процессы принадлежат одному "пользователю". Поэтому применение сложных механизмов управления файлами в системах реального времени обычно не оправдано.

Наиболее сложные операционные системы для достижения оптимальных характеристик позволяют настраивать параметры управления процессором и памятью. Необходимо должным образом подобрать приоритеты процессов, продолжительность квантов времени, размер страницы виртуальной памяти и другие параметры операционной системы.

Лекция 2.5. Переходные процессы в логических схемах. Гонки.

1. Переходные процессы в логических схемах.
2. Гонки.
3. Гонки по входу.

1. Переходные процессы в логических схемах

Задержка логической схемы слагается из задержек срабатывания логических элементов и задержек распространения сигналов по цепям связи между ними. Трудоемкость учета задержек зависит от соотношения значений задержек самих логических элементов и задержек в цепях связи. Если эти значения близки, то задержки различных трактов схемы можно определить только после размещения элементов на поверхности платы или кристалла большой интегральной схемы (БИС), когда станут, известны фактические длины связей. Если при этом задержки некоторых цепей не соответствуют требуемым, то нужно или переставлять элементы, или вносить изменения в функциональную схему, снова трассировать связи и снова определять задержки в них. Процесс становится итерационным, длительным. Именно в таком положении оказываются разработчики аппаратуры на быстрых элементах ЭСЛ, устанавливаемых на платах в виде микросхем или изготавливаемых прямо на поверхности кристаллов БИС. Сложность учета задержек -одна из причин, препятствующих широкому распространению элементов ЭСЛ в схемах цифровой автоматики.

В цифровой автоматике в основном используются элементы с временем переключения не менее 20 нс, что примерно на порядок превышает

задержку распространения сигнала в любом проводе монтажной платы типового размера. Паразитная емкость монтажа при использовании типовых плат также не настолько велика, чтобы существенно изменить задержку элемента. В этих случаях задержку внутриплатного и близкого межплатного монтажа рационально не учитывать отдельно, а, оценив худший случай, включить ее в состав задержки логического элемента. Небольшая потеря потенциально достижимого быстродействия окупается упрощением разработки схем, поскольку задержки могут быть учтены без каких-либо итераций, сразу, и притом уже на этапе логического проектирования. Технические этапы проектирования - размещение элементов и трассировка связей - выполняются только один раз и не вызывают необходимости корректировать функциональные схемы. Будем предполагать, что задержки в цепях связи включены в состав задержек логических элементов.

Ситуации, когда задержки в связях превышают задержки в элементах, возникают и при использовании не очень быстродействующих элементов - когда сигналы передаются между блоками на достаточно большое расстояние. Однако доля подобных связей невелика, поэтому их можно выделить особо и учесть задержку в кабеле. Задержки различных экземпляров элементов какого-то определенного типа имеют технологический разброс, который обычно описывают некоторым статистическим законом. Задержка каждого конкретного элемента зависит от его температуры, длительности фронта входного сигнала, от того, насколько элементов, и притом каких он нагружен, от паразитной емкости монтажа, числа лет с момента выпуска и ряда других факторов. В паспортах элементов некоторых серий влияние части этих факторов учитывается дифференцированно в виде графиков, таблиц, линеаризованных зависимостей, но чаще это влияние просто оценивается по максимуму. При этом паспортные значения задержек и фронтов приводятся для худшего случая, который может встретиться при соблюдении указанных в паспорте ограничений. В первом случае удается полнее использовать возможности элемента, во втором - упрощается проектирование.

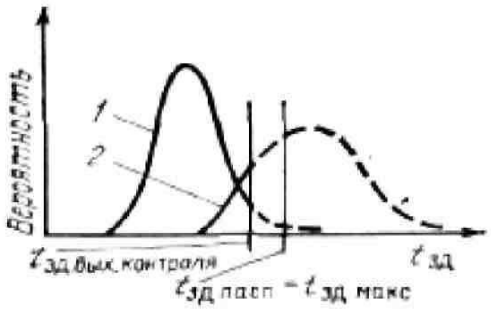


Рисунок 1. – Плотность вероятности распределения задержки элемента в условиях налаженного производства 1 и в период освоения 2

На рис. 1 показан возможный вид кривой технологического разброса задержки элементов при испытаниях на предприятии-изготовителе. Выходной контроль отсекает хвост кривой в соответствии с техническими

условиями (ТУ) на элемент с учетом необходимого запаса на старение, допуски и т. п. Если правильно налажены производство и контроль, то потребитель всегда имеет дело с элементами, задержка которых не превышает паспортную.

Однако, разработчику весьма полезно знать кроме максимальной и минимально возможную задержку. Однако, для большинства серийно выпускаемых микросхем значение минимальной задержки в ТУ не указано и, следовательно, изготовителем не гарантируется. Опыт работы схемотехника с данными элементами здесь бесполезен, поскольку кривая технологического разброса у разных изготовителей различна и к тому же чувствительна к перестройкам производства, что и иллюстрируют две кривые на рис.1. Поэтому если разработчик аппаратуры использует микросхемы, в паспорте которых не оговорено минимальное значение задержки, то он вынужден полагать минимальное время задержки равным нулю. Никаких юридических оснований считать, что это значение больше нуля, у него нет.

Уровень выхода элемента в течение отрезка времени от минимально возможного до максимально возможного значения задержки, когда фактическое состояние выхода элемента разработчику не известно, называют состоянием неопределенности и обозначают символом x. Состояние x, поступая на входы других логических элементов, может в зависимости от типа элемента порождать на их выходе или определенные состояния 1 или 0, или также неопределенное, обозначаемое X. Поведение логических элементов задается при этом законами уже не двоичной, а одного из видов троичной логики. Соотношения, расширяющие основные функции на третью переменную x, достаточно очевидны:

$$\begin{aligned} &\bar{x} = X;\ x \cdot 0 = 0; \quad x \cdot 1 = X; \quad x_1 \cdot x_2 = X; \\ &x \vee 0 = X; \quad x \vee 1 = 1; \quad x_1 \vee x_2 = X; \\ &x \oplus 0 = X; \quad x \oplus 1 = X; \quad x_1 \oplus x_2 = X, \end{aligned} \tag{1}$$

где x, x_1 x_2 - неопределенные значения сигналов на входах элементов.

Эффективным средством анализа переходных процессов в схемах являются временные диаграммы. При их построении состояние неопределенности изображают одним из двух способов, которые показаны для элемента И (рис. 2а). Изображение на рис. 2б строже, но менее наглядно; изображение на рис. 2в нагляднее, но может быть спутано с состоянием высокого импеданса элемента, имеющего три состояния выхода.

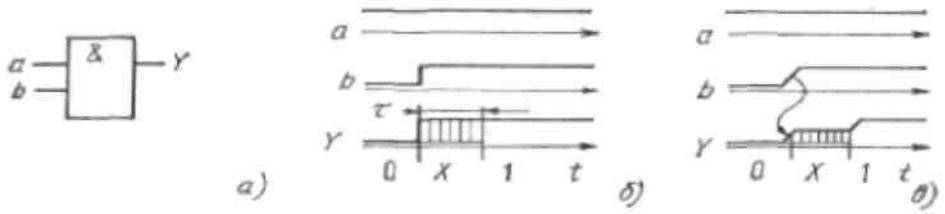


Рисунок 2.- Способы изображения состояния неопределенности логического элемента

Линии со стрелками обозначают причинно-следственные отношения в цепочке переключений. Линия начинается на фронте, который непосредственно вызывает переключение рассматриваемого элемента и оканчивается стрелкой на фронте выходного сигнала этого элемента. Наличие таких указателей заметно облегчает понимание работы сложных схем.

На рис. 3а показан фрагмент схемы и варианты начертания временных диаграмм переходных процессов. Здесь и в дальнейшем для обозначения выходного сигнала элемента используется номер самого элемента. Диаграмма на рис.3б игнорирует переходные процессы в элементах и схеме. Такие диаграммы применяют, когда основной целью является иллюстрация логических и причинно-следственных отношений, а длительностью переходных процессов по сравнению с интервалами между поступлением сигналов можно пренебречь.

Диаграмма на рис. 3а построена в предположении, что все элементы имеют максимально возможные значения задержки. Эта диаграмма наглядна, поэтому удобна для первого знакомства с поведением сложной схемы. Но она годится лишь для оценки максимальной длительности переходного процесса. Делать по такой диаграмме выводы о состояниях элементов во время переходного процесса нельзя: это лишь один частный случай из множества возможных процессов.

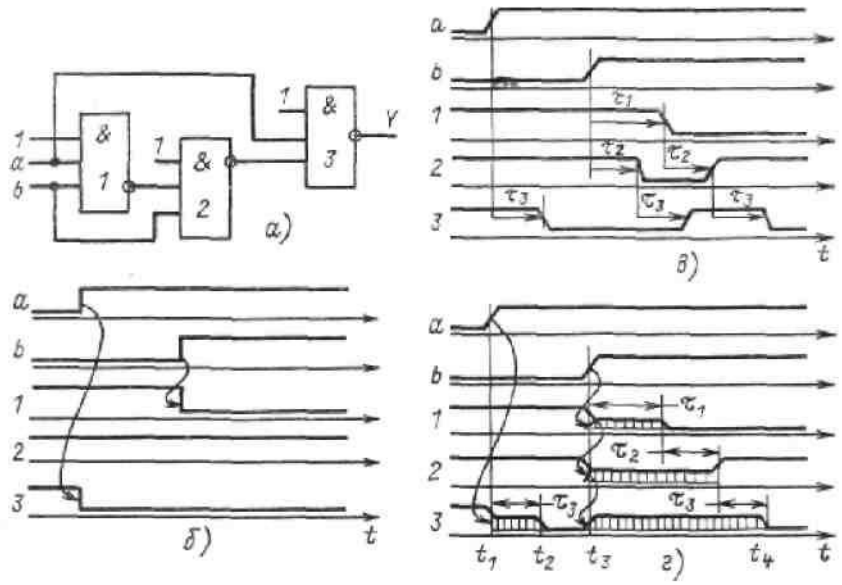


Рисунок 3. - Временные диаграммы переходных процессов:
а - фрагмент схемы; б, в, г - изображение переходных процессов; б - без учета задержек элементов, в - в предположении, что задержки максимальны; г - с использованием состояния неопределенности

Диаграмма рис. 3г учитывает состояния неопределенности элементов в соответствии с (1). Она достаточно строго моделирует поведение схемы при любых комбинациях задержек, допускаемых паспортами элементов. Полезно сравнить диаграммы на рис. 3 в и г, обращая внимание на их

расхождения, причиной которых является общность диаграммы г и частность диаграммы в.

Быстрое чтение и особенно построение временных диаграмм требуют некоторой тренировки. Полезно самостоятельно построить несколько вариантов диаграмм, изменяя моменты поступления входных сигналов и соотношения задержек элементов схемы на рис. 3а. Построение диаграммы нужно начинать с тех элементов, для которых известны все входные сигналы, в данном примере - с элемента 1. После определения выхода элемента 1 известными становятся все входы элемента 2 и т. д. Если построение диаграммы с учетом состояния неопределенности вызывает затруднение, можно рекомендовать сначала построить диаграмму с нулевыми задержками, показанной на рис. 3б, затем на том же чертеже наложить на нее диаграмму с максимальными задержками, после чего интервалы состояний неопределенности выделяются намного легче.

В организациях, специализирующихся на разработке логических схем, построение и анализ временных диаграмм выполняются на ЭВМ с помощью специальных моделирующих программ. Для выявления некоторых тонких случаев неопределенности используют не только троичные, но и пятеричные, и более сложные формы представления ситуаций во время переходных процессов. Введение состояния неопределенности позволяет выявить важный, хотя и не очевидный с первого взгляда эффект, который всегда нужно учитывать.

На рис. 4 показана цепочка из двух элементов, на вход которой поступает сигнал в виде единичного импульса длительностью Т. У выходного сигнала в его начале и конце будут зоны неопределенности длительностью по 2Т каждая. Если задержки включения и выключения равны и максимальны, то полученный сигнал Y2, будет сдвинут относительно выходного на 2Т. В результате может оказаться, что один и тот же сигнал, переданный по двум цепочкам на два блока устройства, запустит их не одновременно. Понятие одновременности расплывается и становится относительным.

Если задержки включения существенно отличаются от задержек выключения, получится укороченный на 2Т (Y3) или удлиненный на 2Т (Y4) сигнал. В случае Y4 укороченной окажется пауза между последовательными импульсами. Могут получиться и любые промежуточные формы рассмотренных частных случаев, причем предугадать характер эффекта заранее невозможно. Если цепочка содержит k элементов, то во всех рассмотренных случаях вместо двойки в качестве множителя при Т войдет k. У разработчика нет никаких официальных документов, позволяющих проигнорировать любой из возможных эффектов, и он вынужден проектировать схему так, чтобы ни один из них не привел к сбою в работе. Если на выходе цепочки требуется получить импульс с минимальной длительностью, то длительность импульса на входе цепочки должна быть на k% больше. Аналогично нужно обеспечивать на выходе цепочки и минимальную

длительность паузы, и максимальную длительность импульса, если это требуется.

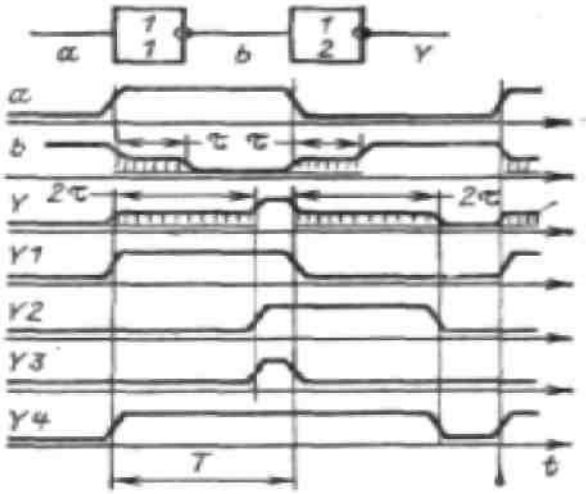


Рисунок 4. – Изменение задержки и длительности импульса при прохождении его по цепочке элементов

2. Гонки

В логических схемах встречаются участки, где сигнал разветвляется, получившиеся два сигнала распространяются по двум независимым цепочкам элементов, а затем оба сигнала снова встречаются на входах одного элемента. Подобная ситуация показана на рис. 5а, где в рассматриваемый момент времени в представленном фрагменте схемы два тракта оказались прозрачными для входного сигнала благодаря тому, что все конъюнкторы фрагмента в этот момент открыты сигналами единичного уровня. Пусть в тракте чет четное число инверторов, а в тракте нечет - нечетное. Анализ подобной схемы методами алгебры Буля без учета задержек даст на ее выходе 0 при любом значении входного сигнала (рис. 5б). Но реальные элементы имеют конечную задержку срабатывания, и если обозначить задержки в трактах чет и нечет через $T_{чет}$ и $T_{нечет}$, то в зависимости от соотношения этих величин получится один из процессов, изображенных на рис. 5 в и г. В обоих случаях в выходном сигнале появится помеха, не предусмотренная булевыми выражениями. Легко проверить, что замена последнего элемента И на элемент ИЛИ не ликвидирует помеху, а лишь проинвертирует ее и изменит момент появления.

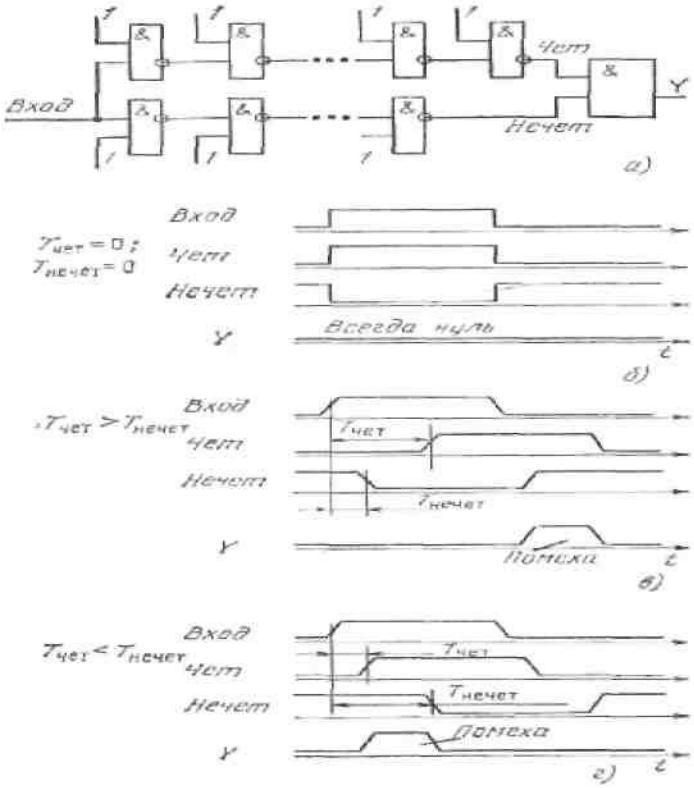


Рисунок 5. - Варианты временных диаграмм (б, в, г), иллюстрирующие гонки в логической схеме (а)

Существенно, что полученная помеха - это не пренебрежимо короткий всплеск напряжения малой амплитуды. При достаточно большой разности $T_{чет}$ и $T_{нечет}$ помеха будет иметь длительность, во много раз превышающую время переключения элемента, и амплитуду, равную номинальному сигналу. Это уже полноценный логический сигнал, на который могут реагировать последующие элементы. Если выход схемы подключен к запоминающему элементу (триггеру), то помеха может запомниться, и будет влиять на последующие процессы в устройстве. Если выход схемы подан в качестве обратной связи на вход, там появится непредвиденный сигнал, который может вызвать непредвиденное повторное срабатывание этой же схемы.

Описанное явление называют гонками или состязаниями (races). Два сигнала идут разными путями, и схема может реагировать на них по-разному (верно или неверно) в зависимости от того, какой сигнал выиграет гонку.

Основная проблема в том, что разработчик, как правило, не знает, в каком тракте задержка окажется меньше. Изготовитель элементов гарантирует лишь максимальное время задержки элемента данного типа и ничего не говорит ни о конкретной задержке конкретного элемента, ни о минимально возможном времени задержки. Поэтому разработчик логических схем не может воспользоваться тем фактом, что число элементов в цепочке чет, скажем, больше, чем в цепочке нечет. При массовом производстве схем из произвольно взятых элементов найдется достаточно большое число таких узлов, в которых в цепочку чет попадут более быстрые элементы, а в цепочку нечет - более медленные и, вопреки ожидаемому, будет выполняться

неравенство $T_{чет}<T_{нечет}$. Даже если в цепочке нечет один элемент, а в цепочке чет - два, то в последнюю вполне могут попасть элементы, имеющие время задержки втрое меньшее, чем элемент цепочки нечет.

Специальный подбор элементов по задержке в условиях современного автоматизированного массового производства недопустим, проверка реально получившегося соотношения задержек обычно неприемлема, так как сильно удорожает наладку аппаратуры. Кроме того, при изменении температуры и старении задержки различных элементов изменяются с разной скоростью, и по этому поводу изготовитель, как правило, никаких гарантий не дает. Единственное, что гарантирует изготовитель элементов и на что может опереться разработчик схем (фактически или хотя бы юридически), это то, что задержка не выйдет за пределы, указанные в ТУ на элемент, и если в борьбе с гонками разработчик хочет одержать победу, то он должен основывать свои расчеты лишь на этих данных.

Распространены три метода борьбы с гонками: введение тактирования, построение противогоночных схем и учет минимального времени задержки. Наиболее универсальным, эффективным и поэтому широко используемым методом борьбы с гонками является тактирование. Основная суть его заключается в следующем. По всему цифровому устройству разводится единая система тактирующих (синхронизирующих) сигналов. В широко распространенной двухтактной или двухфазной системе синхронизации используются две периодические последовательности синхросигналов - синхросигнал С1 и синхросигнал С2. Взаимное расположение этих сигналов во времени показано на рис. 6б.

Схема (рис. 6а) разделена штриховой линией на две части. Левая принимает и обрабатывает сигнал ВХОД: ее выходной сигнал Y1 является входным для схемы правой части, которая запоминает результат в триггере Тг.

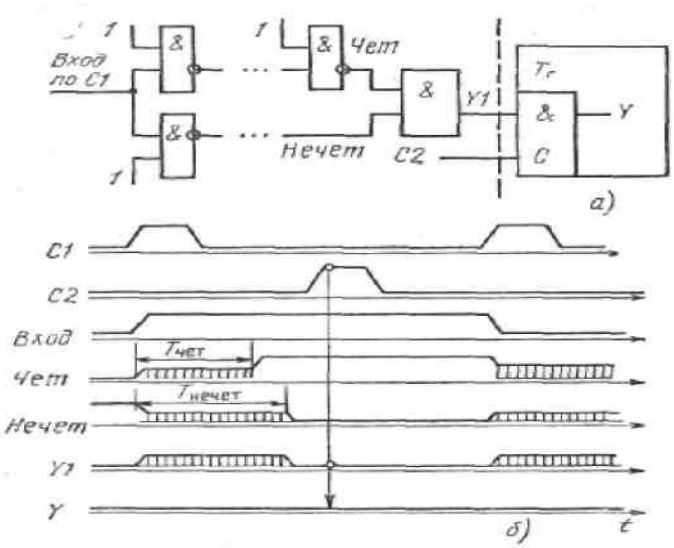


Рис. 5.6. Исключение помех, порожденных гонками, за счет тактирования

Если сигнал ВХОД каким-либо образом «привязан» к одной из синхросе-рий, например к С1, то этот сигнал будет изменяться только в момент поступления синхроимпульсов С1, а в промежутках между ними будет оставаться постоянным. Схема, показанная в левой части рис. 6а, имеет параллельные пути, в ней существуют гонки и возможно появление на выходе Y1 ложных сигналов. На рис. 6б эту ситуацию в общем виде отражают интервалы неопределенности трактов чет и нечет. На входной конъюнктор С правой части схемы подается сигнал син-хросерий С2. Обязательным условием является такой временной сдвиг С2 по отношению к С1, который превышает самый длинный интервал неопределенности, т. е. самую большую задержку из всех параллельных трактов схемы. Это значит, что сигнал С2 откроет конъюнктор С заведомо после окончания всех переходных процессов в схеме и пропустит логически правильное, не искаженное гонками установившееся значение функции Y1. Как правило, конъюнктор С вводят в состав триггера, что делает триггер синхронным. Такой триггер переключается только по команде синхросигнала и не воспринимает информацию при его отсутствии. Конъюнктор С применяют и без триггера - в тех случаях, когда запоминать выходной сигнал схемы не требуется, а нужно лишь очистить его от помех, порожденных гонками. Тогда сигнал, открывающий конъюнктор С, обычно называют не синхросигналом, а стробом, а сам процесс отсечки помех - стробированием.

Система синхронизации едина для всего цифрового устройства, и интервал между синхросигналами задается в начале разработки. Схемотехник, разрабатывая каждый фрагмент логической схемы, так подбирает число последовательно включаемых в нее элементов и их типы, чтобы все переходные процессы в этом фрагменте с гарантией закончились к моменту поступления очередного синхросигнала. Достоинство синхронизации как средства борьбы с гонками в том, что разработчику не требуется вникать в специфику протекания переходных процессов, в характер возникающих гоночных ситуаций, не нужно знать минимального значения задержки и т. д. Все, что должен знать разработчик - это максимально возможную задержку самого длинного тракта логической схемы, а это легко вычисляется по паспортным данным используемых элементов.

Противогоночные схемы - это схемы, построенные так, что в них если и возникают, то только неопасные гонки, то есть такие, при которых отсутствует риск появления на выходе сигналов, не предусмотренных логическим выражением. Примером неопасной гонки может служить гонка фронта по двум трактам, в каждом из которых содержится четное (или в каждом нечетное) число инверторов и которые объединяются на выходах элементом ИЛИ: кто бы ни выиграл гонку, результат все равно будет верный, изменится лишь задержка его получения. Для исключения опасных гонок можно вводить в схему дополнительные связи и элементы так, чтобы нежелательные параллельные пути запирались самим входным сигналом еще

до достижения им опасной развилки тракта. Есть целый ряд других приемов, используемых при построении противогоночных схем.

Полезным свойством противогоночных схем оказывается их способность обрабатывать данные по мере поступления, асинхронно, то есть без привязки к тактирующим сигналам и связанным с этим потерям времени. Однако процедура построения таких схем очень сложна, она требует скрупулезного изучения характера протекания переходных процессов, выявления всех возможных гоночных путей, отделения опасных состязании от неопасных. С ростом числа элементов схемы разработчик очень скоро оказывается в плену невыполнимого по объему перебора вариантов. Специалистами разработан ряд удачных, остроумных асинхронных противогоночных схем, в основном - сложных типов триггеров с числом элементов в пределах десятка. Они широко применяются, в том числе и в тактируемых устройствах, для быстрого выполнения операций внутри самого такта.

Учет минимального времени задержки. Если известно минимально возможное время задержки элемента, то во многих практически важных случаях можно постулировать отсутствие гонок. Пусть в схеме на рис. 5а глубина цепочки Чет настолько больше глубины цепочки Нечет, что задержка в длинной цепочке, даже если последняя состоит только из самых быстрых элементов, будет все равно больше задержки сигнала в короткой цепочке, даже если в нее попадут только самые медленные элементы. Схема со столь большой разницей в длине путей всегда будет вести себя так, как показано на рис. 5в. То есть, пока входной сигнал равен единице, помеха на выходе не появится. Помеху после выключения входного сигнала можно ликвидировать введением дополнительной блокирующей связи. Можно, например, взять в качестве выходного элемента трехвходовой элемент и на его третий вход подать сам входной сигнал. В этом случае выход будет заперт сразу после перехода входного сигнала в нуль. Несложно предложить и другие формы использования для борьбы с гонками сведений о минимально возможной задержке или о наибольшей возможной кратности максимального и минимального значений задержки.

По возможностям применения этого метода борьбы с гонками разработчики, использующие различную элементную базу, находятся в неодинаковых условиях, В лучшем положении обычно находятся разработчики схем, предназначенных для реализации на поверхности кристалла. Результирующая задержка элементов на кристалле определяется длиной и шириной проводников межэлементных связей и временем переключения транзисторов, которое в свою очередь зависит от геометрических размеров элементов его маски. На все эти размеры разработчик схемы кристалла в может влиять, что позволяет ему в случаях, когда это важно, делать задержку одной группы элементов гарантированно больше задержки другой группы. Правда, по мере роста числа элементов, размещаемых на одном кристалле, разработчик логических схем все реже допускается к воздействию на геометрические размеры элементов маски.

Технология автоматизированного проектирования схем на перспективных матричных кристаллах разрешает схемотехнику лишь соединять между собой проводниками стандартной ширины уже размещенные на кристалле стандартные транзисторы или логические элементы. Однако и здесь положение разработчика все-таки лучше, чем того, который использует отдельные микросхемы, поскольку задержки однотипных элементов, расположенных на одном кристалле, существенно между собой коррелируют, чего нельзя сказать о микросхемах даже одной закупочной партии, одного изготовителя. Эта корреляция позволяет изготовителю кристаллов после проведения соответствующих исследований постулировать для схемотехника максимально возможную кратность задержек элементов, расположенных на одном кристалле данного типа. С точки зрения борьбы с гонками это во многих случаях не хуже, чем знание минимально возможной задержки. Поэтому внутри интегральных схем метод борьбы с гонками за счет назначения параллельными путями таких соотношений задержек, при которых опасные гонки невозможны, используется, особенно при построении небольших узлов - типа триггеров, счетчиков и т. п.

В худшем положении находится разработчик, использующий готовые микросхемы, поскольку юридического документа о минимальном значении задержки он чаще всего не имеет. Правда, и тут опытный инженер может утверждать, что при использовании любой современной серии элементов и при любом их сочетании пробег сигнала по цепочке из, скажем, 64 элементов длится «наверняка» дольше, чем пробег сигнала по параллельной ветви из одного элемента. На сегодня нет серий, задержка элементов внутри которых отличалась бы в 64 раза. И в 32 раза тоже нет. И в 16, пожалуй, не найдется. Относительно восьми можно задуматься, в защиту четырех большинство специалистов серьезно спорить уже не станут, а отклонение времени задержки вдвое встретится в большинстве серий. Таким образом, если отсутствие гонок обосновывается большой кратностью числа элементов параллельных путей, то нужно отдавать себе отчет в том, что есть зоны явно допустимых решений (например, кратность 64) и зоны явно недопустимых (кратность 2), а граница между ними не определена. Каждый разработчик определяет ее для себя индивидуально в зависимости от опыта, соотношения поощрения за экономичную схему и наказания за сбои в ней из-за гонок. Проблема из технической становится психологической, организационной. В инженерной практике пользуются этим приемом и строят схемы, в которых в принципе, юридически, гонки возможны, но по утверждению разработчика их «наверняка» не будет.

В последние годы растет интерес к еще одному методу борьбы с гонками -построению самосинхронизирующихся схем. Рабочие узлы в этом случае строятся непротивогоночными, но они дополняются специальными схемами, которые обнаруживают факт окончания переходных процессов и вырабатывают разрешающий сигнал для следующих схем, играющий в каком-то смысле роль «асинхронного синхросигнала». Это направление рассматривается как весьма перспективное для построения БИС и особенно

сверхБИС, где применение обычной синхронизации встречает ряд трудностей. Однако в схемах и микросхемах обычного размера и технологии это направление пока не находит применения ввиду как сложности построения такого рода схем, так и, приблизительно, удвоения их аппаратурных затрат

Проблема гонок в цифровой схемотехнике является очень серьезной. Большинство трудно обнаруживаемых и удивительно разнообразно проявляющихся ошибок в функциональных схемах связано с гонками, возможности появления которых разработчик не заметил. Основная причина - ограниченность поля внимания человека. При разработке сложной схемы все внимание поглощается конструированию главного пути распространения сигнала, непосредственно решающего поставленную задачу. При этом побочные, не нужные для дела пути выпадают из поля зрения, а в них и «таится погибель».

Гонки во вновь разработанной схеме нужно искать специально. Если есть такая возможность, то наиболее надежным и простым методом является моделирование работы схемы с помощью специальных программ. При поиске гонок «вручную» сначала нужно выявить все подозрительные места и затем методически их исследовать. Обнаружению таких мест способствуют специально предпринимаемые просмотры схемы, нацеленные на выявление всех возможных параллельных путей распространения сигнала. Полезен анализ временных диаграмм, в которых зоны неопределенности указывают на возможность появления ложных сигналов.

В борьбе с гонками наряду с излагаемым подходом, ориентированным на гарантированную работу даже при наиболее неблагоприятном сочетании задержек, существуют менее строгие походы, в основе которых лежит утверждение, что худший случай встречается редко, поэтому для оценки задержки схемы можно брать некоторое вероятностное значение, меньшее, чем максимально возможное время задержки. Подобные подходы, против которых в принципе возражать нельзя, для цифровой техники оказываются плодотворными, только если есть возможность провести абсолютно полное тестирование готового устройства во всех возможных режимах и условиях окружающей среды.

Если же надежность работы устройства основывается лишь на статистической правильности работы отдельных его цепей, то, когда цепей много, что типично для цифровой техники, даже небольшое уменьшение надежности срабатывания элементов приводит к резкому снижению надежности всего устройства. Можно подсчитать, что если в каждой цепочке допустить вероятность помехи из-за гонок всего в I %, то вероятность работоспособности устройства, содержащего 100 таких цепочек, будет около 37 %. В среднем из каждых трех устройств, два будут неработоспособны. Поэтому приемами, приносящими в жертву надежность срабатывания отдельных цепочек, в цифровой технике пользоваться не следует.

3. Гонки по входу Гонки по входу возникают, когда ветвящийся сигнал поступает на элементы, имеющие разброс по уровню срабатывания

(рис. 7а), а фронт этого сигнала излишне пологий (рис. 7б). Если длительность фронта входного сигнала заметно больше времени срабатывания элементов, то где-то в середине фронта будет существовать отрезок времени, когда с точки зрения одного элемента входной сигнал уже равен 1, а с точки зрения другого - еще равен 0. Элементы будут реагировать на один и тот же сигнал как на два различных, а такая ситуация при проектировании схемы ее алгоритмом не предусматривается. В результате схема в течение этого времени может выработать ложные сигналы. Это явление и называют «гонки по входу». Гонки по входу не наблюдаются, если логическая схема собрана на элементах одной серии микросхем. Потенциально опасны схемы, собранные из элементов различных серий, имеющих одинаковый уровень сигналов, но существенно различные времена задержек и фронтов. Гонки по входу возникают в схемах некоторых БИС, если их межэлементные связи сильно заваливают фронты. Опасностью возникновения гонок по входу объясняются ограничения на максимальную длительность фронтов входных сигналов, приводимые в паспортах многих микросхем.

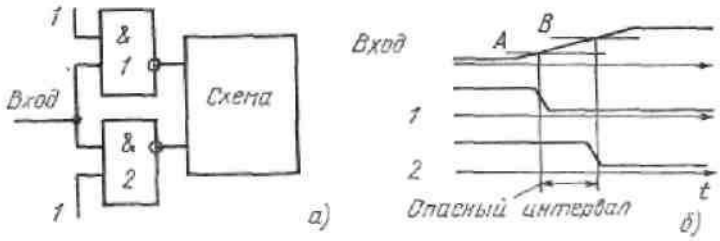


Рисунок 7. - Гонки по входу: иллюстрация условий их возникновения

Если нет возможности увеличить крутизну фронта, то единственным средством борьбы с гонками по входу остается тактирование, поскольку в тактированном устройстве выходной сигнал схемы не используется до тех пор, пока в этой схеме не окончатся абсолютно все переходные процессы независимо от их физической природы. Однако тактирование не спасает от гонок по самому тактирующему входу. Поэтому, если крутизна фронтов синхросигналов мала, то нужно применять такие синхронные триггеры, в которых гонки по входу не возникают.

ТЕМА 3. ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Лекция 3.1. Архитектура систем реального времени

1. Основные параметры и механизмы операционных систем реального времени.
2. Базовые концепции построения операционных систем реального времени.
3. Монолитная архитектура.
4. Модульная архитектура на основе микроядра.
5. Объектная архитектура на основе объектов – микроядер.

1. Основные параметры и механизмы операционных систем реального времени

Одно из коренных внешних отличий систем реального времени от систем общего назначения - четкое разграничение систем разработки и систем исполнения. Система исполнения операционных системах реального времени - набор инструментов (ядро, драйверы, исполняемые модули), обеспечивающих функционирование приложения реального времени.

Большинство современных ведущих операционных систем реального времени поддерживают целый спектр аппаратных архитектур, на которых работают системы исполнения (Intel, Motorola, RISC,MIPS, PowerPC, и другие). Это объясняется тем, что набор аппаратных средств является частью комплекса реального времени и аппаратура должна быть также адекватна решаемой задаче. Поэтому ведущие операционные системы реального времени перекрывают целый ряд наиболее популярных архитектур, чтобы удовлетворить самым разным требованиям по части аппаратуры. Систему исполнения операционных системах реального времени и компьютер, на котором она исполняется называют "целевой" (target) системой. Система разработки – это набор средств, обеспечивающих создание и отладку приложения реального времени.

Системы разработки работают, как правило, в популярных и распространенных ОС, таких, как UNIX. Кроме того, многие операционные системы реального времени имеют и так называемые резидентные средства разработки, исполняющиеся в среде самой операционной системы реального времени - особенно это относится к операционным системам реального времени класса "ядра".

Функционально средства разработки операционных систем реального времени отличаются от привычных систем разработки, таких, например, как Developers Studio, TaskBuilder, так как часто они содержат средства удаленной отладки, средства профилирования (измерение времен выполнения отдельных участков кода), средства эмуляции целевого процессора, специальные средства отладки взаимодействующих задач, а иногда и средства моделирования. Рассмотрим основные параметры операционных системах реального времени.

Время реакции системы. Практически все производители систем реального времени приводят такой параметр, как время реакции системы на прерывание (interrupt latency). Если главным для системы реального времени является ее способность вовремя отреагировать на внешние события, то такой параметр, как время реакции системы является ключевым.

В настоящее время нет общепринятых методологий измерения этого параметра, поэтому он является полем битвы маркетинговых служб производителей систем реального времени. Но уже появился проект сравнения операционных систем реального времени, который включает в себя, в том числе, и разработку методологии тестирования. Рассмотрим времена, которые необходимо знать для того, чтобы предсказать время реакции системы. События, происходящие на объекте, регистрируются датчиками, данные с

датчиков передаются в модули ввода-вывода (интерфейсы) системы. Модули ввода-вывода, получив информацию от датчиков и преобразовав ее, генерируют запрос на прерывание в управляющем компьютере, подавая ему тем самым сигнал о том, что на объекте произошло событие. Получив сигнал от модуля ввода-вывода, система должна запустить программу обработки этого события. Интервал времени - от события на объекте и до выполнения первой инструкции в программе обработки этого события и является временем реакции системы на события. Проектируя систему реального времени, разработчики должны уметь вычислять этот интервал и знать из чего он складывается.

Время выполнения цепочки действий - от события на объекте до генерации прерывания - не зависит от операционных систем реального времени и целиком определяется аппаратурой, а интервал времени - от возникновения запроса на прерывание и до выполнения первой инструкции обработчика определяется целиком свойствами операционной системы и архитектурой компьютера. Причем это время нужно уметь оценивать в худшей для системы ситуации, то есть в предположении, что процессор загружен, что в это время могут происходить другие прерывания, что система может выполнять какие-то действия, блокирующие прерывания.

Основанием для оценки времен реакции системы могут служить результаты тестирования с подробным описанием архитектуры целевой системы, в которой проводились измерения с точным указанием, какие промежутки времени измерялись. Некоторые производители операционных систем реального времени результаты такого тестирования предоставляют.

Время переключения контекста. В операционные системы реального времени заложен параллелизм, возможность одновременной обработки нескольких событий. Поэтому все операционные системы реального времени являются многозадачными (многопроцессными, многонитиевыми). Чтобы уметь оценивать накладные расходы системы при обработке параллельных событий, необходимо знать время, которое система затрачивает на передачу управления от процесса к процессу (от задачи к задаче, от нити к нити), то есть время переключения контекста.

Размеры системы. Для систем реального времени важным параметром является размер системы исполнения, а именно суммарный размер минимально необходимого для работы приложения системного набора (ядро, системные модули, драйверы и т. д.). С течением времени значение этого параметра уменьшается, тем не менее, он остается важным, и производители систем реального времени стремятся к тому, чтобы размеры ядра и обслуживающих модулей системы были невелики. Примеры: размер ядра операционной системы реального времени OS-9 на микропроцессорах MC68xxx - 22 Кб, VxWorks - 16 Кб.

Возможность исполнения системы из ПЗУ (ROM). Это свойство операционных систем реального времени - одно из базовых. Оно позволяет создавать компактные встроенные СРВ повышенной надёжности, с ограниченным энергопотреблением, без внешних накопителей.

Важным параметром при оценке операционных систем реального времени является набор инструментов, механизмов реального времени, предоставляемых системой.

Механизмы систем реального времени. Процесс проектирования конкретной системы реального времени начинается с тщательного изучения объекта. Разработчики проекта исследуют объект, изучают возможные события на нем, определяют критические сроки реакции системы на каждое событие и разрабатывают алгоритмы обработки этих событий. Затем следует процесс проектирования и разработки программных приложений.

У разработчиков СРВ введено понятие **«идеальная операционная система реального времени»**, в которой приложения реального времени разрабатываются на языке событий объекта. Такая система имеет свое название, хотя и существует только в теории. Называется она: "система, управляемая критическими сроками". Разработка приложений реального времени в этой системе сводится к описанию возможных событий на объекте. В каждом описателе события указывается два параметра:

временной интервал - критическое время обслуживания данного события;

адрес подпрограммы его обработки.

Всю дальнейшую заботу о том, чтобы подпрограмма обработки события стартовала до истечения критического интервала времени, берет на себя операционная система.

Но это теоретически возможный предел. В реальности разработчик должен перевести язык событий объекта в сценарий многозадачной работы приложений операционных систем реального времени, стараясь оптимально использовать предоставленные ему специальные механизмы и оценить времена реакций системы на внешние события при этом сценарии.

Рассмотрим механизмы, используемые в операционных системах реального времени, которые делают систему реального времени предсказуемой.

Система приоритетов и алгоритмы диспетчеризации. Базовыми инструментами разработки сценария работы системы являются система приоритетов процессов (задач) и алгоритмы планирования (диспетчеризации) операционных систем реального времени.

В многозадачных ОС общего назначения используются, как правило, различные модификации алгоритма круговой диспетчеризации, основанные на понятии непрерывного кванта времени ("time slice"), предоставляемого процессу для работы. Планировщик по истечении каждого кванта времени просматривает очередь активных процессов и принимает решение, кому передать управление, основываясь на приоритетах процессов (численных значениях, им присвоенных). Приоритеты могут быть фиксированными или меняться со временем. Это зависит от алгоритмов планирования в данной ОС. Но рано или поздно, процессорное время получат все процессы в системе.

Алгоритмы круговой диспетчеризации неприменимы в чистом виде в операционных системах реального времени. Основной недостаток - непрерывный квант времени ("time slice"), в течение которого процессором владеет только один процесс. Планировщики же операционных систем реального времени имеют возможность сменить процесс до истечения кванта времени, если в этом возникла необходимость. Один из возможных алгоритмов планирования при этом – алгоритм "приоритетный с вытеснением". Мир операционных систем реального времени отличается богатством различных алгоритмов планирования: динамические, приоритетные, монотонные, адаптивные и пр., цель же всегда преследуется одна - предоставить инструмент, позволяющий в нужный момент времени исполнять именно тот процесс, который необходим.

Механизмы межзадачного взаимодействия. Другой набор механизмов реального времени относится к средствам синхронизации процессов и передачи данных между ними. Для операционных систем реального времени характерна развитость этих механизмов. К таким механизмам относятся: семафоры, мьютексы, события, сигналы, средства для работы с разделяемой памятью, каналы данных (pipes), очереди сообщений. Многие из подобных механизмов используются и в ОС общего назначения, но их реализация в операционных системах реального времени имеет свои особенности. В ОСРВ - время исполнения системных вызовов почти не зависит от состояния системы, и в каждой ОСРВ есть, по крайней мере, один быстрый механизм передачи данных от процесса к процессу.

Средства для работы с таймерами. Такие инструменты, как средства работы с таймерами, необходимы для систем с жестким временным регламентом, поэтому развитость средств работы с таймерами - необходимый атрибут операционных систем реального времени. Эти средства позволяют:

измерять и задавать различные промежутки времени (от 1 мкс и выше);

генерировать прерывания по истечении временных интервалов;

создавать разовые и циклические будильники.

В лекции были рассмотрены только базовые, обязательные механизмы, использующиеся в ОСРВ. Кроме того, почти в каждой операционной системе реального времени присутствует целый набор дополнительных, специфических только для нее механизмов, касающихся системы ввода-вывода, управления прерываниями, работы с памятью. Каждая система содержит также ряд средств, обеспечивающих ее надежность: встроенные механизмы контроля целостности кодов, инструменты для работы с таймерами.

2. Базовые концепции построения операционных систем реального времени

При описании операционной системы часто указываются особенности ее структурной организации и основные концепции, положенные в ее основу.

К базовым концепциям относятся:

Способы построения ядра системы - **монолитное ядро** или **микроядерный подход**. Большинство ОС использует монолитное ядро, которое компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский, и наоборот. Альтернативой является построение ОС на базе микроядра, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС - серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским, но система получается более гибкой - ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

Построение ОС на базе объектно-ориентированного подхода дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри операционной системы. Ее основными достоинствами являются: аккумуляция удачных решений в форме стандартных объектов; возможность создания новых объектов на базе имеющихся, с помощью механизма наследования; хорошая защита данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне; структурированность системы, состоящей из набора хорошо определенных объектов.

Наличие нескольких прикладных сред дает возможность в рамках одной ОС одновременно выполнять приложения, разработанные для нескольких ОС. Многие современные операционные системы поддерживают одновременно прикладные среды MS-DOS, Windows, UNIX (POSIX), OS/2 или хотя бы некоторого подмножества из этого популярного набора. Концепция множественных прикладных сред наиболее просто реализуется в ОС на базе микроядра, над которым работают различные серверы, часть которых реализуют прикладную среду той или иной операционной системы.

Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах. В распределенной ОС реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера. Характерными признаками распределенной организации ОС являются:

наличие единой справочной службы разделяемых ресурсов;

единой службы времени;

использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам;

многонитевой обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети;

наличие других распределенных служб.

3. Монолитная архитектура

Организация монолитной системы представляет собой структуру, у которой ОС написана в виде набора процедур, каждая из которых может вызывать другие, когда ей это нужно. При использовании такой техники каждая процедура системы имеет строго определенный интерфейс в терминах параметров и результатов, и каждая имеет возможность вызвать любую другую для выполнения необходимой для нее работы.

Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их в единый объектный файл с помощью компоновщика. Здесь, по существу, полностью отсутствует сокрытие деталей реализации - каждая процедура видит любую другую процедуру (в отличие от структуры, содержащей модули, в которой большая часть информации является локальной для модуля и процедуры модуля можно вызвать только через специально определенные точки входа).

Однако даже такие монолитные системы могут иметь некоторую структуру. При обращении к системным вызовам, поддерживаемым операционной системой, параметры помещаются в строго определенные места - регистры или стек, после чего выполняется специальная команда прерывания, известная как вызов ядра или вызов супервизора. Эта команда переключает машину из режима пользователя в режим ядра и передает управление операционной системе. Затем операционная система проверяет параметры вызова, чтобы определить, какой системный вызов должен быть выполнен. После этого операционная система обращается к таблице как к массиву с номером системного вызова в качестве индекса. В k-м элементе таблицы содержится ссылка на процедуру обработки системного вызова. Такая организация операционной системы предполагает следующую структуру:

1. Главная программа, которая вызывает требуемую служебную процедуру.

2. Набор служебных процедур, выполняющих системные вызовы.

3. Набор утилит, обслуживающих служебные процедуры. В этой модели для каждого системного вызова имеется одна служебная процедура. Утилиты выполняют функции, которые нужны нескольким служебным процедурам. Деление процедур на три уровня показано на рис. 1.

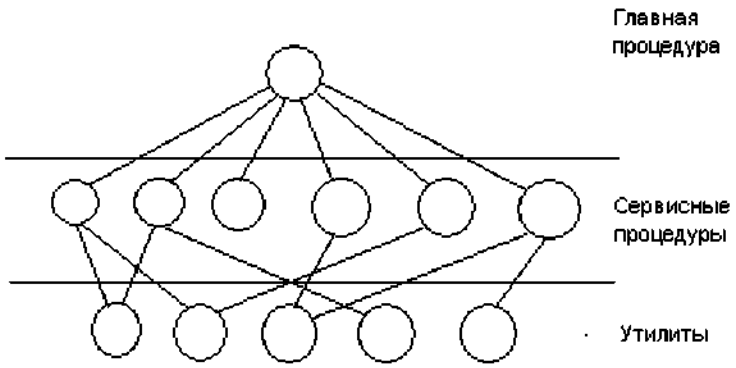


Рисунок 1. – Модель монолитной системы

В силу преимуществ объектно-ориентированного подхода приложения создаются на его основе, используя один из языков программирования, наилучшим образом поддерживающий этот подход. Архитектуры же классических операционных систем реального времени основаны на архитектурах UNIX систем и используют традиционный процедурный подход к программированию. Сочетание объектно-ориентированных приложений и процедурных операционных систем имеет ряд недостатков:

1. Происходит разрыв парадигмы программирования: в едином работающем комплексе (приложение + ОСРВ) разные компоненты используют разные подходы к разработке программного обеспечения.

2. Не используются все возможности объектно-ориентированного подхода.

3. Возникают некоторые потери производительности из-за разного типа интерфейсов в ОСРВ и приложении.

Естественно, возникает идея строить саму СРВ, используя объектно-ориентированный подход. При этом:

- как приложение, так и операционная система полностью объектно-ориентированны и используют все преимущества этого подхода;

- приложение и ОСРВ могут быть полностью интегрированы, поскольку используют один объектно-ориентированный язык программирования;

- обеспечивается согласование интерфейсов ОСРВ и приложения;

- приложение может «моделировать» ОСРВ для своих потребностей, заказывая нужные ему объекты;

- единый комплекс (приложение + ОСРВ) является модульным и легко модернизируемым. Идея реализована в ОСРВ SoftKernel, целиком написанной на C++. ОСРВ с монолитной архитектурой можно представить в виде:

- прикладного уровня: состоящего из работающих прикладных процессов;

- системного уровня: состоящего из монолитного ядра операционной системы, в котором можно выделить следующие части:

а) интерфейс между приложениями и ядром (API);

б) собственно ядро системы;

в) интерфейс между ядром и оборудованием (драйверы устройств). API в таких системах играет двойную роль:

1) управляет взаимодействием прикладных процессов и системы;

2) обеспечивает непрерывность выполнения кода системы (отсутствие переключения задач во время исполнения кода системы).

Основным преимуществом монолитной архитектуры является ее относительная быстрота работы по сравнению с другими архитектурами. Однако, достигается это, в основном, за счет написания значительных частей системы на ассемблере.

Недостатки монолитной архитектуры:

1. Системные вызовы, требующие переключения уровней привилегий (от пользовательской задачи к ядру), должны быть реализованы как прерывания или ловушки (специальный тип исключений). Это значительно увеличивает время их работы.

2. Ядро не может быть прервано пользовательской задачей. Это может приводить к тому, что высокоприоритетная задача может не получить управления из-за работы низкоприоритетной задачи. Например, низкоприоритетная задача запросила выделение памяти, сделала системный вызов, до окончания которого сигнал активизации высокоприоритетной задачи не сможет ее активизировать.

3. Сложность переноса на новые архитектуры процессора из-за значительных ассемблерных вставок.

4. Негибкость и сложность развития: изменение части ядра системы требует его полной перекомпиляции.

4. Модульная архитектура на основе микроядра

Модульная архитектура появилась, как попытка убрать узкое место API и облегчить модернизацию системы и перенос ее на новые процессоры.

API в модульной архитектуре играет только одну роль: обеспечивает связь прикладных процессов и специального модуля менеджера процессов. Однако теперь микроядро играет двойную роль:

1) управление взаимодействием частей системы (например, менеджеров процессов и файлов);

2) обеспечение непрерывности выполнения кода системы (отсутствие переключения задач во время исполнения микроядра).

Недостатки модульной архитектуры фактически те же, что и у монолитной архитектуры. Проблемы перешли с уровня API на уровень микроядра. Системный интерфейс по-прежнему не допускает переключения задач во время работы микроядра, только сократилось время пребывания в этом состоянии. API по-прежнему может быть реализован только на ассемблере, проблемы с переносимостью микроядра уменьшились (в связи с сокращением его размера).

5. Объектная архитектура на основе объектов-микроядер

В этой архитектуре (используемой в ОСРВ SoftKernel) API отсутствует вообще. Взаимодействие между компонентами системы

(микроядрами) и пользовательскими процессами осуществляется посредством обычного вызова функций, поскольку и система, и приложения написаны на одном языке (C++). Это обеспечивает максимальную скорость системных вызовов.

Фактическое равноправие всех компонент системы обеспечивает возможность переключения задач в любое время, то есть система полностью управляема.

Объектно-ориентированный подход обеспечивает модульность, безопасность, легкость модернизации и повторного использования кода.

Роль API играет компилятор и динамический редактор объектных связей (linker). При старте приложения динамический linker загружает нужные ему микроядра (в отличие от предыдущих систем, не все компоненты самой операционной системы должны быть загружены в оперативную память). Если микроядро уже загружено для другого приложения, то оно повторно не загружается, а используется код и данные уже имеющегося микроядра. Это позволяет сократить объем требуемой памяти.

Поскольку разные приложения разделяют одни микроядра, то они должны работать в одном адресном пространстве. Следовательно, система не может использовать виртуальную память и тем самым работает быстрее (так как исключаются задержки на трансляцию виртуального адреса в физический).

Поскольку все приложения и сами микроядра работают в одном адресном пространстве, то они загружаются в память, начиная с неизвестного на момент компиляции адреса. Следовательно, приложения и микроядра не должны зависеть от начального адреса (как по коду, так и по данным). Это свойство автоматически обеспечивает возможность записи приложений и модулей в ПЗУ, с последующим их исполнением, как в самом ПЗУ, так и в оперативной памяти.

Микроядра по своим характеристикам напоминают структуры, используемые в других операционных системах, однако есть и свои различия.

Микроядра и модули. Многие ОС поддерживают динамическую загрузку компонент системы, называемых модулями. Однако модули не поддерживают объектно-ориентированный подход ввиду того, что микроядро является фактически представителем некоторого класса. Далее, обмен информацией с модулями происходит посредством системных вызовов, что достаточно дорого.

Микроядра и драйверы. Многие ОС поддерживают возможность своего расширения посредством драйверов (специальных модулей, обычно служащих для поддержки оборудования). Однако драйверы часто должны быть статически связаны с ядром (образовывать с ним связанный загрузочный образ еще до загрузки) и должны работать в привилегированном режиме. Как и модули, они не поддерживают объектно-ориентированный подход и доступны приложениям только посредством системных вызовов.

Микроядра и DLL (Dynamically Linked Libraries, динамически связываемые библиотеки). Многие системы оформляют библиотеки, из которых берутся функции при динамическом связывании, в виде специальных модулей, называемых DLL. DLL обеспечивает разделение своего кода и данных для всех работающих приложений, в то время, как для микроядер можно управлять доступом для каждого конкретного приложения. DLL не поддерживает объектно-ориентированный подход, код DLL не является позиционно-независимым, и потому не может быть записан в ПЗУ.

Лекция 3.2 Механизмы синхронизации и взаимодействия процессов

1. Синхронизация процессов в системах реального времени.
2. Критические секции.
3. Семафоры.
4. События.

1. Синхронизация процессов в системах реального времени Организация некоторого порядка исполнения процессов называется **синхронизацией** (synchronization). Синхронизация процессов является основной функцией многозадачных операционных систем и используется для защиты ресурсов - с помощью механизма синхронизации упорядочивается доступ к ресурсу. То есть, процесс может получить доступ к ресурсу только после того, как другой процесс освободил его. Введение дополнительных переменных для защиты ресурсов не лучший выход, поскольку эти переменные сами становятся общим ресурсом. Существо проблемы состоит в том, что операции проверки и изменения значения переменной защиты разделены во времени и могут быть прерваны в любой момент. Более того, непрерывный контроль значений этих переменных представляет собой излишние затраты процессорного времени.

Существует достаточно обширный класс средств операционных систем, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Ситуации, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются **гонками.**

Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия.

Выполнение потока в мультипрограммной среде всегда имеет асинхронный характер. Очень сложно с полной определенностью сказать, на каком этапе выполнения будет находиться процесс в определенный момент

времени. Даже в однопрограммном режиме не всегда можно точно оценить время выполнения задачи. Это время во многих случаях существенно зависит от значения исходных данных, которые влияют на количество циклов, направления разветвления программы, время выполнения операций ввода-вывода и т. п. Так как исходные данные в разные моменты запуска задачи могут быть разными, то и время выполнения отдельных этапов и задачи в целом является весьма неопределенной величиной.

Еще более неопределенным является время выполнения программы в мультипрограммной системе. Моменты прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения - все эти события являются результатом стечения многих обстоятельств и могут быть интерпретированы как случайные. В лучшем случае можно оценить вероятностные характеристики вычислительного процесса, например вероятность его завершения за данный период времени.

Таким образом, потоки в общем случае (когда программист не предпринял специальных мер по их синхронизации) протекают независимо, асинхронно друг другу, Это справедливо как по отношению к потокам одного процесса, выполняющим общий программный код, так и по отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Любое взаимодействие процессов или потоков связано с их синхронизацией, которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными. Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он должен быть приостановлен.

При совместном использовании аппаратных ресурсов синхронизация также совершенно необходима. Когда, например, активному потоку требуется доступ к последовательному порту, а с этим портом в монопольном режиме работает другой поток, находящийся в данный момент в состоянии ожидания, то операционная система (ОС) приостанавливает активный поток и не активизирует его до тех пор, пока нужный ему порт не освободится. Часто нужна также синхронизация с событиями, внешними по отношению к вычислительной системе, например реакции на нажатие комбинации клавиш Ctrl+C.

Ежесекундно в системе происходят сотни событий, связанных с распределением и освобождением ресурсов, и операционная система должна иметь надежные и производительные средства, которые бы позволяли ей синхронизировать потоки с происходящими в системе событиями.

Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы. Например, два потока одного

прикладного процесса могут координировать свою работу с помощью доступной для них обоих глобальной логической переменной, которая устанавливается в единицу при осуществлении некоторого события, например выработки одним потоком данных, нужных для продолжения работы другого. Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые операционной системой в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества операционной системы они не могут приостановить друг друга или оповестить о произошедшем событии. Средства синхронизации используются операционной системой не только для синхронизации прикладных процессов, но и для ее внутренних нужд.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, быть функционально специализированными. Например, средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т. д. Часто функциональные возможности разных системных вызовов синхронизации перекрываются, так что для решения одной задачи программист может воспользоваться несколькими вызовами в зависимости от своих личных предпочтений.

Пренебрежение вопросами синхронизации в многопоточной системе может привести к неправильному решению задачи или даже к краху системы.

2. Критические секции

Важным понятием синхронизации потоков является понятие «критической секции» программы. **Критическая секция** - это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным **критическим данным,** при несогласованном изменении которых могут возникнуть нежелательные эффекты. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция. В разных потоках критическая секция состоит в общем случае из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток. При этом неважно, находится этот поток в активном или в приостановленном состоянии. Этот прием называют **взаимным исключением.** Операционная система использует разные способы реализации взаимного исключения. Некоторые способы пригодны для взаимного исключения при вхождении в критическую секцию только

потоков одного процесса, в то время как другие могут обеспечить взаимное исключение и для потоков разных процессов.

Самый простой и в то же время самый неэффективный способ обеспечения взаимного исключения состоит в том, что операционная система позволяет потоку запрещать любые прерывания на время его нахождения в критической секции. Однако этот способ практически не применяется, так как опасно доверять управление системой пользовательскому потоку - он может надолго занять процессор, а при крахе потока в критической секции крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Блокирующие переменные. Для синхронизации потоков одного процесса прикладной программист может использовать глобальные блокирующие переменные. С этими переменными, к которым все потоки процесса имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС.

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает.

Блокирующие переменные могут использоваться не только при доступе к разделяемым данным, но и при доступе к разделяемым ресурсам любого вида.

Если все потоки написаны с учетом вышеописанных соглашений, то взаимное исключение гарантируется. При этом потоки могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Однако следует заметить, что одно ограничение на прерывания все же имеется. Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Поясним это. Пусть в результате проверки переменной поток определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой поток занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому потоку, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен принцип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе команд многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной (например, команды BTC, BTK и BT5 процессора Pentium). При отсутствии такой команды в процессоре соответствующие действия должны реализовываться специальными системными примитивами (примитив - базовая функция ОС), которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация взаимного исключения описанным выше способом имеет существенный недостаток: в течение времени, когда один поток находится в

критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы для работы с критическими секциями.

3. Семафоры

Семафоры (semaphore) - это основной метод синхронизации. Он, в сущности, является наиболее общим методом синхронизации процессов.

В классическом определении семафор представляет собой целую переменную, значение которой больше нуля, то есть просто счетчик. Обычно семафор инициализируется в начале программы 0 или 1. Семафоры, которые могут принимать лишь значения 0 и 1, называются двоичными. Над семафорами определены две операции - **signal** и **wait.** Операция **signal** увеличивает значение семафора на 1, а вызвавший ее процесс продолжает свою работу. Операция wait приводит к различным результатам, в зависимости от текущего значения семафора. Если его значение больше 0, оно уменьшается на 1, и процесс, вызвавший операцию **wait,** может продолжаться. Если семафор имеет значение 0, то процесс, вызвавший операцию **wait,** приостанавливается (ставится в очередь к семафору) до тех пор, пока значение соответствующего семафора не увеличится другим процессом с помощью операции **signal.** Только после этого операция **wait** приостановленного процесса завершается (с уменьшением значения семафора), а приостановленный процесс продолжается.

Важно, что проверка и уменьшение значения семафора в операции **wait** выполняются за один шаг. Операционная система не может прервать выполнение операции **wait** между проверкой и уменьшением значения. Операция wait для семафора имеет такое же функциональное значение, что и инструкция **test_and_set.**

Если несколько процессов ждут одного и того же семафора, то после выполнения операции **signal** только один из них может продолжить свое развитие. В зависимости от реализации процессы могут ждать в очереди, упорядоченной либо по принципу FIFO (Firstln, FirstOut - первым вошел, первым вышел), либо в соответствии с приоритетами, или выбираться случайным образом.

Названия управляющей структуры "семафор" и операций **signal** и **wait** имеют очевидное мнемоническое значение. В литературе вместо **signal** и **wait** применяются и другие названия с тем же самым функциональным смыслом.

С помощью семафоров проблема защиты ресурсов решается следующим образом:

```
program sem_example (* защита ресурса *)
var P1: semaphore
begin
P1 := 1;
cobegin
```

```
while true do (* бесконечный цикл *)
begin (* процесс A *)
wait(P1);
(* защищенный ресурс *)
signal(P1);
…
end; (* процесс A *)
while true do (* бесконечный цикл *)
begin (* процесс B *)
wait(Pl);
(* защищенный ресурс *)
signa(Pl);
…
end; (* процесс B *)
coend;
end. (* sem_example *)
```

Семафор гарантирует, что два процесса могут получить доступ к защищенному ресурсу только по очереди. При этом не создается никаких дополнительных связей - если один процесс исполняется быстрее другого, то за определенный промежуток времени он будет чаще получать доступ к ресурсу. Процесс вынужден ждать окончания другого только в том случае, когда последний находится в критической секции. Одновременно гарантируется и живучесть. Если исполнение процесса по каким-либо причинам прекращается, то, при условии, что он находился вне критической секции, это не мешает развитию другого процесса.

Само по себе применение семафоров не гарантирует предотвращения тупиковых ситуаций. Если два процесса используют семафоры следующим образом

```
wait(Pl) wait(P2)
wait(P2) wait(Pl)
… …
(* защищенный ресурс *) (* защищенный ресурс *)
… …
signal(Pl) signa(P2)
signal(P2) signal(Pl)
```

то по-прежнему существует риск возникновения тупика. Если переключение процессов происходит между двумя операторами **wait** первой программы, а вторая программа выполнит свои операторы **wait**, то это приводит к тупику, поскольку каждая программа ожидает от другой освобождения семафора. Проблема состоит в том, что, хотя семафор гарантирует неразрывность проверки и установки значения, он сам остается защищенным ресурсом. В приведенном примере явно нарушен запрет последовательного выделения, и это приводит к возможности тупиковых ситуаций.

Семафор может помочь при синхронизации взаимосвязанных действий. Например, если процесс должен работать с данными только после

того, как они считаны с внешнего порта, программа может иметь следующий вид:

Process "Чтение данных" Process "Обработка данных"
while true do while true do
begin begin
(* чтение новых данных *) **wait**(data_available);
signal(data_available); (*обработка данных *)
end; end;

Это решение отделяет операцию ввода данных от их обработки. На появление новых данных указывает значение семафора, отличное от 0. Если существует механизм буферизации (промежуточного хранения) новых данных, то процедура обработки сможет получить все данные, даже если они поступают быстрее, чем она в состоянии их принять. В системах реального времени принято отделять процедуры, требующие быстрой реакции, например прием данных с внешнего порта, от других процессов.

Для защиты критических секций, в которые по определению в любой момент времени может входить только один процесс, используются двоичные семафоры, также называемые **mutex** (от mutual exclusion - взаимное исключение). В этом случае нельзя использовать обычные семафоры, так как их значение может превышать 1 и, следовательно, несколько программ могут получить доступ к ресурсу, уменьшая значения семафора. Операция **signal** над двоичным семафором всегда устанавливает его значение в 1. Операция **wait** уменьшает это значение с 1 до 0 и разрешает процессу продолжаться дальше. Если семафор имеет значение 0, то процесс, выполняющий **wait**, должен ждать до тех пор, пока значение семафора не изменится.

Ошибки синхронизации, связанные с неправильным использованием семафоров, трудно выявляются. Процесс, не выполняющий операцию **wait**, может войти в критическую секцию одновременно с другим процессом, что приведет к непредсказуемым результатам. Естественно, нельзя говорить, что такая ошибка выявится при тестировании; она даже может никогда не произойти за все время существования системы. Легче найти противоположную ошибку - отсутствующая операция **signal** может в определенный момент привести к остановке, по крайней мере, одного из процессов, что достаточно просто обнаружить.

Компилятор не имеет возможности проверить, правильно ли используются семафоры, то есть, согласованы ли операции **wait** с операциями **signal** в других модулях и связаны ли семафоры с соответствующими ресурсами, поскольку это зависит от логики алгоритма. Более того, размещение семафоров в программе, как и других команд, произвольно. Забота о проверке правильности программы лежит на программисте. Использование структурного программирования существенно облегчает решение этой задачи.

Семафоры являются удобным средством высокого уровня для замещения операции **test_and_set** и помогают избежать циклов занятого ожидания.

Однако их неправильное использование может привести к ситуации гонок и к тупикам.

4. События

В некоторых случаях несколько процессов, имеющих доступ к общим данным, должны работать с ними только при выполнении некоторых условий, необязательно связанных с данными и разных для каждого процесса. Условием, например, может быть поступление новых данных на входной порт. Все процессы имеют следующую структуру

begin
wait until condition;
modify data; **end**

Программа делится на две основные части. Сначала проверяются условия, а затем производятся операции над данными. Процедура проверки условия не изменяет данных и поэтому не требует какой-либо специальной защиты доступа. Однако доступ к данным для модификации должен быть координирован между процессами.

Если использовать семафоры, то их потребуется два: один - для контроля доступа в защищенную область с данными, а другой - для индикации изменения общих данных и, соответственно, необходимости повторной проверки условий.

Применение первого семафора просто, а для второго необходимо следить за числом ожидающих процессов и обеспечить, что при изменении условия ожидающие процессы будут активизированы с целью его проверки, то есть генерацию сигналов семафора, число которых равно числу ожидающих процессов. Это решение неудовлетворительно из-за большого расхода машинного времени на многочисленные проверки, при этом в программе довольно легко ошибиться.

Для решения этой проблемы была введена новая переменная синхронизации **event** (событие), с которой связаны операции **await** (ждать) и **cause** (вызвать). Процесс, выполнивший операцию **await** (event), остается в состоянии ожидания, пока значение переменной event не изменится. Это изменение контролируется с помощью операции **cause**. При наступлении события, то есть выполнении операции **cause** (event), освобождаются все ожидающие его процессы, в то время как в случае семафора освобождается лишь один процесс. Операции с событиями можно реализовать либо с помощью двоичной переменной, либо с помощью счетчика, при этом основные принципы остаются одинаковыми.

Глагол to await имеет значение не только "ждать", но и "предстоять", то есть конструкцию await (A) можно трактовать, как "предстоит событие A". Глагол to cause означает "быть причиной", побудительным мотивом, "вызвать что-либо". Конструкция cause (A) интерпретируется как "вызвать событие A" (в литературе и в операционных системах иногда используются и другие названия).

В противоположность семафору, переменную события нельзя использовать для защиты ресурса от конкурирующего доступа нескольких процес-

сов, поскольку по определению она освобождает все ожидающие процессы. Вышеприведенная проблема решается с помощью переменной события и семафора, если все программы имеют следующий вид

var mutex: **semaphore**; change: **event**;
begin
while not condition **do await**(change); **wait**(mutex);
(* обработка общих переменных *) **signal**(mutex); **cause**(change);
end;

При каждом изменении переменной event все процессы проверяют condition, и только те из них, для которых condition выполнено, могут продолжаться. Доступ к общему ресурсу защищен с помощью семафора mutex, при этом продолжается только один процесс. Это решение проще, чем основанное только на семафорах. Оно также более эффективно, поскольку процессы проверяют условия только тогда, когда это имеет смысл, т.е. после изменения значения соответствующих переменных.

Важный тип события в системах реального времени связан с внешними прерываниями. Программа обработки - обработчик прерываний - ждет прерывания. Когда оно происходит, исполнение обработчика возобновляется.

Лекция 3.3. Механизмы защиты ресурсов

1. Взаимные исключения.
2. Предотвращение тупиков.
3. Синхронизирующие объекты операционных систем.
4. Сигналы.

1. Взаимные исключения

Запрет прерываний может носить только исключительный характер. Другой подход к защите ресурсов основан на взаимном исключении (mutual exclusion). Никакой процесс не может получить доступ к ресурсу, пока этот ресурс не будет явно освобожден процессом, который захватил его первым.

Корректная защита ресурсов предполагает следующее:

1. В любой момент времени доступ к защищенному ресурсу имеет только один процесс.

2. Процессы остаются взаимно независимыми. Остановка одного из процессов не должна препятствовать продолжению других.

Сформулированные требования соответствуют двум характеристикам -безопасности и живучести. Безопасность (safety) означает, что доступ к защищенному ресурсу в любой момент времени возможен только со стороны одного из процессов. Живучесть (liveness) означает, что программа когда-нибудь обязательно будет выполнена, иными словами, что она не остановится, и не будет ждать бесконечно. Безопасность - это статическое свойство, а живучесть - динамическое. Безопасности можно добиться за счет частичного или полного отказа от параллельного исполнения процессов. В действительности наиболее надежными являются строго последовательные программы,

поскольку в этом случае вообще невозможен параллельный доступ к ресурсу из различных частей программы.

Распространенный метод управления доступом к ресурсам - применение переменных защиты. Простейший метод защиты основан на одной двоичной переменной f1. Эта переменная изменяется обоими процессами таким образом, что один из них имеет доступ к защищенному ресурсу, когда f1 = true, а другой - когда f1 = false.

```
program protect_example (* защита ресурса *)
var fl: boolean;
begin
f1 : = true;
cobegin
while true do (* бесконечный цикл *)
begin (* процесс А *)
repeat until f1 = true;
(* защищенный ресурс *)
f1 := false;
…
end; (* процесс А *)
while true do (* бесконечный цикл *)
begin (* процесс В *) repeat until f 1 = false;
(* защищенный ресурс *)
f1: = true;
…
end; (* процесс В *)
coend;
end (* protect_example *)
```

Это решение удовлетворяет принципу взаимного исключения - два процесса проверяют переменную f1 и входят в критическую секцию только тогда, когда f1 имеет разные значения. Процесс, находящийся в критической секции, может считать, что он владеет ресурсом монопольно.

С другой стороны, это решение создает новые проблемы. Наиболее медленный процесс определяет общую скорость исполнения. Не имеет значения, является ли А быстрее, чем В или наоборот, поскольку каждый процесс для своего развития должен ждать, когда другой изменит значение f1. Кроме этого, если исполнение процесса по той или иной причине будет приостановлено, второй тоже должен быть остановлен, даже после одного цикла. Более того, циклы занятого ожидания (busy loop), в которых проверяется переменная защиты, напрасно расходуют процессорное время.

Эти проблемы - следствие введения управляющей переменной f1, которая для синхронизации доступа к ресурсу создает дополнительные связи между процессами. Модули, которые должны быть в принципе независимыми, связаны через f1, которая делает из двух модулей фактически последовательный процесс. Тот же результат можно получить, исключив f1 и выполняя оба процесса последовательно в одном цикле.

Другое решение — переустанавливать защитную переменную f1 после проверки ее значения и перед доступом к защищенному ресурсу т.е. все процессы должны иметь следующий вид

```
repeat until f1 = true;
f1 := false;
(* защищенный ресурс *)
f1 := true;
…
```

В этом случае процессы не связаны, и условие живучести выполнено, но решение не является корректным. Если прерывание для переключения процесса останавливает процесс А после контроля f1 = true, но перед присваиванием f1 = false, а процесс В производит аналогичную проверку f1, то оба процесса получают доступ к защищенному ресурсу, что противоречит требованию безопасности. Использование для защиты ресурса только одной переменной приводит к необходимости защищать переменную, поскольку она сама становится общим ресурсом.

Было предложено несколько решений, основанных на нескольких переменных защиты, но они скорее могут считаться курьезами, имеющими мало практического значения. В заключение отметим, что для синхронизации параллельных процессов лучше не вводить новых переменных, поскольку они добавляют новые связи и сами становятся общими ресурсами.

Чтобы обойти эту проблему, некоторые процессоры имеют команду test_and_set ("проверить_и_установить"), выполняющую проверку значения булевой переменной и ее переустановку в ходе одной операции, которую нельзя прервать. Смысл команды test_and_set в том, что на ее базе можно построить процедуры синхронизации и защиты ресурсов. Объединения в одной операции проверки переменной и ее модификации достаточно для обеспечения защиты.

Команда test_and_set функционально эквивалентна циклу read-modify-write на шине VMEbus. В обоих случаях гарантируется неразрывность двух операций - чтения и записи. Если команда test_and_set отсутствует в используемом языке программирования или в наборе команд процессора, то ее можно смоделировать другими средствами при условии, что допустим запрет прерываний на короткое время.

Реализация критических секций и взаимного исключения в распределенной системе сама по себе представляет проблему. Для начала, нет прямого эквивалента команды test_and_set, поскольку в этом случае имеется более одного процессора. В принципе, для каждого ресурса можно установить единого координатора. Любой процесс, желающий получить доступ к ресурсу, сначала запрашивает координатора, который дает разрешение только одному из запрашивающих процессов. Однако это решение не является столь простым, как кажется. Единый координатор процессов является узким местом - и при его отказе ресурс остается либо заблокированным, либо незащищенным. Более того, если ресурс является просто переменной в памяти, то строить целый алгоритм для его защиты нерационально. На самом деле координатор

сам является ресурсом, за доступ к которому будет происходить конкуренция, не говоря уже о том, что в распределенной системе для посылки запроса нужно еще получить и доступ к каналу связи.

Возможной альтернативой является использование маркера, который перемещается между процессами. При этом в критическую секцию может войти только владелец маркера. Здесь возникают те же проблемы, что и в сетях: в случае сбоя в процессе, являющемся владельцем маркера, должен существовать механизм регенерации маркера. Поэтому для защиты небольшого числа переменных в памяти этот метод может оказаться громоздким и трудно реализуемым.

В заключение отметим, что в распределенных системах не существует практичного, эффективного и простого метода защиты ресурсов, сравнимого с командой test_and_set в однопроцессорных конфигурациях. Каждый случай необходимо оценивать индивидуально и выбирать соответствующее практическое решение.

2. Предотвращение тупиков

Рассмотрим ситуацию, в которой два или больше процессов в системе приостановлены и ожидают каких-нибудь событий. Если такие события для каждого из ожидающих процессов могут быть инициированы только другим ожидающим процессом, то все процессы окажутся в состоянии бесконечного ожидания. Такая ситуация называется тупиком (deadlock) (рис. 1).

Похожая ситуация возникает, когда один или несколько процессов продолжают исполняться фактически на одном месте. Это называется зависанием (starvation). Например, если процесс непрерывно проверяет значение условной переменной, которое не изменяется, поскольку все остальные процессы также заняты проверкой. Иными словами, процессы, оказавшиеся в тупике, находятся в очереди ожидания (т. е. они блокированы), а зависшие процессы исполняются, но фактически на одном месте.

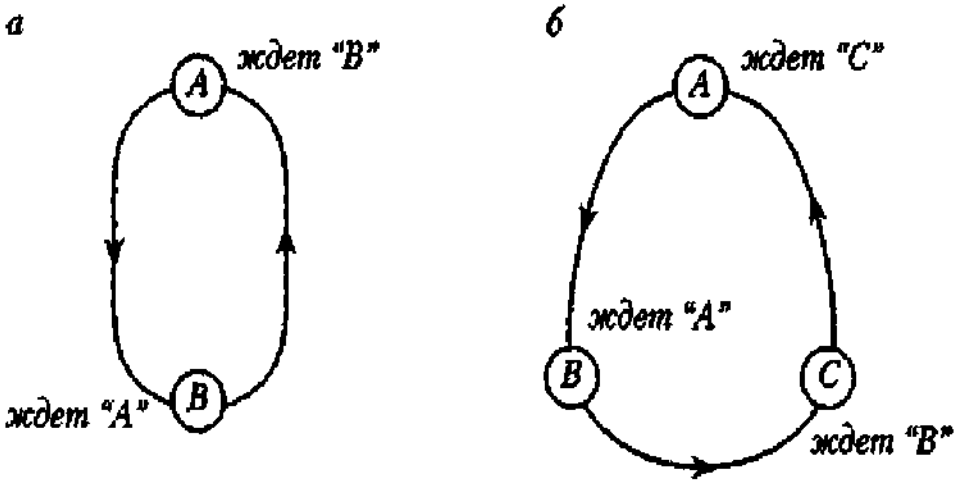


Рисунок 1. - Тупик: а – взаимный; б – циркулярный

Тупик и одновременный доступ к защищенному ресурсу являются двумя симметричными проблемами, относящимися к чрезвычайным ситуа-

циям. В одном случае каждый процесс ждет других процессов, во втором - несколько процессов выполняются параллельно.

Можно предложить несколько подходов к решению проблемы тупиков. Простейшим из них является полное игнорирование проблемы и работа в режиме, когда при возникновении тупика некоторые процессы уничтожаются, например оператором, или производится ручная перезагрузка системы. Естественно, такое решение неприемлемо для систем реального времени, в особенности, если они должны работать без участия оператора.

Кроме стратегии с привлечением оператора, существуют еще и другие -автоматического обнаружения и предотвращения. Первая предполагает перераспределение ресурсов для разрешения ситуации или, в крайнем случае, уничтожение одного из процессов. Вторая - распределение ресурсов так, что тупики не возникают вообще.

Для обнаружения тупиковой ситуации необходимо непрерывно проверять состояние всех исполняющихся процессов и их взаимодействие для обнаружения циклов типа показанных на рис. 1. Такой контроль можно делать с помощью фоновой (background) программы, периодически запускаемой планировщиком. Однако и такая программа не может гарантированно выявить все тупиковые ситуации. В распределенных системах информация о состоянии всех процессов на всех ЭВМ также должна поступать к программе обнаружения тупиковых ситуаций. Помимо повышенной нагрузки на сеть, которая при этом возникает, имеется риск несогласованности сообщений о состоянии процессов, в результате которого может произойти ошибочное выявление тупиковой ситуации с последующим уничтожением соответствующих процессов.

Предотвращение означает попытку вообще избежать ситуаций, которые могут привести к тупику. Программы должны быть структурированы и взаимодействие процессов должно быть организовано таким образом, чтобы тупиковые ситуации были исключены вообще.

Для возникновения тупика должны выполниться одновременно несколько условий. Если хотя бы одно из них не выполнено, тупик не может возникнуть.

1. Взаимное исключение. Существуют системные ресурсы, к которым разрешен монопольный доступ.

2. Невытесняющее распределение ресурсов. Ресурс может быть освобожден только тем процессом, который его захватил, или, иначе говоря, ресурс не может быть освобожден извне захватившего его процесса.

3. Последовательный захват ресурсов. Процесс запрашивает ресурсы по одному, т. е. по мере необходимости.

4. Захват ресурсов в обратном порядке.

Эти четыре утверждения косвенно дают ключ к предотвращению тупиковых ситуаций. Достаточно, чтобы одно из них не выполнялось, и тупик не возникнет.

Первое утверждение выполняется всегда, так как взаимное исключение является принципиальным для гарантии упорядоченного управления общими ресурсами.

Второе утверждение требует, чтобы операционная система распознавала тупиковые ситуации и реагировала соответственно, вынуждая процесс освободить ресурс. Это решение приемлемо лишь в случае, если допускается принудительное уничтожение процесса, и зависит от механизма восстановления.

В соответствии с третьим утверждением альтернативой выделению по одному является выделение всех необходимых ресурсов одновременно. Практичность этого решения, естественно, зависит от вида этих ресурсов и от того, могут ли без них обойтись другие процессы, пока не завершится захвативший их процесс. Если процессу выделяется десяток переменных на время выполнения небольшого количества команд, без информации о том, какие из них будут действительно использоваться — это одна ситуация, а если один процесс надолго блокирует доступ к целой базе данных — это совсем другое дело.

Если система структурирована в соответствии с моделью "клиент-сервер" и работает на основе замкнутых транзакций, то проблему тупиков решить проще. В случае возникновения тупика можно просто отменить транзакцию, а не уничтожать один или несколько процессов.

Нарушение четвертого запрета чаще всего приводит к тупикам. Если двум процессам требуются ресурсы А и В и первый их запрашивает в порядке А - В, а второй - В - А, то для возникновения тупика достаточно того, чтобы первый процесс был прерван после захвата ресурса А и управление было передано второму, который, в свою очередь, захватывает ресурс В. После этого каждый процесс будет бесконечно ждать, пока другой не освободит захваченный ресурс.

Четвертое утверждение дает практический способ избежать тупиков. Тупик можно предотвратить, если определен точный порядок (последовательность) запроса ресурсов, соблюдаемый всеми процессами. В приведенном примере это означает, что "А должен быть распределен перед В" и что все процессы строго следуют этому правилу. При этом освобождение ресурсов должно происходить в порядке, обратном их выделению. Этот метод, в принципе, несложно применить при разработке системы реального времени, пока процессы находятся в руках одного или небольшой группы программистов, но его практическая ценность быстро уменьшается при возрастающем числе ресурсов или разделяемых переменных.

4. Сигналы

Сигнал дает возможность задаче реагировать на событие, источником ко торого может быть операционная система или другая задача. Сигналы вызывают прерывание задачи и выполнение заранее предусмотренных действий. Сигналы могут вырабатываться синхронно, то есть как результат работы самого процесса, а могут быть направлены процессу другим

процессом, то есть вырабатываться асинхронно. Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например деление на нуль, ошибка адресации, нарушение защиты памяти и т. д.

Примером асинхронного сигнала является сигнал с терминала. Во многи ОС предусматривается оперативное снятие процесса с выполнения. Для этого пользователь может нажать некоторую комбинацию клавиш (Ctrl+C, Ctr1+Break) в результате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса (то есть он является асинхронным), требуя от процесса немедленного завершения работы. В данном случае реакцией на сигнал является безусловное завершение процесса

В системе может быть определен набор сигналов. Программный код процесса, которому поступил сигнал, может либо проигнорировать его, либо прореагировать на него стандартным действием (например, завершиться), либо выполнить специфические действия, определенные прикладным программистом. В последнем случае в программном коде необходимо предусмотреть специальные системные вызовы, с помощью которых операционная система информируется, какую процедуру надо выполнить в ответ на поступление того или иного сигнала.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут получить данные об идентификаторах друг друга.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, семафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются непригодными. В таких системах синхронизация может быть реализована только посредством обмена сообщениями.

Лекция 3.4. Обмен информацией между процессами

1. Общие области памяти.
2. Почтовые ящики.
3. Каналы.
4. Удаленный вызов процедур.
5. Сравнение методов синхронизации и обмена данными.

1. Общие области памяти

Взаимодействующие процессы нуждаются в обмене информацией. Поэтому многозадачная операционная система должна обеспечивать необходимые для этого средства. Обмен данными должен быть прозрачным

для процессов, т.е. передаваемые данные не должны изменяться, а сама процедура должна быть легко доступна для каждого процесса.

Простейший метод - использование общих областей памяти, к которым разные процессы имеют доступ для чтения/записи. Очевидно, что такая область представляет собой разделяемый ресурс, доступ к которому должен быть защищен, например, семафором. Главное преимущество общих областей памяти заключается в том, что к ним можно организовать прямой и мгновенный доступ, например один процесс может последовательно записывать поля, а другой затем считывать целые блоки данных.

При программировании на машинном уровне общие области размещаются в оперативной памяти по известным адресам. В языках высокого уровня вместо этого используются глобальные переменные, доступные нескольким дочерним процессам. Так, например, происходит при порождении потоков, для которых переменные родительского процесса являются глобальными и работают как общие области памяти. В случае возможных конфликтов доступа к общим областям они должны быть защищены семафорами.

2. Почтовые ящики

Другой метод, позволяющий одновременно осуществлять обмен данными и синхронизацию процессов, - это почтовые ящики. Почтовый ящик представляет собой структуру данных, предназначенную для приема и хранения сообщений (рис. 1). Для обмена сообщениями различного типа можно определить несколько почтовых ящиков.

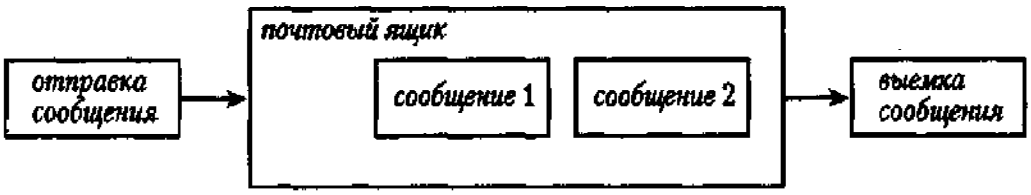


Рисунок 1. - Работа почтового ящика

Во многих операционных системах почтовые ящики реализованы в виде логических файлов, доступ к которым аналогичен доступу к физическим файлам. С почтовыми ящиками разрешены следующие операции: создание, открытие, запись/чтение сообщения, закрытие, удаление. В некоторых системах поддерживаются дополнительные служебные функции, например счетчик сообщений в почтовом ящике или чтение сообщения без удаления его из ящика.

Почтовые ящики размещаются в оперативной памяти или на диске и существуют лишь до выключения питания или перезагрузки. Если они физически расположены на диске, то считаются временными файлами, уничтожаемыми после выключения системы. Почтовые ящики не имеют имен подобно реальным файлам - при создании им присваиваются логические идентификаторы, которые используются процессами при обращении.

Для создания почтового ящика операционная система определяет указатели на область памяти для операций чтения/записи и соответствующие переменные для защиты доступа. Основными методами реализации являются

либо буфер, размер которого задается при создании ящика, либо связанный список, который, в принципе, не накладывает никаких ограничений на число сообщений в почтовом ящике.

В наиболее распространенных реализациях процесс, посылающий сообщение, записывает его в почтовый ящик с помощью оператора, похожего на оператор записи в файл

put_mailbox (# 1, message)

Аналогично, для получения сообщения процесс считывает его из почтового ящика с помощью оператора вида

get _mailbox (# 1, message)

Запись сообщения в почтовый ящик означает, что оно просто копируется в указанный почтовый ящик. Может случиться, что в почтовом ящике не хватает места для хранения нового сообщения, то есть почтовый ящик либо слишком мал, либо хранящиеся в нем сообщения еще не прочитаны.

При чтении из почтового ящика самое старое сообщение пересылается в принимающую структуру данных и удаляется из ящика. Почтовый ящик -это пример классической очереди, организованной по принципу FIFO. Операция чтения из пустого ящика приводит к различным результатам в зависимости от способа реализации — либо возвращается пустая строка (нулевой длины), либо операция чтения блокируется до получения сообщения. В последнем случае, чтобы избежать нежелательной остановки процесса, необходимо предварительно проверить число сообщений, имеющихся в данный момент в ящике.

3. Каналы

Канал (pipe) представляет собой средство обмена данными между двумя процессами, из которых один записывает, а другой считывает символы. Этот механизм был первоначально разработан для среды UNIX как средство перенаправления входа и выхода процесса. В ОС UNIX физические устройства ввода/вывода рассматривают как файлы, а каждая программа имеет стандартное устройство ввода (вход) и стандартное устройство вывода (выход), клавиатуру и экран монитора - можно переопределить, например, с помощью файлов. Когда выход одной программы перенаправляется на вход другой, создается механизм, называемый каналом (в операционных системах для обозначения канала используется символ "|"). Каналы применяются в операционных системах UNIX, OS/9 и Windows NT в качестве средства связи между процессами (программами).

Каналы можно рассматривать как частный случай почтового ящика. Различие между ними заключается в организации потока данных - почтовые ящики работают с сообщениями, то есть данными, для которых известны формат и длина, а каналы принципиально ориентированы на неструктурированные потоки символов. В некоторых операционных системах, однако, возможно определить структуру передаваемых по каналу данных. Обычно процесс, выполняющий операцию чтения из канала, ждет, пока в нем не появятся данные. В настоящее время операционные системы включают методы, по-

зволяющие избежать блокировки программы, если это нежелательно с точки зрения ее логики.

Операции над каналами эквивалентны чтению/записи физических файлов. Они включают функции, как определить, открыть, читать, записать, закрыть, удалить. Дополнительные операции могут устанавливать флаги режима доступа, определять размер буфера и т.д.

Благодаря тому, что ввод/вывод в файл и на физические устройства и вход/выход процессов трактуются одинаково, каналы являются естественным средством взаимодействия между процессами в системах "клиент-сервер". Механизм каналов в UNIX может в некоторых случаях зависеть от протокола TCP/IP, а в Windows NT каналы работают с любым транспортным протоколом. Следует иметь в виду, что внешне простой механизм каналов может требовать больших накладных расходов при реализации, особенно в сетевых системах.

4. Удаленный вызов процедур

Модель "клиент-сервер" построена на обмене сообщениями "регулярной" структуры, которые можно передавать, например, через механизм каналов.

Однако основной процедурой обмена данными и синхронизации в среде "клиент-сервер" является удаленный вызов процедур (Remote Procedure Call - RPC). Последний может рассматриваться как вызов подпрограммы, при котором операционная система отвечает за маршрутизацию и доставку вызова к узлу, где находится эта подпрограмма. Нотация обращения к процедуре не зависит от того, является ли она локальной или удаленной по отношению к вызывающей программе. Это существенно облегчает программирование.

В системе реального времени существенно, является RPC блокирующим или нет. Блокирующий RPC не возвращает управление вызывающему процессу, пока не закончит свою работу, например, пока не подготовит данные для ответа. Неблокирующие RPC возвращают управление вызывающей процедуре по истечении некоторого времени (time out) независимо от того, завершила ли работу вызываемая процедура; в любом случае вызывающая программа получает код, идентифицирующий результат выполнения вызова, - код возврата. Таким образом, неблокирующие RPC имеют важное значение, с точки зрения гарантии живучести системы.

5. Сравнение методов синхронизации и обмена данными Может показаться, что основные задачи, связанные с параллельным программированием, взаимным исключением, синхронизацией и коммуникациями между процессами, имеют мало общего, но, в сути - это просто разные способы достижения одной цели. Методы синхронизации можно использовать для организации взаимного исключения и коммуникаций. Аналогично, с помощью техники коммуникаций между процессами можно реализовать функции синхронизации и взаимного исключения процессов.

Например, семафор эквивалентен почтовому ящику, в котором накапливаются сообщения нулевой длины, - операции signal и wait эквивалентны

операциям put и get почтового ящика, а текущее значение семафора эквивалентно числу помещенных в почтовый ящик сообщений. Аналогично можно организовать взаимное исключение и защиту ресурсов с помощью почтовых ящиков. В этом случае сообщение выполняет функцию "маркера". Процесс, получивший этот "маркер", приобретает право входить в критическую секцию или распоряжаться ресурсами системы. При выходе из секции или освобождении ресурса процесс помещает "маркер" в почтовый ящик. Следующий процесс читает из почтового ящика, получает "маркер" и может войти в критическую секцию.

Связь между разными подходами имеет практическое значение в том случае, если в системе применяется только один из них, а все остальные нужно строить на его основе. Современные операционные системы, поддерживающие многозадачный режим и операции в реальном времени, применяют все упомянутые методы. Передача сообщений и доступ к общим областям памяти медленнее, чем проверка и обновление семафора и переменной события, и требует дополнительных накладных расходов. Если есть выбор между различными методами синхронизации и взаимодействия, следует использовать тот из них, который лучше решает конкретную проблему - результирующая программа будет понятнее и, возможно, быстрее работать. Кроме того, весьма важно оценить, насколько эффективно в имеющейся программной среде реализуются конкретные решения. Следует избегать незнакомых и неестественных конструкций.

В распределенных системах всегда существует риск потерять сообщение в сети. Если сетевая система сконфигурирована так, что она контролирует правильность передачи сообщения, и имеются средства для повторной передачи утраченных сообщений, то прикладная программа не должна осуществлять дополнительные проверки. Обычно нижний уровень операционной системы и процедуры сетевого интерфейса передают на более высокий уровень код возврата, который прикладная программа должна проверить, чтобы убедиться, была ли попытка успешной или нет, и при необходимости повторить ее.

Если контроль не предусмотрен, например, используется служба IP без транспортного протокола TCP, то прикладная программа несет ответственность за проверку результата передачи. Эта операция сложнее, чем это кажется. Можно использовать сообщение, подтверждающее прием, но нет гарантии, что оно само, в свою очередь, не будет потеряно и отправитель не начнет новую передачу. Эта проблема не имеет общего решения - стратегии передачи сообщений должны в каждом случае рассматриваться индивидуально. Возможным решением является помечать и нумеровать каждое сообщение таким образом, чтобы отправитель и получатель могли следить за порядком передачи. Этот метод используется в некоторых типах коммуникационных протоколов.

Лекция. 3.5. Операционные системы реального времени для интеллектуальных информационных систем

1. Обзор основных направлений развития операционных систем реального времени.

2. Операционная система Spox.

3. Операционная система Multiprox.

4. Операционная система VCOS.

5. Операционная система DEASY.

6. Операционная система UNIX.

7. Операционная система OSF/1 и DCE.

8. Операционная система VAX/VMS.

1. Обзор основных направлений развития операционных систем реального времени

Операционные системы реального времени (ОСРВ) используются в тех случаях, когда работоспособность обслуживаемой ими цифровой системы определяется не только результатом обработки поступившей информации, но и длительностью времени получения результата. Области практического применения ОСРВ очень широки. Это системы автоматизации производства, контрольно-измерительные системы, телекоммуникационная аппаратура, авиационно-космическая и военная техника, транспорт, системы обеспечения безопасности и ряд других приложений. В этих приложениях ОСРВ должна обеспечить не только получение необходимого логического результата — отклика на внешние события, но и реализовать требуемые интервалы времени между событиями и откликом или заданную частоту приема внешних данных и выдачи результатов.

Современные ОСРВ должны удовлетворять ряду противоречивых требований: малый объем, достаточный для размещения в резидентной памяти системы; малое время отклика; реализация многозадачного режима с гибким механизмом приоритетов, наличие сервисных функций и средств поддержки для разработки прикладных программ и ряд других. В настоящее время разработчику систем предлагается ряд ОСРВ, имеющих различные характеристики и прошедших апробацию в многочисленных областях применения, что позволяет ему найти компромиссное решение для выполнения поставленной задачи. Наиболее часто в системах на базе микропроцессоров и микроконтроллеров фирмы Motorola используются следующие ОСРВ: OS-9 фирмы Microware Systems; VxWorks фирмы WindRiver Systems; LynxOS фирмы Lynx Real-Time Systems; pSOS+ фирмы Integrated Systems; QNX фирмы Quantum Software Systems; VRTX/OS 3.0 фирмы Ready Systems; Nucleus фирмы Accelerated Technology; RTXC фирмы Embedded System Products; OSE фирмы Enea Data, Precise/MQX фирмы Intermetrics Microsystems Software; VMEexec фирмы Motorola.

Подразделяют ОСРВ на два класса - системы "жесткого" и "мягкого" реального времени (РВ). Системы "жесткого" РВ имеют минимальные объем и время отклика, но обладают ограниченными сервисными средствами. Типичным примером ОСРВ этого класса служит VMEexec. Системы "мягкого" РВ требуют большего объема памяти, имеют более длительное время откли-

ка, но удовлетворяют широкому спектру требований пользователя по режиму обслуживания задач, уровню предоставляемого сервиса. Примером такой ОСРВ может служить OS-9/9000.

Однако для современных ОСРВ данная классификация является весьма условной. Ряд ОСРВ, относящихся к классу "жестких", имеют средства интерфейса, которые позволяют, в случае необходимости, использовать высокоэффективные отладчики или интегрированные среды разработки, обеспечивая, таким образом, пользователя набором средств поддержки программирования-отладки систем. Например, VMEexec может использоваться совместно с интегрированной средой MULTI фирмы GreenHills Software, VxWorks с интегрированной средой Tornado, в составе которой поставляются отладчик CrossWind и GNU-компиляторы фирмы Cygnus Support, VRTX и pSOS с отладчиком XRAY и компиляторами фирмы Microtec Research. С другой стороны, системы "мягкого" РВ реализуются по модульному принципу, что позволяет использовать только те средства, которые необходимы в данном приложении. В результате для конкретного применения достигается существенное сокращение объема необходимой памяти и времени отклика. Например, для ядра OS9/9000 время отклика не превышает 20 мкс (для VMEexec, VxWorks, pSOS - менее 10 мкс), что является вполне достаточным для многих приложений.

Для создания многопроцессорных систем, работающих в режиме реального времени (РВ), необходимо базовое программное обеспечение, а именно операционная система. ПО этого направления делится на две большие группы. К первой группе можно отнести небольшие модули, загружаемые на ЦОС-процессоре, а также библиотеки подпрограмм для основного процессора, позволяющие реализовать обмен данными. ЦОС-процессор в такой системе является подчиненным процессором, управляемым основным (host-процессором). Организация функций систем РВ основана на обработке прерываний и механизме обмена сообщениями. Главное достоинство этих систем – небольшая цена. К системам этого типа можно отнести VCOS и DEASY.

Вторая группа операционных систем – это операционные системы реального времени типа Spox или Multiprox. Цена этих систем составляет порядка 20-40 тыс. долларов, но возможности их значительно выше. Операционная система Spox фирмы Spectron Microsystems – одна из ведущих систем, используемых в системах РВ для различных применений, в том числе на платформах с модулями ЦОС. Spox – это специализированная операционная система реального времени, создающая операционное окружение для приложений по обработке данных в реальном режиме.

Другая система – Multiprox фирмы Comdisco. Multiprox – это система разработки, которая позволяет инженерам графически формировать приложения и разделять ЦОС-задачи для нескольких ЦОС-процессоров.

2. Операционная система Spox

Spox (создана фирмой Spectron в 1987 г.) является операционной системой, которая структурирована для обработки сигналов и приложений с ин-

тенсивной математикой. Это высокоуровневое окружение для приложений имеет простые в использовании свойства, включая независимый от устройств ввод-вывод, удобные установки процессора и интерфейс с основной машиной (имеется в виду основная ОС). Spox обеспечивает объектно-ориентированную модель для ЦОС и математической обработки.

В последние годы фирма Spectron ввела OSPA (открытая архитектура обработки сигналов) – расширение к Spox для ЦОС-приложений на основной машине. Запускаясь под MS Windows, OSPA обеспечивает интерфейс на уровне основной машины. Используя этот интерфейс, host-приложения могут планировать и контролировать работу многочисленных программ на ЦОС-сопроцессорах (но это не параллельная обработка). OSPA является своего рода интерфейсом API (интерфейсом прикладных программ), который облегчает интеграцию ЦОС-обработки в интерактивное приложение.

Spectron изначально развивал Spox для TMS320C30, но сейчас операционная система запускается также и на Motorola 96002, TI C40 и Analog Device 21020. Spectron также выпускает версию Spox для параллельной обработки. Используемая модель обработки сообщений поддерживает многозадачность. Многозадачное расширение построено вокруг примитивов, основанных на сообщениях, и может обеспечить высокоскоростную передачу данных через каналы ввода-вывода. Spox позволяет совместно использовать память, установленную на отдельном модуле. SPOX поставляется в следующих четырех основных конфигурациях.

Однородные встраиваемые системы. Процессор ЦОС играет роль как общецелевого, так и специализированного процессора. По существу, ЦОС-процессор замещает специализированный контроллер. Со SPOX ЦОС-процессор одновременно выполняет алгоритмы обработки сигналов вместе со сложным контролем по связи задач, прежде выполняемых на специализированных контроллерах. Для приложений, требующих дополнительных мощностей, можно просто использовать дополнительные ЦОС-процессоры. SPOX поддерживает многопроцессорность.

Разнородные встраиваемые системы. Встроенные компьютерные системы реального времени с полными чертами операционной системы (например, VXWorks, OS-9, LynxOS) выполнены на основе ЦОС-подсистем. Это традиционная конфигурация, где ЦОС-подсистема прибавляется к встраиваемой компьютерной системе. Извлекая выгоду из приложений ЦОС в этих системах, Spectron предлагает сбалансированный подход, объединяющий традиционные встраиваемые компьютерные системы (используемые в промышленности) и DSP. Это открывает новый диапазон возможностей для проектирования встроенного управления.

Компьютерные интегрированные системы. В данной конфигурации рабочие станции контролируют ЦОС-подсистемы. Приложение ЦОС запускается в привычном интерактивном окружении (MS-Winows, Unix, DOS), выполняя приложение как тест или измерение, мониторинг контроля процесса, медицинские представления, сбор данных.

Здесь приложение имеет ресурсы как основной, так и ЦОС-системы, действуя под управлением рабочей станции.

Мультимедиа системы. Компьютер требует мощных вычислительных затрат по воспроизведению мультимедийных приложений, что соответствует задачам ЦОС: аудиозапись и воспроизведение, видео в реальном режиме, распознавание речи, синтез звука, телекоммуникационные функции, такие, как факс, модем. ЦОС-модуль размещается либо на материнской плате, либо на дополнительной плате, а SPOX усиливает возможности мультимедиа в привычном пользовательском окружении.

SPOX поддерживает высокопроизводительную многозадачность, обработку прерываний, управление памятью, ввод-вывод в реальном времени и большой набор функций по обеспечению взаимосвязи между задачами и процессорами. SPOX имеет математическую и специализированную ЦОС-библиотеку функций, многие из которых написаны на ассемблере для повышения производительности. SPOX поддерживает модель объектно-ориентированного программирования над векторами, матрицами и фильтрами. Для обеспечения этого имеется символьный отладчик и компиляторы языков высокого уровня, таких, как Си. Библиотека SPOX может использоваться на различных платформах, позволяя сосредоточиться на создании собственного приложения, не отвлекаясь на зависимость от платформы.

3. Операционная система Multiprox

Второе направление в развитии ПО – это Multiprox фирмы Comdisco, – пакет, который является новым выбором SPW (Signal Processing Workstation). Используя инструментальное множество, инженеры могут определять ЦОС-приложения графически, используя графические объекты, которые представляют компоненты ЦОС- обработки. Multiprox позволяет инженерам выделять разделы среди потоков данных, рисовать диаграмму течения данных и определять порции, работающие на разных процессорах.

Дополнительно SPW-инструментальное множество и Multiprox автоматически преобразуют диаграммы к процессорно-зависимому Си-коду и встраивают в ПО связи или межпроцессорную коммуникацию, чтобы передавать данные от одного процессора другому. Диаграммы потоков данных преобразуются в Си-программу, содержащую подпрограммы, некоторые из которых написаны на ассемблере и вручную оптимизированы. Таким образом, инженер может использовать инструменты, чтобы распределять высокоуровневое ПО на различные процессоры или смешивать процессоры.

4. Операционная система VCOS

VCOS (Visible Caching Oparating System) делает процессоры ЦОС со-процессорами. VCOS – переносимая, многозадачная и многопроцессорная операционная система реального времени. VCAS (VCOS Application Server) – резидентная на host-системе программа, загружает и связывает ЦОС-задачи и обеспечивает управление памятью и буферизацию ввода-вывода между host- и ЦОС-процессорами.

VCOS является “минимальной” операционной системой – она занимает менее 400 32-разрядных слов в памяти процессора. ОС использует

память host-системы для запоминания программы и данных. Она использует ее как ресурс, чтобы кэшировать данные и код для более быстрой обработки на процессоре. VCOS является “подчиненной” по отношению к host ОС, которая располагает и контролирует VCOS структуры данных, избегая таким образом соперничества между подсистемами при доступе к памяти. VCOS выполняется в высоко приоритетном режиме.

VCOS поставляется вместе с полной библиотекой ЦОС-функций для мультимедийных приложений. Эти приложения включают V.32 модем, V.29 FAX модем, видеозапись, обработку речи, графику, функции сжатия аудио- и видеоинформации.

5. Операционная система DEASY

Операционная система DEASY предназначена для разработки, отладки и выполнения программ ЦОС для процессорных модулей DSP3x фирмы “Инструментальные системы”. DEASY – однозадачная система реального времени. Ни одна из системных функций не блокирует обработку внешних событий (прерываний процессора) на величину более 500 нс.

Основная концепция при разработке программ с использованием DEASY заключается в том, что вокруг сигнального процессора создается виртуальная операционная среда, позволяющая ему играть роль центрального процессора для всего вычислительного комплекса, построенного на базе ПК. Таким образом, обеспечивается прозрачный доступ процессора ЦОС ко всем ресурсам ПК, включая экран монитора, клавиатуру, дисковые устройства, память, порты ввода-вывода и т.д. При этом облегчается процесс переноса программ из другой среды программирования и быстрое прототипирование программ обработки сигналов с использованием традиционных способов.

Операционная система DEASY включает набор библиотек и утилит для составления и отладки прикладных программ ЦОС.

Библиотека System.a30 содержит набор функций для управления процессором TMS320C30 и доступа к его внутренним регистрам из Си-программ.

Библиотека Host.lib состоит из функций для инициализаций, загрузки, управления и обмена информацией между платой ЦОС и ПК.

Библиотека Deasy.a30 содержит набор функций-аналогов библиотеки компилятора Borland C и используется для “прозрачного“ доступа к ресурсам ПК из прикладной программы, выполняемой на плате ЦОС.

Библиотека Bgi.a30 включает в себя функции-аналоги графической библиотеки компилятора Boland C и используется для доступа к графическим ресурсам IBM PC.

Исполняющая среда Deasy.exe предназначена для загрузки и выполнения прикладных программ, составленных с использованием приведенных выше библиотек. Она также содержит простейший диалоговый монитор интерактивного взаимодействия с платой ЦОС.

Символьный отладчик Kg30.exe, или символьный отладчик Cq30.exe, предназначен для отладки прикладных программ, выполняемых на плате ЦОС.

Библиотеки системных функций System и Host. Для организации работы многозадачного режима необходимо обмениваться сообщениями, которые отображают текущее состояние работы программ. Например, после окончания вычислений на одном процессоре (DSP) необходимо сообщить центральному процессору (CPU) о том, что он может забрать данные. При пересылке данных с CPU на DSP необходимо сообщить DSP, что он должен производить расчет. Библиотеки системных функций System и Host позволяют организовать обмен сообщениями и данными.

Библиотека System предназначена для компоновки с программами пользователя, написанными на языке Си или на Ассемблере для процессора TMS320C30. Библиотека Host предназначена для управления платой DSP со стороны ПК из программы пользователя, написанной на языке Си. Данная библиотека поставляется для компиляторов: Borland C/C++, Microsoft C, Watcom C.

6. Операционная система UNIX

Операционная система UNIX представляет собой многозадачную, многопользовательскую операционную систему и является в настоящее время одной из наиболее распространенных в мире. Она была первоначально разработана в 1970-е годы в AT&T Bell Laboratories. Особое внимание к переносимости, интерфейс пользователя, построенный на немногих базовых принципах, и возможность объединения различных UNIX-систем в сети независимо от аппаратной платформы очевидным образом способствовали успеху и распространению UNIX.

С момента своего появления система UNIX непрерывно развивалась и в настоящее время существует в нескольких модификациях. Основными ее распространителями являются компании AT&T Bell Laboratories и Berkeley Software Distribution. Почти все производители вычислительной техники предлагают UNIX либо как коммерческий продукт третьих фирм, либо как специально адаптированную версию для собственной аппаратной платформы. Некоторые специальные предложения отличаются скорее особенностями лицензирования, а не различием в выполняемых ими функциями. Кроме того, для сохранения совместимости и переносимости версии UNIX разных производителей не могут слишком сильно отличаться друг от друга.

В UNIX были введены средства, которые впоследствии были позаимствованы другими операционными системами. На базе UNIX была разработана операционная система OSF/1, а многие функции были включены в Windows NT. UNIX также явилась основной базой для разработки важных коммуникационных интерфейсов, в частности протокола TCP/IP и протокола пользовательского терминала X Window.

UNIX состоит из небольшого ядра, управляющего системными ресурсами (процессор, память и ввод/вывод), а остальная часть процедур операционной системы, и в частности управление файловой системой, работают как пользовательские процессы. Типичная операционная система UNIX содержит 10000-20000 строк на языке С и 1000-2000 строк машинно-ориентированных программ на ассемблере, которые разрабатываются от-

дельно для каждой аппаратной платформы. Ядро представляет собой единую резидентную программу размером от 100 Кбайт до 1 Мбайт в зависимости от платформы и выполняемых функций. При переносе системы UNIX на конкретную платформу требуется переписать заново только машинно-зависимую часть ядра. Это означает, что UNIX может работать на многих аппаратных платформах с идентичным системным интерфейсом.

Ядро UNIX имеет недостаточно продуманную структуру. Это следствие ее быстрого успеха и распространения, поскольку каждая новая версия должна была быть совместима с предыдущей. Первоначально система UNIX была разработана как многопользовательская, а не для приложений реального времени. Из-за того, что подпрограммы операционной системы работают как пользовательские процессы, но с наивысшим приоритетом, назначенным системой, невозможно прерывать также те системные вызовы, выполнение которых занимает много времени, что увеличивает время реакции системы. Это является существенным недостатком для задач реального времени, особенно управляемых прерываниями. В UNIX используется довольно сложное описание контекста, что увеличивает время переключения процессов. Из-за того, что в UNIX все операции с каналом построены на основе переключения процессов, применение этого механизма для связи между процессами в приложениях реального времени может приводить к задержкам.

Стандартно процессы в UNIX протекают с разделением времени. Для того чтобы дать всем процессам возможность исполняться, применяется динамическое распределение приоритетов. Процессу, готовому для исполнения, сначала присваивается его номинальный приоритет. Во время исполнения значение этого приоритета уменьшается до тех пор, пока он не становится меньше приоритета следующего из ожидающих процессов, который после этого выбирается для исполнения. В результате процессы с более высоким начальным приоритетом получают большую долю процессорного времени, и при этом все процессы периодически исполняются. Системные обращения синхронизированы с вызывающим процессом - он должен ждать, пока запрошенная операция не выполнится и ему не будет возвращено управление.

Важной особенностью, реализованной в UNIX, является одинаковая трактовка всех устройств. Внешние устройства ввода/вывода рассматриваются как файлы. Это существенно упрощает программы, требующие определенной гибкости, так как можно осуществить перенаправление ввода/вывода между файлами или внешними устройствами, такими как локальный или удаленный терминал или принтер, без изменения кода программы. Это также важно и с точки зрения машинной независимости программ.

Общим и вызывающим критику недостатком UNIX является его недружественный пользовательский интерфейс. Все еще в ходу старые и непонятные команды, а если и есть заменяющие их, то с именами или сокращениями, которые столь же неестественны, как и предыдущие. В некоторых системах пользовательские оконные интерфейсы и меню способны в основном "транслировать" выбранные действия в стандартные команды UNIX. Положительной особенностью команд UNIX является то, что благодаря

стандартизации ввода/вывода и механизму каналов несколько команд можно объединить в одной строке, причем выход одной команды является входом следующей. Такая техника позволяет для выполнения сложных операций вместо длинных командных файлов использовать всего несколько строк.

Хотя в начале UNIX была многозадачной операционной системой, не предназначенной для работы в реальном времени, из-за широкого распространения в научной и технической среде стала очевидной необходимость ее адаптации и к задачам реального времени. Поэтому новые версии поддерживают такие функциональные элементы систем реального времени, как семафоры, разделяемую память, обмен сигналами между процессами, приоритетное управление задачами и прямой доступ к внешним устройствам. POSIX представляет собой машинно-независимый интерфейс операционной системы, базирующийся на UNIX, определенный стандартом IEEE 1003.1-1988.

7. Операционная система OSF/1 и DCE

Первоначальные версии UNIX не требовали лицензий и были доступны практически всем для свободного использования, что отчасти объясняет популярность этой системы. При выпуске System V компания AT&T решила распространять ее только с оплатой лицензий. Некоторые наиболее крупные производители ЭВМ - Digital, Equipment, Hewlett Packard, IBM и др. - отреагировали на это, создав организацию Open Software Foundation (OSF) для того, чтобы не зависеть от диктата одной единственной компании-поставщика операционных систем. OSF разработала UNIX-совместимую операционную систему, а также другие продукты без лицензионных ограничений со стороны одной компании.

OSF/1 является модульной операционной системой, основанной на Mach, машинно-независимом мультипроцессорном ядре, разработанном в Carnegie-Mellon University (г. Питтсбург, США) в качестве инструмента для эмуляции других операционных систем. На основе Mach действительно удается одновременно эксплуатировать различные операционные системы на одной ЭВМ.

Для обеспечения переносимости OSF/1 совместима с AT&T UNIX System V и спецификациями программных интерфейсов Berkeley. Поскольку Mach и OSF/1 не содержит какого-либо кода UNIX, проблема лицензирования со стороны третьих компаний полностью снята.

В дополнение к средствам UNIX OSF/1 предлагает собственный набор функций, облегчающих разработку и выполнение программ. OSF/1 предназначена для работы в сетевой среде и поддерживает протокол TCP/IP. Файловая система OSF/1 также совместима со службой NFS протокола TCP/IP.

OSF разработала и другие продукты для распределенной вычислительной среды. OSF/Motif является графическим интерфейсом пользователя, обеспечивающим стандартное взаимодействие приложения с графическим терминалом.

Распределенная вычислительная среда (Distributed Computing Environment - DCE) представляет собой набор служб и средств для разработки, исполнения и поддержки приложений в распределенной среде. DCE может быть интегрирована с OSF/1, но является независимой от нее и в действительности может эксплуатироваться на базе других операционных систем.

8. Операционная система VAX/VMS

VMS является операционной системой для ЭВМ компании Digital Equipment с 32-разрядным процессором серии VAX. Ее популярность в приложениях управления связана в основном с качеством техники, на которой она используется, и большим количеством предусмотренных средств разработки. VMS может применяться как в среде реального времени, так и в многопользовательской среде с соответствующими средствами защиты.

VMS предоставляет широкий набор функций, стандартный и ясный интерфейс для прямых обращений из программ. Это позволяет осуществлять интеграцию любых языков со всеми функциями операционной системы. Из функций реального времени VMS имеет почтовые ящики в форме логических, состоящих из записей файлов, возможность создания резидентных подпрограмм и обработку прерываний. Процесс в VMS может управлять условиями своего собственного исполнения (приоритет, распределение памяти), создавать другие процессы и управлять их исполнением. Иерархическое управление препятствует процессам с низким приоритетом модифицировать параметры исполнения процессов с высоким приоритетом.

Как и во всех больших операционных системах, в VMS возникают проблемы в случаях, когда предъявляются жесткие требования по времени. По этой причине и ввиду популярности системы VMS, была разработана специальная версия, приспособленная для приложений реального времени, которая называется VAX/ELN. Она состоит из двух различных продуктов - рабочей среды для исполнения прикладных программ на целевой ЭВМ и пакета для разработки программ с компиляторами для различных языков. Разработка программ осуществляется на большом комплексе, имеющем ресурсы для подготовки системы, которая в итоге содержит только программные модули, необходимые для конкретного приложения. Затем в окончательном виде система загружается на рабочую ЭВМ.

Таким образом, операционная система предоставляет процессам логическую среду, состоящую из времени ЦП и оперативной памяти. Операционные системы для многопользовательских приложений и приложений реального времени имеют много общего, но техника программирования должна быть разной - приложения реального времени могут требовать времени реакции порядка 1 мс. При программировании в реальном времени используются специальные функции для координации работы различных процессов. Для обычных программ эти функции не требуются. Кроме этого, программы реального времени управляются прерываниями и могут явно ссылаться на время.

Центральная проблема многозадачного программирования и программирования в реальном времени - координация доступа к защищенным ресур-

сам. Существует много общего между распределением процессорного времени, защитой ресурсов и управлением доступом к общей шине. Во всех этих случаях ресурс - процессорное время, память, шина - в определенном смысле ограничен и должен распределяться между различными объектами безопасно, эффективно и справедливо. Стратегия разделения ресурсов, которая может основываться на простом циклическом или сложном динамическом механизме планирования, должна позволять избегать тупиков и блокировок, обеспечивать выделение ресурсов всем запрашивающим объектам и максимальную эффективность исполнения процессов. На нижнем уровне наиболее простым средством синхронизации является инструкция test_and_set. Наиболее часто используемые методы синхронизации и связи — это семафоры и почтовые ящики, которые в разных операционных системах реализуются поразному.

Результаты теории параллельного программирования играют важную роль на практике, так как соответствующие решения подкреплены формальными доказательствами. Это справедливо в особенности для систем реального времени, поскольку тестирование программ в этом случае представляет особую трудность. Применение проверенных методов дает разумную гарантию правильности соответствующих приложений.

Лекция 3.6. Операционные системы реального времени OS-9 и VxWorks

1. Операционная система реального времени OS-9.
2. Операционная система VxWorks.

1. Операционная система реального времени OS-9

В качестве примера современной ОСРВ рассмотрим OS-9, которая широко используются в системах автоматизации производства и телекоммуникационных системах, реализованных на базе микропроцессоров и микроконтроллеров фирмы Motorola. Эта ОСРВ имеет две версии: OS-9 написана на языке Ассемблера Motorola 68K и предназначена для работы с семействами M680x0 и M683xx, OS-9000 написана на языке C и может работать с семействами MPC6xx, MPC5xx, MPCSxx, MCF52xx, а также с микропроцессорами ряда других производителей: Intel 486, Pentium, SPARC, MIPS. Обе версии обеспечивают полную совместимость объектных кодов, поэтому для них обычно используется общее название OS-9. В качестве инструментального компьютера OS-9 использует IBM-PC, работающие в среде Windows, или рабочие станции SUN, HP, IBM RS/6000 с операционными системами типа UNIX.

Характерными особенностями OS-9 являются модульность и гибкость ее структуры. Модульность обеспечивает возможность конфигурации целевой ОСРВ в соответствии с классом решаемых задач. За счет исключения неиспользуемых модулей достигается сокращение объема памяти и стоимости системы. Гибкость структуры позволяет достаточно

просто и быстро производить реконфигурацию системы, расширение ее функциональных возможностей.

Все функциональные компоненты OS-9 - ядро реального времени, файловые менеджеры, средства (OS-9 kernel), средства общего управления внешними устройствами (I/O man), разработки - реализованы в виде автономных модулей (рис.1). Комбинируя эти модули, разработчик может создавать целевые операционные системы различной конфигурации и функциональных возможностей - от несложных резидентных ОСРВ, хранящихся во внутреннем ПЗУ микроконтроллера, до сложно-функциональных многопользовательских систем разработки. Все модули OS-9 могут размещаться в ПЗУ. Любые модули могут удаляться или добавляться с помощью простых команд, не требующих повторной компиляции или перекомпоновки.

OS-9 предоставляет пользователю возможность выбора ядра в зависимости от функционального назначения системы.

Ядро Atomic имеет малый объем (28 Кбайт) и обеспечивает минимальное время отклика. Например, при использовании микропроцессора MC68040 с тактовой частотой 25 МГц время реакции ядра на запрос прерывания составляет всего 3 мкс, что соответствует быстродействию систем "жесткого" РВ. Это ядро реализует минимальное число сервисных функций (дистанционную загрузку, связь с локальной сетью, управление ведомыми микроконтроллерами) и применяется в системах, встраиваемых в различную аппаратуру.

Ядро Standard обеспечивает выполнение широкого набора функций сервиса и разработки прикладных программ. Однако для реализации этих функций требуется больший объем памяти - 67 Кбайт ПЗУ и 38 Кбайт ОЗУ для систем на базе M68x0x,M683xx (версия OS-9), до 512 Кбайт для этих же систем, использующих пакет поддержки обмена по сети Интернет, 75 Кбайт ПЗУ и 24 Кбайт ОЗУ для систем на базе PowerPC (версия OS-9000). Применение ядра Standard с различным набором других функциональных модулей позволяет реализовать системы различной сложности и назначения - от встраиваемых в аппаратуру контроллеров с резидентным программным обеспечением и простейшими средствами ввода-вывода до сложно-функциональных систем класса рабочих станций с развитой сетевой поддержкой и обеспечением разнообразных функций сервиса, включая мультимедиа.

Файловыми менеджерами называются модули, управляющие логическими потоками данных, каждый из которых имеет определенное функциональное назначение и спецификацию. В состав OS-9 входят более 20 файловых менеджеров, которые можно разделить на три группы: стандартные, сетевые и коммуникационные, графические и мультимедиа. Рассмотрим основные из них (рис.2).

Стандартные менеджеры входят в основной комплект системы и предназначены для выполнения базовых функций обмена с внешними устройствами. К ним относятся следующие файловые менеджеры:

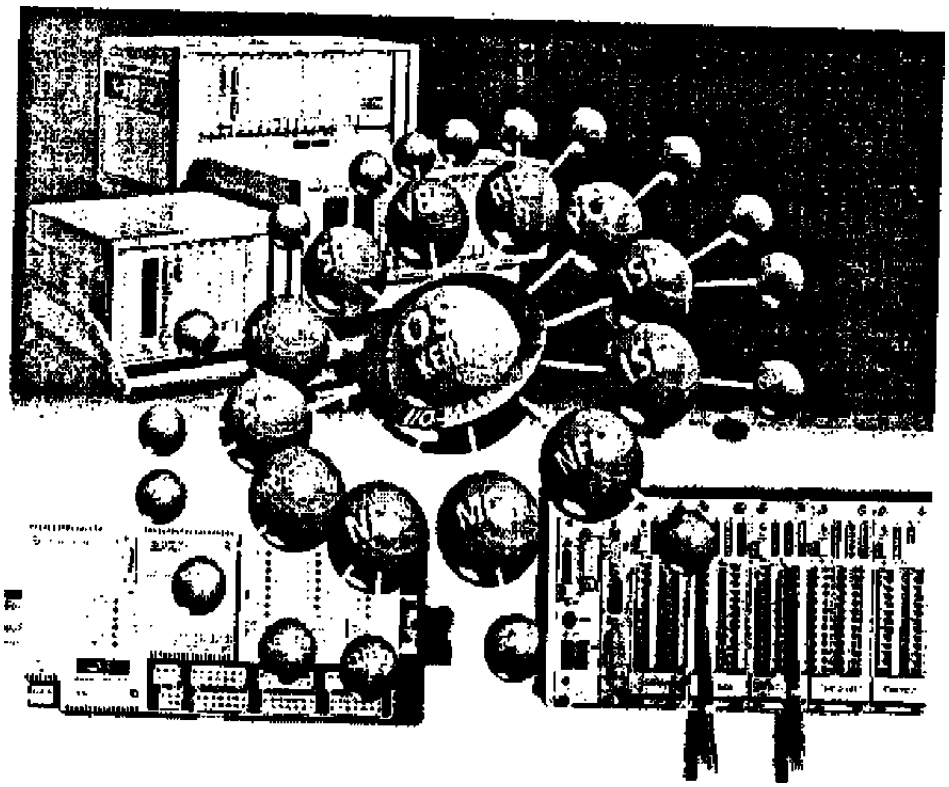


Рисунок 1. - Модульная структура ОСРВ OS-9

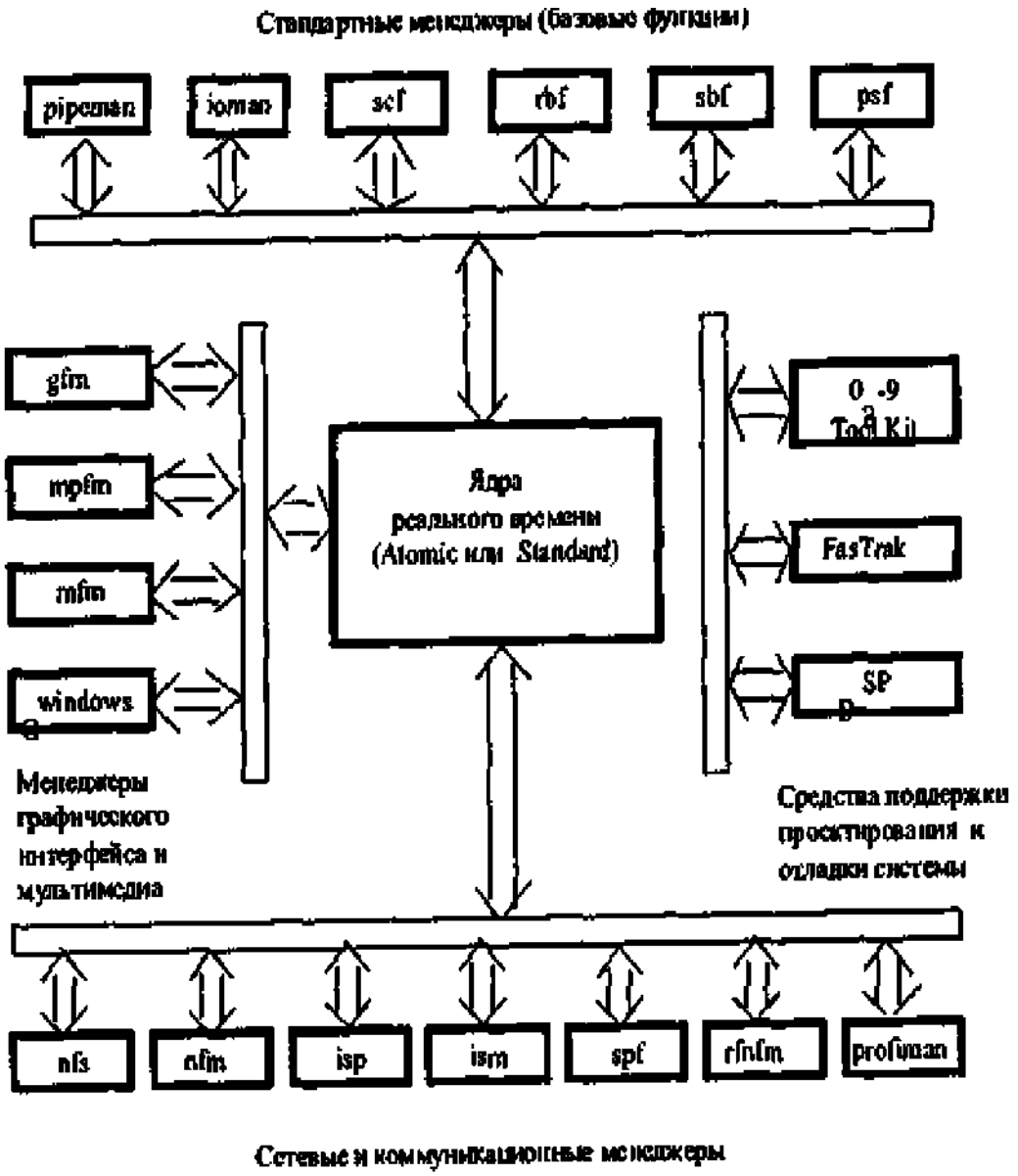


Рисунок.2. - Функциональный состав основных программных средств OS-9

pipeman - организующий очередь поступающих команд;

ioman - выполняющий общее управление внешними устройствами;

scf - управляющий байтовым последовательным обменом (связь с терминалом и другими устройствами по последовательному каналу);

rbf - управляющий блочным обменом с прямым доступом памяти (связь с дисковой памятью);

sbf - управляющий блочным последовательным обменом (связь с накопителями на магнитных лентах и другими устройствами);

pcf - поддерживающий файловую DOS-систему (поставляется по требованию заказчика).

Сетевые и коммуникационные менеджеры обеспечивают работу OS-9 с различными сетями и обмен данными по каналам связи с наиболее распространенными стандартами протоколе обмена. Чаще всего из этой группы используются следующие менеджеры:

nfs, nfm, isp - обеспечивающие основные протоколы сетевого обмена;

profiman - реализующий протокол обмена с шиной Profibus;

ism - реализующий обмен по сети цифровой связи стандарта ISDN;

spf - обеспечивающий пакетный обмен по стандартному протоколу X.25;

rtnfm - поддерживающий обмен по сети реального времени.

Для реализации графического интерфейса и работы с мультимедиа-приложениями используются файловые менеджеры:

gfm - графический менеджер реального времени;

mpfm - управляющий движущимися изображения;

mfm - обеспечивающий пользовательский интерфейс с мультимедиа;

g-windows - система оконного графического интерфейса, разработанная фирмой Gespac в виде файлового менеджера для OS-9.

Физический интерфейс OS-9 с разнообразными внешними устройствами обеспечивается большим набором драйверов, которые созданы как фирмой Microware Systems, так и многочисленными разработчиками аппаратуры, использующей эту операционную систему для конкретных приложений. Большинство драйверов, реализующих интерфейс со стандартными внешними устройствами, входят в пакет, из которого пользователь может выбрать необходимые средства для своего проекта.

В составе OS-9 имеется также пакет программ BSP для поддержки плат развития, который обеспечивает совместную работу OS-9 с целым рядом SBC, включая SBC типа MVME162, 172, 1603, 1604 фирмы Motorola. Используя BSP, OS-9 и какой-либо из этих SBC, разработчик может быстро сконфигурировать мощную целевую систему для своего конкретного приложения.

OS-9 содержит средства поддержки программирования, позволяющие проектировщику создавать прикладное программное обеспечение. Эти средства включают компиляторы Ultra C/C++, текстовый редактор EMACS, три вида отладчиков, в том числе символьные, а также разнообразные утилиты для организации контроля и сборки программных проектов. Кроме того, проектировщик может использовать большой набор совместимых с OS-9 текстовых редакторов, компиляторов C/C++, Forth, Ada, Modula-2 и других языков, которые разработаны рядом других фирм.

Для удобства пользователя совместно с OS-9 поставляются набор средств программирования-отладки OS-9 Tool Kit, интегрированная среда разработки FasTrak. В состав OS-9 Tool Kit входят основные средства разработки программ, указанные выше.

Интегрированная среда разработки FasTrak предоставляет пользователю наиболее полный комплект средств программирования-отладки. FasTrak имеет две версии: для функционирования в среде Windows на инструментальных компьютерах IBM-PC; для функционирования с системой UNIX на рабочих станциях SUN, HP, IBM RS/6000. Часть программных средств FasTrak инсталлируется на инструментальном компьютере, а часть - на целевой системе пользователя. Интерфейс инст-

рументального компьютера и целевой системы осуществляется файловым менеджером isp, который реализует протокол TCP/IP, обеспечивая связь по последовательному каналу или по сети Ethernet.

Среда FasTrak интегрирует все средства, необходимые для поддержки проектирования-отладки целевых систем. Версия FasTrak для IBM-PC содержит высокоэффективный текстовый редактор Premia's Codewright, который имеет средства перекодировки клавиатуры, обеспечивающие пользователю возможность вести редактирование в удобном для него формате. Версия для UNIX-станций позволяет использовать любой редактор, функционирующий с ОС UNIX. В состав FasTrak входят компиляторы Ultra C/C++, возможно также использование других компиляторов, например GNU C/C++ фирмы Cygnus Support. Отладчики FasTrak обеспечивают два режима отладки: пользовательский для создания прикладных программ, и системный, который выполняет обслуживание прерываний, системных вызовов и обращение к ядру РВ. Реализуется также отладка мультипроцессорных систем. При выполнении контрольных прогонов рабочей программы программа-профилировщик дает информацию о количестве обращений к различным программным модулям и времени их выполнения. В составе среды FasTrak имеются средства интерфейса с логическими анализаторами фирмы Hewlett-Packard и схемными эмуляторами фирм

Hewlett-Packard, EST, Applied Microsystems, Orion. Широкий набор функциональных возможностей делает среду FasTrak эффективным средством создания программного обеспечения для разнообразных микропроцессорных и микроконтроллерных систем.

Модульная структура ОСРВ OS-9 позволяет легко конфигурировать ее в соответствии с потребностями заказчиков. В настоящее время фирма Microware Systems поставляет ряд системных пакетов, ориентированных на различные сферы приложения:

Wireless OS-9 - для разработки устройств беспроводной связи: сотовых телефонов, пейджеров, портативных цифровых ассистентов (PDA);

Internet OS-9 - для разработки устройств с доступом к сети Интернет;

Digital Audio/Video Interactive Decoder (DAVID) OS-9 - для разработки распределенных систем цифрового интерактивного телевидения.

Таким образом, ОСРВ OS-9 позволяет удовлетворить запросы широкого круга разработчиков, создающих системы реального времени и программное обеспечение для них. Полную информацию об этой ОСРВ и возможностях ее использования для конкретного применения можно получить в сети Интернет по адресам http:/www.microware.com или http:/www.rtsoft.ru.

2. Операционная система VxWorks

VxWorks относится к классу систем "жесткого" РВ. Эта ОСРВ предназначена для работы с семействами M680xO, M683xx, MPC6xx, MPC5xx, MPCSxx, MCF52xx, а также с микропроцессорами других производителей: Intel 486, Pentium, SPARC, MIPS, DEC Alpha, HP PA-RISC. В качестве инструментального компьютера используются IBM-PC,

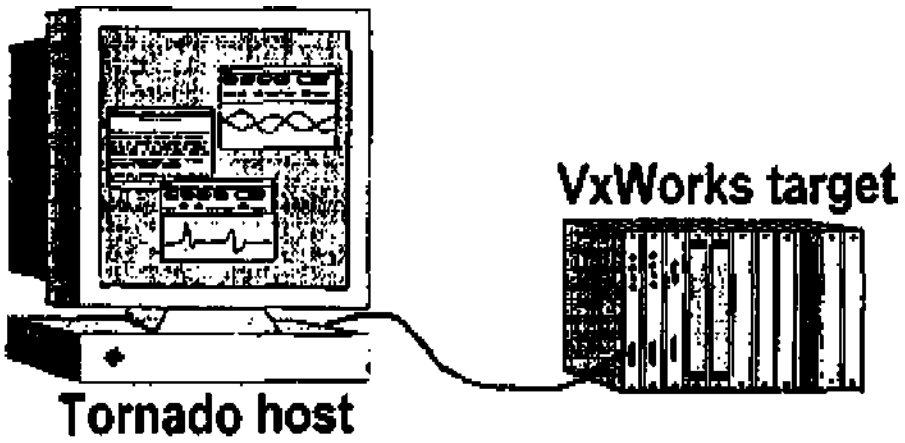
работающие в среде Windows, или рабочие станции SUN, HP, DEC, IBM RS/6000 с операционными системами типа UNIX. При выполнении отладки VxWorks, которая инсталлируется на отлаживаемой целевой системе, работает совместно с интегрированной средой разработки Tornado, функционирующей на инструментальном компьютере (рис. 3).

VxWorks имеет иерархическую организацию, нижним уровнем которой служит микроядро РВ, выполняющее базовые функции планирования задач и управления их коммуникацией-синхронизацией. Все остальные функции - управление памятью, вводом-выводом, сетевым обменом и другие, реализуются дополнительными модулями. Микроядро с минимальным набором модулей занимает 20...40 Кбайт памяти. Для встроенных систем, имеющих жесткие ограничения на объем памяти, разработано редуцированное ядро Wind Stream, которое требует для работы всего 8 Кбайт ПЗУ и 2 Кбайт ОЗУ.

Для реализации графических приложений используется система графического интерфейса VX-Windows. В тех случаях, когда ограниченный объем памяти целевой системы не позволяет использовать VX-Windows, предлагается графическая библиотека RTGL, которая содержит базовые графические примитивы, наборы шрифтов и цветов, драйверы типовых устройств ввода и графических контроллеров. В состав VxWorks входят также различные средства поддержки разнообразных сетевых протоколов: X.25, ISDN, ATM, SS7, Frame Relay и ряда других.

Специальные средства отладки в реальном масштабе времени обеспечивают трассировку заданных событий и их накопление в буферной памяти для последующего анализа. Трассировку системных событий выполняет динамический анализатор WindView, который работает аналогично логическому анализатору, отображая на экране временные диаграммы переключения задач, записи в очередь сообщений, установки программных светофоров и другие процессы. Монитор данных StethoScope позволяет анализировать динамическое изменение пользовательских и системных переменных, включая в себя также профилировщик процедур.

В составе VxWorks имеется пакет программ BSP для постановки данной ОСРВ на ряд плат развития, включая SBC фирмы Motorola, что позволяет конфигурировать таким образом целевую систему для конкретного приложения. Для комплексной отладки целевых систем VxWorks обеспечивает интерфейс со схемными эмуляторами (например, типа 64700 фирмы Hewlett-Packard) и эмуляторами ПЗУ (например, Net ROM фирмы XLNT Designs).



Инструментальный компьютер Целевая система

Рисунок 3. Аппаратно-программный комплекс программирования-отладки систем на базе ОСРВ VxWorks и интегрированной среды Tornado

Симулятор VxSim позволяет моделировать на инструментальном компьютере многозадачную среду VxWorks и интерфейс с целевой системой. Он позволяет разрабатывать и отлаживать программное обеспечение без подключения целевой системы. Среда VxWorks обеспечивает также возможности программирования мультипроцессорных систем.

Для поддержки программирования предлагается интегрированная среда разработки Tornado, в состав которой входит VxWorks 5.3 - ядро РВ и системные библиотеки, средства программирования C/C++ Toolkit, высокоуровневый отладчик CrossWind и ряд других средств. Пакет C/C++ Toolkit содержит компиляторы GNU C/C++ фирмы Cygnus Support. Отладчик CrossWind является расширенной версией отладчика GDB фирмы Cygnus Support. Он имеет графический пользовательский интерфейс и поддерживает отладку как на прикладном, так и на системном уровне. Дополнительные средства среды Tornado обеспечивают управление процессом отладки, визуализацию состояния целевой системы, другие сервисные функции.

Tornado может использоваться совместно с VX-Windows, WindView, StethoScope, VxSim и рядом других средств из состава VxWorks.

Характерной особенностью среды Tornado является ее открытая архитектура, которая позволяет пользователю подключать собственные специализированные инструментальные средства и расширять возможности стандартных. Открытость реализована с помощью прикладных программных интерфейсов API, которые дают возможность различным программным продуктам обмениваться между собой данными на инструментальном компьютере и взаимодействовать с VxWorks, установленной на целевой системе.

ОСРВ VxWorks вместе с интегрированной средой Tornado является мощным средством реализации целевых систем, работающих в условиях жестких ограничений на объем используемой памяти и время отклика на внешние события. Подробную информацию по VxWorks и сопутствующим

программным продуктам можно получить в сети Интернет по адресу http:/www. wrs.com.

Лекция 3.7. Сетевая операционная система реального времени QNX

1. Принципы построения СРВ QNX.
2. Архитектура системы QNX.
3. Основные механизмы QNX для организации распределенных вычислений.

1. Принципы построения СРВ QNX

Операционная система QNX является мощной операционной системой, позволяющей проектировать сложные программные системы, работающие в реальном времени как на одном компьютере, так и в локальной вычислительной сети. Встроенные средства операционной системы QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети. Основным языком программирования в системе является язык С. Основная операционная среда соответствует стандартам POSIX-интерфейса. Это позволяет с небольшими доработками перенести необходимое накопленное программное обеспечение в среду операционной системы QNX для организации их работы в среде распределенной обработки.

ОС QNX является сетевой, мультизадачной, многопользовательской (многотерминальной) и масштабируемой. С точки зрения пользовательского интерфейса и API она похожа на UNIX. Однако QNX - это не версия UNIX. QNX была разработана канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США. Причем эта система построена на совершенно других архитектурных принципах, отличных от принципов, использованных при создании ОС UNIX.

QNX была первой коммерческой ОС, построенной на принципах микроядра и обмена сообщениями. Система реализована в виде совокупности независимых (но взаимодействующих через обмен сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид сервиса. Эти идеи позволили добиться нескольких важнейших преимуществ:

Предсказуемость, означающая ее применимость к задачам жесткого реального времени. QNX является операционной системой, которая дает полную гарантию в том, что процесс с наивысшим приоритетом начнет выполняться практически немедленно, и что критическое событие (например, сигнал тревоги) никогда не будет потеряно. Ни одна версия UNIX не может достичь подобного качества, поскольку код ядра слишком велик. Любой системный вызов из обработчика прерывания в UNIX может привести к непредсказуемой задержке (то же самое касается Windows NT).

Масштабируемость и эффективность, достигаемые оптимальным использованием ресурсов и означающие ее применимость для встроенных систем. Драйверы и менеджеры можно запускать и удалять (кроме файловой системы) динамически, из командной строки. Вы можете иметь только тот сервис, который вам реально нужен, причем это не требует серьезных усилий и не порождает проблем.

Расширяемость и надежность одновременно, поскольку написанный вами драйвер не нужно компилировать в ядро. Менеджеры ресурсов (сервис логического уровня) работают в кольце защиты 3, и возможно добавлять свои, не опасаясь за систему. Драйверы работают в кольце с уровнем привилегий 1 и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать.

Быстрый сетевой протокол FLEET, прозрачный для обмена сообщениями, автоматически обеспечивающий отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа.

Компактная графическая подсистема Photon, построенная на тех же принципах модульности, что и сама ОС, позволяет получить полнофункциональный графический интерфейс пользователя GUI, работающий вместе с POSIX-совместимой ОС всего в 4 Мбайт памяти, начиная с i80386 процессора.

Основными принципами, которые являются обязательными при реализации и создании ОСРВ являются:

Первым обязательным требованием к архитектуре ОСРВ является многозадачность. Очевидно, что варианты с псевдомногозадачностью (а точнее - не вытесняющая многозадачность) типа MS Windows 3.x или Novell NetWare неприемлемы, поскольку они допускают возможность блокировки или даже полного развала системы одним неправильно работающим процессом. Для предотвращения блокировок ОСРВ должна использовать квантование времени (то есть вытесняющую многозадачность).

Вторая проблема (организация надежных вычислений) может быть эффективно решена при полном использовании возможностей процессоров Intel 80386 и старше, что предполагает работу ОС в 32-разрядном режиме процессора. Для эффективного обслуживания прерываний ОС должна использовать алгоритм диспетчеризации, обеспечивающий вытесняющее планирование, основанное на приоритетах. Крайне желательны эффективная поддержка сетевых коммуникаций и наличие развитых механизмов взаимодействия между процессами, поскольку реальные технологические системы обычно управляются целым комплексом компьютеров и/или контроллеров. Важно, чтобы ОС поддерживала множественные потоки управления (не только мультипрограммный, но и мультизадачный режимы), а также симметричную мультипроцессорность.

При соблюдении всех перечисленных выше условий, ОС должна быть способна работать на ограниченных аппаратных ресурсах, поскольку одна из ее основных областей применения - это встроенные системы. К сожалению,

данное условие обычно реализуется путем урезания стандартных сервисных средств.

2. Архитектура системы QNX

QNX - это ОС реального времени, позволяющая эффективно организовать распределенные вычисления. В системе реализована концепция связи между задачами на основе сообщений, посылаемых от одной задачи к другой, причем задачи эти могут находиться как на одном и том же компьютере, так и на удаленных, но связанных локальной вычислительной сетью. Реальное время и концепция связи между процессами в виде сообщений оказывают решающее влияние на разрабатываемое для QNX программное обеспечение и на программиста, стремящегося с максимальной выгодой использовать преимущества системы.

Микроядро имеет объем и несколько десятков килобайт (в одной из версий - 10 Кбайт, в другой - менее 32 Кбайт), то есть это одно из самых маленьких ядер среди всех существующих операционных систем. В этом объеме помещаются:

. механизм передачи сообщений между процессами (IPC);

. редиректор прерываний;

. блок планирования выполнением задач;

. сетевой интерфейс для перенаправления сообщений (менеджер Net).

Механизм передачи межпроцессорных сообщений занимается пересылкой сообщений между процессами и является одной из важнейших частей операционной системы, так как все общения между процессами, в том числе и системными, происходит через сообщения. Сообщение в QNX - это последовательность байтов произвольной длины (0-65 535 байтов) произвольного формата. Протокол обмена сообщениями выглядит таким образом. Например, задача блокируется для ожидания сообщения. Другая же задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача разблокируется, обрабатывает сообщение и отвечает, разблокируя при этом вторую задачу.

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты. Это значит, что, с одной стороны, уменьшается вероятность повреждения сообщения в процессе передачи, а с другой - уменьшается объем оперативной памяти, необходимый для работы ядра. Кроме того, уменьшается число пересылок из памяти в память, что разгружает процессор. Особенностью процесса передачи сообщений является то, что в сети, состоящей из нескольких компьютеров под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов. Определены в QNX еще и два дополнительных метода передачи сообщений - **метод представителей** (Proxy) и **метод сигналов** (Signal).

Представители используются в случаях, когда процесс должен передать сообщение, но не должен при этом блокироваться на передачу. В таком случае вызывается функция **qnx_proxy_attach( )** и создается

представитель. Он накапливает в себе сообщения, которые должны быть доставлены другим процессам. Любой процесс, знающий идентификатор представителя, может вызвать функцию **Trig-ger( ),** после чего будет доставлено первое в очереди сообщение. Функция **Trigger( )** может вызываться несколько раз, и каждый раз представитель будет доставлять следующее сообщение. При этом представитель содержит буфер, в котором может храниться до 65 535 сообщений.

Как известно, сигналы уже давно используются и ОС UNIX. Система QNX поддерживает множество сигналов, совместимых с POSIX, большое количество сигналов, традиционно использовавшихся в UNIX (поддержка этих сигналов реализована для совместимости с переносимыми приложениями, и ни один из системных процессов QNX их не генерирует), а также несколько сигналов, специфичных для самой QNX. По умолчанию любой сигнал, полученный процессом, приводит к завершению процесса (кроме нескольких сигналов, которые по умолчанию игнорируются). Но процесс с приоритетом уровня «superuser» может защититься от нежелательных сигналов. В любом случае процесс может содержать обработчик для каждого возможного сигнала. Сигналы удобно рассматривать как разновидность программных прерываний.

Редиректор прерываний является частью ядра и занимается перенаправлением аппаратных прерываний в связанные с ними процессы. Благодаря такому подходу возникает один побочный эффект - с аппаратной частью компьютера работает ядро, оно перенаправляет прерывания процессам - обработчикам прерываний. Обработчики прерываний обычно встроены в процессы, хотя каждый из них исполняется асинхронно с процессом, в который он встроен. Обработчик исполнялся в контексте процесса, и имеет доступ ко всем глобальным переменным процесса. При работе обработчика прерываний прерывания разрешены, но обработчик приостанавливается только в том случае, если произошло более высокоприоритетное прерывание. Если это позволяется аппаратной частью, к одному прерыванию может быть подключено несколько обработчиков, и каждый из них получит управление при возникновении прерывания.

Этот механизм позволяет пользователю избежать работы с аппаратным обеспечением напрямую и тем самым избегать конфликтов между различными процессами, работающими с одним и тем же устройством. Для обработки сигналов от внешних устройств, чрезвычайно важно минимизировать время между возникновением события и началом непосредственной его обработки. Этот фактор является существенным в любой области применения, от работы терминальных устройств до обработки высокочастотных сигналов.

Блок планирования выполнения задач (диспетчер задач) занимается обеспечением многозадачности. В этой части ОС QNX предоставляет разработчику огромный простор для выбора той методики выделения ресурсов процессора задаче, которая обеспечит наиболее подходящие условия для критических приложений или обеспечит такие

условия для некритических приложений, что они выполнятся за разумное время, не мешая работе критических приложений.

К выполнению своих функций как диспетчера ядро приступает в следующих случаях:

. какой-либо процесс вышел из блокированного состояния;

. истек квант времени для процесса, владеющего CPU;

. работающий процесс прерван каким-либо событием.

Диспетчер выбирает процесс для запуска среди неблокированных процессов в порядке значении их приоритетов, которые располагаются в диапазоне от 0 (наименьший) до 31 (наибольший). Обслуживание каждого из процессов зависит от метода диспетчеризации, с которым он работает (уровень приоритета и метод диспетчеризации могут динамически меняться во время работы). В QNX существуют три метода диспетчеризации: **FIFO** (первым пришел - первым обслужен), **round-robin** (процессу выделяется определенный квант времени для работы) и **адаптивный**, который является наиболее используемым.

Первый наиболее близок к кооперативной многозадачности. То есть процесс выполняется до тех пор, пока он не перейдет в состояние ожидания сообщения, состояние ожидания ответа на сообщение или не отдаст управление ядру. При переходе в одно из таких состояний процесс помещается последним в очередь процессов с таким же уровнем приоритета, а управление передается процессу с наибольшим приоритетом.

Во втором варианте все происходит так же, как и в предыдущем, с той разницей, что период, в течение которого процесс может работать без перерыва, ограничивается неким квантом времени.

Процесс, работающий с адаптивным методом, в QNX ведет себя следующим образом:

1. Когда процесс полностью использовал выделенный ему квант времени, его приоритет снижается на 1, если в системе есть процессы с тем же уровнем приоритета, готовые к исполнению.

2. Если процесс с пониженным приоритетом остается не обслуженным в течение секунды, его приоритет увеличивается на 1.

Если процесс блокируется, ему возвращается оригинальное значение приоритета.

По умолчанию процессы запускаются в режиме адаптивной многозадачности. В этом же режиме работают все системные утилиты QNX. Процессы, работающие в разных режимах многозадачности, могут одновременно находиться в памяти и исполняться. Важный элемент реализации многозадачности — это приоритет процесса. Обычно приоритет процесса устанавливается при его запуске. Но есть дополнительная возможность, называемая «вызываемый клиентом приоритет». Как правило, она реализуется для серверных процессов (исполняющих запросы на какое-либо обслуживание). При этом приоритет процесса-сервера устанавливается только на время обработки запроса и становится равным приоритету процесса-клиента.

Сетевой интерфейс в системе QNX является неотъемлемой частью ядра. Он взаимодействует с сетевым адаптером через сетевой драйвер, но базовые сетевые сервисы реализованы на уровне ядра. При этом передача сообщения процессу, находящемуся на другом компьютере, ничем не отличается с точки зрения приложения от передачи сообщения процессу, выполняющемуся на том же компьютере. Благодаря такой организации сеть превращается в однородную вычислительную среду. При этом для большинства приложений не имеет значения, с какого компьютера они были запущены, на каком исполняются и куда поступают результаты их работы. Такое решение принципиально отличает QNX от остальных ОС, которые тоже имеют все необходимые средства для работы в сети, и делает системы, работающие под ее управлением, по-настоящему распределенными. Все сервисы QNX, не реализованные непосредственно в ядре, работают как стандартные процессы в полном соответствии с основными концепциями микроядерной архитектуры. С точки зрения операционной системы системные процессы ничем не отличаются от всех остальных, как и драйверы устройств. Единственное, что нужно сделать, написав новый драйвер устройства в QNX, чтобы он стал частью операционной системы, - это изменить конфигурационный файл системы так, чтобы драйвер запускался при загрузке.

3. Основные механизмы QNX для организации распределенных вычислений QNX является сетевой операционной системой и позволяет организовать эффективные распределенные вычисления. Для организации сети на каждой машине, называемой узлом, помимо ядра и менеджера процессов должен быть запущен менеджер Net. Менеджер Net не зависит от аппаратной реализации сети. Данная аппаратная независимость обеспечивается за счет использования сетевых драйверов. В QNX имеются драйверы для различных сетей, например Ethernet, Arcnet, Token Ring. Кроме этого, имеется возможность организации сети через последовательный канал или модем.

В QNX четвертой версии полностью реализовано встроенное сетевое взаимодействие «точка-точка». Например, находясь на машине А, можно скопировать файл с гибкого диска, подключенного к машине В, на жесткий диск, подключенный к машине С. По существу, сеть из машин QNX действует как один мощный компьютер. Любые ресурсы (модемы, диски, принтеры) могут быть добавлены к системе простым их подключением к любой машине в сети. QNX поддерживает одновременную работу в сетях Ethernet, Arcnet, Serial и Token Ring, обеспечивает более чем один-единственный путь для коммуникации, а также балансировку нагрузки в сетях. Если кабель или сетевая плата выходит из строя таким образом, что связь через эту сеть прекращается, то система будет автоматически перенаправлять данные через другую сеть. Это происходит в режиме «on-line», предоставляя пользователю автоматическую сетевую избыточность и увеличивая скорость коммуникаций во всей системе.

Каждому узлу в сети соответствует уникальный целочисленный идентификатор - логический номер узла. Любой поток в сети QNX имеет прозрачный доступ (при наличии достаточных привилегий) ко всем ресурсам сети, то же самое относится и к взаимодействию потоков. Для взаимодействия потоков, находящихся на разных узлах сети, используются те же самые вызовы ядра, что и для потоков, выполняемых на одном узле. В том случае, если потоки находятся на разных узлах сети, ядро переадресует запрос менеджеру сети. Для организации обмена в сети используется надежный и эффективный протокол транспортного уровня FLEET. Каждый из узлов может принадлежать одновременно нескольким QNX-сетям. В том случае если сетевое взаимодействие может быть реализовано несколькими путями, для передачи выбирается незагруженная и более скоростная сеть.

Сетевое взаимодействие является узким местом в большинстве операционных систем и обычно создает значительные проблемы для систем реального времени. Для того чтобы обойти это препятствие, разработчики QNX создали собственную специальную сетевую технологию FLEET и соответствующий протокол транспортного уровня FTL (FLEET transport layer). Этот протокол не базируется ни на одном из распространенных сетевых протоколов типа IPX или NetBios и обладает рядом качеств, которые делают его уникальным. Основные его качества зашифрованы в аббревиатуре FLEET, которая расшифровывается следующим образом.

Благодаря этой технологии сеть компьютеров с QNX фактически можно представлять как один виртуальный суперкомпьютер. Все ресурсы любого из узлов сети автоматически доступны другим. Это значит, что любая программа может быть запущена на любом узле, при этом ее входные и выходные потоки могут быть направлены на любое устройство на любых других узлах.

Например, утилита make в QNX автоматически распараллеливает компиляцию пакетов из нескольких модулей на все доступные узлы сети, а затем собирает исполняемый модуль по мере завершений компиляции на узлах. Специальный драйвер, входящий в комплект поставки, позволяет использовать для сетевого взаимодействия любое устройство, с которым может быть ассоциирован файловый дескриптор, например последовательный порт, что открывает возможность для создания глобальных сетей.

Достигаются все эти удобства за счет того, что поддержка сети частично обеспечивается и микроядром (специальный код в его составе позволяет QNX фактически объединять все микроядра в сети в одно ядро). Разумеется, за такие возможности приходится платить тем, что мы не можем получить драйвер для какой-либо сетевой платы от кого-либо еще, кроме фирмы QSSL, то есть использоваться может только то оборудование, которое уже поддерживается. Однако ассортимент такого оборудования достаточно широк и периодически пополняется новейшими устройствами.

Fault-Tolerant Networking	QNX может одновременно использовать несколько физических сетей. При выходе из строя любой из них данные будут автоматически перенаправлены «на лету» через другую сеть
Load- Balancing on the Fly	При наличии нескольких физически соединений QNX автоматически распараллеливает передачу пакетов по соответствующим сетям
Efficient. Performance	Специальные драйверы, разрабатываемые фирмой QSSL для широкого спектра оборудования, позволяют с максимальной эффективностью использовать сетевое оборудование
Extensable Architecture	Любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов
Transparent Distributed Processing	Благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложении для того, чтобы они могли взаимодействовать через сеть

Когда ядро получает запрос на передачу данных процессу, находящемуся на удаленном узле, оно переадресовывает этот запрос менеджеру Net, в подчинении которого находятся драйверы всех сетевых карт. Имея перед собой полную картину состояния всего сетевого оборудования, менеджер Net может отслеживать состояние каждой сети и динамически перераспределять нагрузку между ними. В случае, когда одна из сетей выходит из строя, информационный поток автоматически перенаправляется в другую доступную сеть, что важно при построении высоконадежных систем. Кроме поддержки своего собственного протокола, Net обеспечивает передачу пакетов TCP/IP, SMB и многих других, используя то же сетевое оборудование. Производительность QNX в сети приближается к производительности аппаратного обеспечения.

При проектировании системы реального времени, как правило, необходимо обеспечить одновременное выполнение нескольких приложений. В QNX/Neutrino параллельность выполнения достигается за счет использования потоковой модели POSIX, в которой процессы в системе представляются в виде совокупности потоков. Поток является минимальной единицей выполнения и диспетчеризации для ядра Neutrino, процесс определяет адресное пространство для потоков. Каждый процесс состоит минимум из одного потока. QNX предоставляет богатый набор функций для синхронизации потоков. В отличие от потоков само ядро не подлежит диспетчеризации. Код ядра исполняется только в том случае, когда какой-нибудь поток вызывает функцию ядра или при обработке аппаратного прерывания.

QNX базируется на концепции передачи сообщений. Передачу сообщений, а также их диспетчеризацию осуществляет ядро системы. Кроме

того, ядро управляет временными прерываниями. Выполнение остальных функций обеспечивается задачами-администраторами. Программа, желающая создать задачу, посылает сообщение администратору задач (модуль task) и блокируется для ожидания ответа. Если новая задача должна выполняться одновременно с порождающей ее задачей, администратор задач task создает ее и, отвечая, выдает порождающей задаче идентификатор (id) созданной задачи. В противном случае никакого сообщения не посылается до тех пор, пока новая задача не закончится сама по себе. В этом случае в ответе администратора задач будут содержаться конечные характеристики закончившейся задачи.

Сообщения отличаются количеством данных, которые передаются от одной задачи точно к другой задаче. Данные копируются из адресного пространства первой задачи в адресное пространство второй, и выполнение первой задачи приостанавливается до тех пор, пока вторая задача не вернет ответное сообщение. В действительности обе задачи кратковременно взаимодействуют во время выполнения передачи. Ничто, кроме длины сообщения (максимальная длина 65 535 байт), не заботит QNX при передаче сообщения. Существует несколько протоколов, которые могут быть использованы для сообщений.

Основные операции над сообщениями - это послать, получить и ответить, а также несколько их вариантов для обработки специальных ситуаций. Получатель всегда идентифицируется своим идентификатором задачи, хотя существуют способы ассоциировать имена с идентификатором задачи. Наиболее интересные варианты операций включают в себя возможность получать (копировать) только первую часть сообщения, а затем получать оставшуюся часть такими кусками, какие потребуются. Это может быть использовано для того, чтобы сначала узнать длину сообщения, а затем динамически распределить принимающий буфер. Если необходимо задержать ответное сообщение до тех пор, пока не будет получено и обработано другое сообщение, то чтение первых нескольких байт дает вам компактный «обработчик», через который позже можно получить доступ ко всему сообщению. Таким образом, ваша задача предохраняется от того, чтобы хранить в себе большое количество буферов.

Другие функции позволяют программе получать сообщения только тогда, когда она уже ожидает их приема, а не блокироваться до тех пор, пока, не прибудет сообщение транслировать сообщение к другой задаче без изменения идентификатора передатчика. Задача, которая транслировала сообщение, в транзакции невидима.

Кроме этого, QNX обеспечивает объединение сообщений в структуру данных, называемую очередью. Очередь - это область данных в третьей, отдельной задаче, которая временно принимает передаваемое сообщение и немедленно отвечает передатчику. В отличие от стандартной передачи сообщений, передатчик немедленно освобождается для того, чтобы продолжить свою работу. Задача администратора очереди хранить в себе сообщение до тех пор, пока приемник не будет готов прочитать его. Это он

делает путем запроса сообщения у администратора очереди. Любое количество сообщений (ограничено только возможностью памяти) может храниться в очереди. Они хранятся и передаются в том порядке, в котором были приняты. Может быть создано любое количество очередей. Каждая очередь идентифицируется своим именем.

Помимо сообщений и очередей в QNX для взаимодействия задач и организации распределенных вычислений имеются так называемые порты, которые позволяют формировать сигнал одного конкретного условия, и механизм исключений.

Порт подобен флагу, известному всем задачам на одном и том же узле (но не на различных узлах). Он имеет два состояния, которые могут трактоваться как «присоединить» и «освободить», хотя пользователь может использовать свою интерпретацию; например, «занят» и «доступен». Порты используются для быстрой простой синхронизации между задачей и обработчиком прерываний устройства. Они нумеруются от нуля до максимума 32 (на некоторых типах узлов возможно и больше). Первые 20 номеров зарезервированы для использования операционной системой.

С портом могут быть выполнены три операции:

- присоединить порт;
- отсоединить порт;
- послать сигнал в порт.

Одновременно к порту может быть присоединена только одна задача Если другая задача попытается «отсоединиться» от того же самого порта, то произойдет отказ при вызове функции, и управление вернется к задаче, которая в настоящий момент присоединена к этому порту. Это самый быстрый способ обнаружить идентификатор другой задачи, подразумевая, что задачи могут договориться использовать один номер порта. Все рассматриваемые задачи должны находиться на одном и том же узле. При работе нескольких узлов специальные функции обеспечивают большую гибкость и эффективность.

Любая задача может посылать сигнал в любой порт независимо от того, была ли она присоединена к нему или нет (предпочтительно, чтобы не была присоединена). Сигнал подобен не блокирующей передаче пустого сообщения. То есть передатчик не приостанавливается, а приемник не получает какие-либо данные, он только отмечает, что конкретный порт изменил свое состояние.

Задача, присоединенная к порту, может ожидать прибытия сигнала или может периодически читать порт. QNX хранит информацию о сигналах, передаваемых в каждый порт, и уменьшает счетчик после каждой операции «приема» сигнала («чтение» возвращает счетчик и устанавливает его в ноль). Сигналы всегда принимаются перед сообщениями, давая им тем самым больший приоритет над сообщениями. В этом смысле сигналы часто используются обработчиками прерываний для того, чтобы оповестить задачу о внешних (аппаратных) событиях (действительно, обработчики прерываний

не имеют возможности посылать сообщения и должны использовать сигналы).

В отличие от описанных выше методов, которые строго синхронизируются, «исключения» обеспечивают асинхронное взаимодействие. То есть исключение может прервать нормальное выполнение потока задачи. Они, таким образом, являются аварийными событиями. QNX резервирует для себя 16 исключений для того, чтобы оповещать задачи о прерывании с клавиатуры, нарушении памяти и подобных необычных ситуациях. Остальные 16 исключений могут быть определены и использованы прикладными задачами.

Системная функция может быть вызвана для того, чтобы позволить задаче управлять своей собственной обработкой исключений, выполняя свою собственную внутреннюю функцию во время возникновения исключения.

Функция исключения задачи вызывается асинхронно операционной системой, а не самой задачей. Вследствие этого исключения могут иметь сильнодействующее побочное влияние на операции (например, передачу сообщений), которые выполняются в это же время. Обработчики исключений должны быть написаны очень аккуратно.

Одна задача может установить одно или несколько исключений на другой задачей. Они могут быть комбинацией системных исключений (определенных выше) и исключений, определяемых приложениями, что обеспечивает другие возможности для межзадачного взаимодействия.

Благодаря такому свойству QNX, как возможность обмена посланиями между задачами и узлами сети, программы не заботятся о конкретном размещении ресурсов в сети. Это свойство придает системе необычную гибкость. Так, узлы могут произвольно добавляться и изыматься из системы, не затрагивая системные программы. QNX приобретает эту конфигурационную независимость благодаря применению концепции о «виртуальных» задачах. У виртуальных задач непосредственный код и данные, будучи на одном из удаленных узлов, возникают и ведут себя так, как если бы они были локальными задачами какого-то узла со всеми атрибутами и привилегиями. Программа, посылающая сообщение в сети, никогда не посылает его точно. Сначала она открывает «виртуальную цепочку». Виртуальная цепочка включает все виртуальные задачи, связанные между собой. На обоих концах такой связи имеются буферы, которые позволяют хранить самое большое послание из тех, которые цепочка может нести в данном сеансе связи. Сетевой администратор помещает в эти буферы все сообщения для соединенных задач. Виртуальная задача, таким образом, занимает всего лишь пространство, необходимое для буфера и входа в таблицу задач. Чтобы открыть связь, необходимо знать идентификатор узла и задачи, с которой устанавливается связь. Для этого необходимо знать идентификатор задачи администратора, ответственного за данную функцию, или глобальное имя сервера. Не раскрывая здесь подробно механизм обмена посланиями, добавим, что наша задача может вообще выполняться на другом узле, где имеется более совершенный процессор.

ТЕМА 4. ОСОБЕННОСТИ ПРОГРАММИРОВАНИЯ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Лекция 4.1. Методы программирования в реальном времени.

1. Последовательное программирование и программирование задач реального времени

2. Среда программирования.

3. Структура программы реального времени.

4. Параллельное программирование, мультипрограммирование и многозадачность.

1. Последовательное программирование и программирование задач реального времени

Программа представляет собой описание объектов - констант и переменных - и операций, совершаемых над ними. Таким образом, программа - это чистая информация. Ее можно записать на какой-либо носитель, например на бумагу или на дискету.

Программы можно создавать и анализировать на нескольких уровнях абстракции (детализации) с помощью соответствующих приемов формального описания переменных и операций, выполняемых на каждом уровне. На самом нижнем уровне используются непосредственное описание - для каждой переменной указывается ее размер и адрес в памяти. На более высоких уровнях переменные имеют абстрактные имена, а операции сгруппированы в функции или процедуры. Программист, работающий на высоком уровне абстракции, не должен думать о том, по каким реальным адресам памяти хранятся переменные, и о машинных командах, генерируемых компилятором.

Последовательное программирование (sequential programming) является наиболее распространенным способом написания программ. Понятие "последовательное" подразумевает, что операторы программы выполняются в известной последовательности один за другим. Целью последовательной программы является преобразование входных данных, заданных в определенной форме, в выходные данные, имеющие другую форму, в соответствии с некоторым алгоритмом - методом решения (рис. 1).

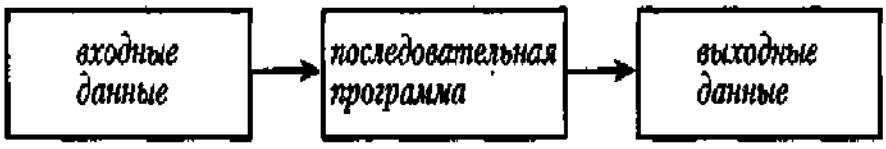


Рисунок 1. - Обработка данных последовательной программой

Таким образом, последовательная программа работает как фильтр для исходных данных. Ее результат и характеристики полностью определяются входными данными и алгоритмом их обработки, при этом временные показатели играют, как правило, второстепенную роль. На результат не влияют ни инструментальные (язык программирования), ни аппаратные (быстродействие ЦП) средства: от первых зависят усилия и время, затраченные на разра-

ботку и характеристики исполняемого кода, а от вторых - скорость выполнения программы, но в любом случае выходные данные будут одинаковыми.

Программирование в реальном времени (real-time programming) отличается от последовательного программирования - разработчик программы должен постоянно иметь в виду среду, в которой работает программа, будь то контроллер микроволновой печи или устройство управления манипулятором робота. В системах реального времени внешние сигналы, как правило, требуют немедленной реакции процессора. В сущности, одной из наиболее важных особенностей систем реального времени является время реакции на входные сигналы, которое должно удовлетворять заданным ограничениям.

Специальные требования к программированию в реальном времени, в частности необходимость быстро реагировать на внешние запросы, нельзя адекватно реализовать с помощью обычных приемов последовательного программирования. Насильственное последовательное расположение блоков программы, которые должны выполняться параллельно, приводит к неестественной запутанности результирующего кода и вынуждает связывать между собой функции, которые, по сути, являются самостоятельными. В большинстве случаев применение обычных приемов последовательного программирования не позволяет построить систему реального времени. В таких системах независимые программные модули или задачи должны быть активными одновременно, то есть работать параллельно, при этом каждая задача выполняет свои специфические функции. Такая техника известна под названием параллельного программирования (concurrent programming). В названии делается упор на взаимодействие между отдельными программными модулями. Параллельное исполнение может осуществляться на одной или нескольких ЭВМ, связанных распределенной сетью.

Программирование в реальном времени представляет собой раздел мультипрограммирования, который посвящен не только разработке взаимосвязанных параллельных процессов, но и временным характеристикам системы, взаимодействующей с внешним миром.

Между программами реального времени и обычными последовательными программами, с четко определенными входом и выходом, имеются существенные различия. Перечислим отличия программ реального времени от последовательных программ:

1. Логика исполнения программы определяется внешними событиями.

2. Программа работает не только с данными, но и с сигналами, поступающими из внешнего мира, например, от датчиков.

3. Логика развития программы может явно зависеть от времени.

4. Жесткие временные ограничения. Невозможность вычислить результат за определенное время может оказаться такой же ошибкой, как и неверный результат ("правильный ответ, полученный поздно - это неверный ответ").

5. Результат выполнения программы зависит от общего состояния системы, и его нельзя предсказать заранее.

6. Программа, как правило, работает в многозадачном режиме. Соответственно, необходимы процедуры синхронизации и обмена данными между процессами.

7. Исполнение программы не заканчивается по исчерпании входных данных - она всегда ждет поступления новых данных.

Важность фактора времени не следует понимать как требование высокой скорости исполнения программы. Скорость исполнения программы реального времени должна быть достаточной для того, чтобы в рамках установленных ограничений реагировать на входные данные и сигналы и вырабатывать соответствующие выходные величины. "Медленная" система реального времени может великолепно управлять медленным процессом. Поэтому скорость исполнения программ реального времени необходимо рассматривать относительно управляемого процесса или необходимой скорости. Типичные приложения автоматизации производственных процессов требуют гарантированное время ответа порядка 1 мс, а в отдельных случаях - порядка 0.1 мс. При программировании в реальном времени особенно важными является эффективность и время реакции программ. Соответственно, разработка программ тесно связана с параметрами операционной системы, а в распределенных системах - и локальной сети.

Особенности программирования в реальном времени требуют специальной техники и методов, не использующихся при последовательном программировании, которые относятся к влиянию на исполнение программы внешней среды и временных параметров. Наиболее важными из них являются перехват прерываний, обработка исключительных (нештатных) ситуаций и непосредственное использование функций операционной системы (вызовы ядра из прикладной программы, минуя стандартные средства). Помимо этого при программировании в реальном времени используются методика мультипрограммирования и модель "клиент-сервер", поскольку отдельный процесс или поток обычно выполняют только некоторую самостоятельную часть всей задачи.

2. Среда программирования

Рассмотрим среду, в которой исполняются программы. Среда выполнения может варьироваться от мини-, персональных и одноплатных микрокомпьютеров и локальных шин, связанных с окружающей средой через аппаратные интерфейсы, до распределенных систем "клиент-сервер" с централизованными базами данных и доступом к системе высокопроизводительных графических рабочих станций. В комплексной системе управления промышленными и технологическими процессами может одновременно использоваться все перечисленное оборудование.

Разнообразие аппаратной среды отражается и в программном обеспечении, которое включает в себя как программы, записанные в ПЗУ, так и комплексные операционные системы, обеспечивающие разработку и исполнение программ. В больших системах создание и исполнение программ осуществляются на одной и той же ЭВМ, а в некоторых случаях даже в одно время. Небольшие системы могут не иметь средств разработки, и программы

для них должны создаваться на более мощных ЭВМ с последующей загрузкой в исполняющую систему. То же касается и микропрограмм, "зашитых" в ПЗУ оборудования производителем (firmware), - они разрабатываются на ЭВМ, отличной от той, на которой исполняются.

Первой задачей программиста является ознакомление с программной средой и доступными инструментальными средствами. Проблемы, с которыми приходится сталкиваться, начинаются, например, с типа представления данных в аппаратуре и программах, поскольку в одних системах применяется прямой, а в других - инверсный порядок хранения бит или байт в слове (младшие байты хранятся в старших адресах). Таких тонкостей очень много, и опытный программист знает, как отделить общую структуру данных и код от технических деталей реализации в конкретной аппаратной среде.

Важно как можно раньше выяснить функции, обеспечиваемые имеющейся средой, и возможные альтернативы. Например, микропроцессор Motorola 68000 имеет в своем наборе команд инструкцию **test_and_set**, и поэтому связь между задачами может осуществляться через общие области памяти. Операционная система VAX/VMS поддерживает почтовые ящики, и синхронизировать процессы можно с помощью механизма передачи сообщений.

В UNIX и других операционных системах связь между процессами наиболее удобно осуществлять через каналы. При разработке программ для среды UNIX следует стремиться, с одной стороны, максимально эффективно использовать ее особенности, например стандартную обработку входных и выходных данных, а с другой - обеспечить переносимость между разными версиями UNIX.

Из-за того, что многозадачные системы и системы реального времени разрабатываются коллективами программистов, необходимо с самого начала добиваться ясности, какие методы и приемы используются.

Структурирование аппаратных и программных ресурсов, то есть присвоение адресов на шине и приоритетов прерываний для интерфейсных устройств, имеет важное значение. Неправильный порядок распределения ресурсов может привести к тупиковым ситуациям. Определение аппаратных адресов и относительных приоритетов прерываний не зависит от разрабатываемой программы. Поэтому оно должно быть произведено на ранней стадии и зафиксировано в техническом задании. Его не следует откладывать до момента непосредственного кодирования, так как в этом случае неизбежны конфликты между программными модулями и возникает риск тупиковых ситуаций.

Правильным практическим решением является использование в программе только логических имен для физического оборудования и его параметров и таблиц соответствия между ними и реальными физическими устройствами. При этом изменение адреса шины или приоритета устройства требует не модификации, а в худшем случае только новой компиляции программы. Разумно также использовать структурированное и организационно оформленное соглашение о наименовании системных ресурсов и программ-

ных переменных. То же относится и к наименованию и определению адресов удаленных устройств в распределенных системах.

Программы следует строить по принципам, применяемым в операционных системах, - на основе модульной и многоуровневой структуры, поскольку это существенно упрощает разработку сложных систем. Должна быть определена спецификация отдельных модулей, начиная с интерфейсов между аппаратными и программными компонентами системы. К основной информации об интерфейсах относится и структура сообщений, которыми будут обмениваться программные модули. Это не означает, что изменения в определении интерфейсов не могут вводиться после начала разработки программы. Но чем позже они вносятся, тем больше затрат потребует изменение кода, тестирование и т. д. С другой стороны, следует быть готовым к тому, что некоторые изменения спецификаций все равно будут происходить в процессе разработки программы, поскольку продвижение в работе позволяет лучше увидеть проблему.

Следует принимать во внимание эффективность реализации функций операционной системы. Нельзя считать, что способ, которым в операционной системе реализованы те или иные услуги, дан раз и навсегда. Для проверки того, насколько хорошо удовлетворяются временные ограничения, желательно провести оценку, например с помощью эталонных тестовых программ. Если результаты тестов неприемлемы, то одним из решений может быть разработка программ, замещающих соответствующие стандартные модули операционной системы. Такое решение требует очень осторожного и дифференцированного подхода, в частности замещение может выполняться не всегда, а только для определенных процессов.

3. Структура программы реального времени

Разработка программы реального времени начинается с анализа и описания задачи. Функции системы делятся на простые части, с каждой из которых связывается программный модуль.

Например, задачи для управления движением манипулятора робота можно организовать следующим образом:

– считать с диска описание траекторий;

– рассчитать следующее положение манипулятора (опорное значение);

– считать с помощью датчиков текущее положение;

– вычислить необходимый сигнал управления;

– выполнить управляющее действие;

– проверить, что опорное значение и текущее положение совпадают в пределах заданной точности;

– получить данные от оператора;

– остановить робота в случае нештатной ситуации (например, сигнал прерывания от аварийной кнопки).

Принципиальной особенностью программ реального времени является постоянная готовность и отсутствие условий нормального, а не аварийного завершения. Если программа не исполняется и не обрабатывает

данные, она остается в режиме ожидания прерывания/события или истечения некоторого интервала времени. Программы реального времени - это последовательный код, исполняющийся в бесконечном цикле.

В каком-то месте программы есть оператор, приостанавливающий исполнение до наступления внешнего события или истечения интервала времени. Обычно программа структурируется таким образом, что оператор **end** никогда не достигается

while true do (*бесконечный цикл*) **begin** (*процедура обработки*) **wait** event at #2,28 (*внешнее прерывание*) (*код обработки*) … **end**; (*процедура обработки*)

end. (*выход из программы; никогда не достигается*)

При разработке каждого программного модуля должны быть четко выделены области, в которых происходит обращение к защищенным ресурсам, - критические секции. Вход и выход из этих областей координируется каким-либо методом синхронизации или межпрограммных коммуникаций, например с помощью семафоров. В общем случае, если процесс находится в критической секции, можно считать, что данные, с которыми он работает, не изменяются каким-либо другим процессом. Прерывание исполнения процесса не должно оказывать влияния на защищенные ресурсы. Это снижает риск системных ошибок.

Аналогичные предосторожности необходимо соблюдать и для потоков, порождаемых как дочерние процессы главного процесса. Разные потоки могут использовать общие переменные породившего их процесса, и поэтому программист должен решить, защищать эти переменные или нет.

Для гарантии живучести программы нештатные ситуации, которые могут блокировать или аварийно завершить процесс, должны своевременно распознаваться и исправляться - если это возможно - в рамках самой программы.

В системах реального времени различные процессы могут обращаться к общим подпрограммам. При простейшем решении эти подпрограммы связываются с соответствующими модулями после компиляции. При этом в памяти хранится несколько копий одной подпрограммы.

При другом подходе в память загружается лишь одна копия подпрограммы, но доступ к ней возможен из разных программ. Такие подпрограммы должны быть повторно входимыми - реентерабельными (reentrant), то есть допускать многократные вызовы и приостановку исполнения, которые не влияют друг на друга. Эти программы должны использовать только регистры процессора или память вызывающих процессов, то есть не иметь локальных переменных. В результате реентерабельный модуль, разделяемый несколькими процессами, можно прервать в любое время и продолжить с другой точки программы, поскольку он работает со стеком вызвавшего его процесса. Таким образом, реентерабельная процедура может оказаться одновременно в контексте нескольких различных процессов.

Эффективность исполнения является одним из наиболее важных параметров систем реального времени. Процессы должны выполняться быстро, и

часто приходится искать компромисс между ясностью и структурированностью программы и ее быстродействием. Жизненный опыт показывает, что если для достижения цели нужно чем-то пожертвовать, что обычно и делается. Не всегда возникает противоречие между структурностью и эффективностью, но если первое должно быть принесено в жертву второму, необходимо полностью документировать все принятые решения, иначе существенно осложняется дальнейшее сопровождение программы.

4. Параллельное программирование, мультипрограммирование и многозадачность

Программирование в реальном времени требует одновременного исполнения нескольких процессов или задач на одной ЭВМ. Эти процессы используют совместно ресурсы системы, но более или менее независимы друг от друга.

Мультипрограммирование (multiprogramming) или многозадачность (multitasking) есть способ одновременного исполнения нескольких процессов. Такого эффекта можно добиться как для одного, так и для нескольких процессоров: процессы исполняются либо на одном, либо на нескольких связанных между собой процессорах. В действительности многие современные вычислительные системы состоят из нескольких процессоров, связанных между собой либо сетью передачи данных, либо общей шиной.

Для записи параллельных процессов можно использовать следующую нотацию

cobegin
x := 1; x := 2; x := 3;
coend;
write (x);

Исполнение команд между ключевыми словами **cobegin** и **coend** происходит параллельно (рис. 2). Пара операторных скобок **cobegin-coend** приводит к генерации потоков в рамках многозадачной системы. Оператор **cobe-gin** не накладывает условий на относительный порядок исполнения отдельных процессов, а оператор **coend** достигается только тогда, когда все процессы внутри блока завершены. Если бы исполнение было последовательным, то окончательное значение переменной x было бы равно 3. Для параллельных процессов конечный результат однозначно предсказать нельзя; задачи выполняются, по крайней мере, с внешней точки зрения, в случайной последовательности. Поэтому окончательное значение x в приведенном примере может быть как 1, так и 2 или 3.

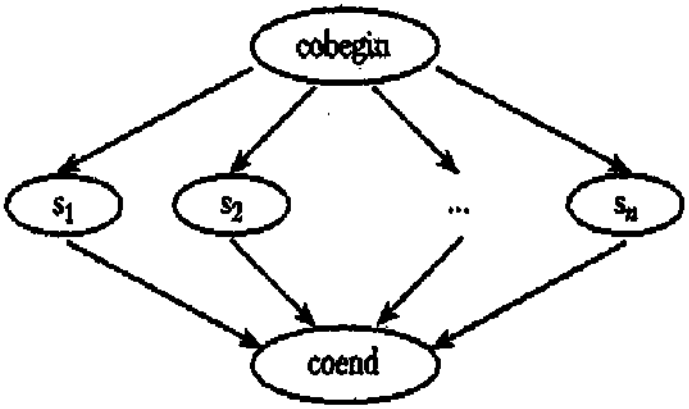


Рисунок 2. - Граф очередности для операторов **cobegin – coend**

Иногда в технической литературе термин "параллельное программирование" используется как синоним мультипрограммирования. Однако эти понятия несколько различаются по смыслу. Параллельное программирование -это абстрактный процесс разработки программ, который потенциально может исполняться параллельно, вне зависимости от программно-аппаратной среды. Иными словами, предполагается, что каждая задача реализуется на собственном виртуальном процессоре. С другой стороны, мультипрограммирование представляет собой практический способ исполнения нескольких программ на одном центральном процессоре или в распределенной вычислительной системе. Параллельное программирование более трудоемко, чем последовательное, поскольку способность человека следить за развитием связанных процессов, и исследовать их взаимодействие, ограничена.

Программирование в реальном времени основано на параллельном программировании и включает в себя технику повышения эффективности и скорости исполнения программ - управление прерываниями, обработку исключений и непосредственное использование ресурсов операционной системы. Кроме того, программы реального времени требуют специальных методов тестирования.

Лекция 4.2. Языки программирования реального времени

1. Требования к языкам программирования реального времени.
2. Языки разработки для систем реального времени.

1. Требования к языкам программирования реального времени Основными критериями при выборе языка для разработки приложения реального времени являются:

1. **Получение наивысшей производительности** приложения реального времени. Из этого требования вытекает, что язык должен быть компилируемого (как C, C++), а не интерпретируемого (как Java) типа, и для него должен существовать компилятор с высокой степенью оптимизации кода. Для современных процессоров качество компилятора особенно важно,

поскольку для них оптимизация может ускорять работу программы в несколько раз по сравнению с не оптимизированным вариантом, причем часто оптимизирующий компилятор может породить код более быстрый, чем написанный на ассемблере. Технологии оптимизации развиваются достаточно медленно и часто требуются годы на разработку высокоэффективного компилятора. Поэтому обычно для более старых и с более простой структурой языков имеются более качественные компиляторы, чем для достаточно молодых, и сложно устроенных языков.

2. **Получение доступа к ресурсам оборудования** либо посредством языковых конструкций, либо посредством имеющихся для выбранного языка библиотечных функций.

3. **Возможность вызова процедур, написанных на другом языке**, например, на языке ассемблера. Из этого требования вытекает, что последовательность вызова подпрограмм (механизм именования объектов, передачи аргументов и получения возвращаемого значения) должна быть документирована для выбранного языка.

4. **Переносимость приложения**, под которой обычно понимают, как возможность его скомпилировать другим компилятором, имеющимся на той же платформе, так и возможность его скомпилировать на другой платформе и/или другой операционной системе.

5. **Поддержка объектно-ориентированного подхода** стала в последнее время необходимостью, зачастую выходя в списке требований на первое место. Это объясняет использование языка Java в ОСРВ.

Программирование в реальном времени требует специальных средств, которые не всегда встречаются в обычных языках последовательного программирования. Язык или операционная система для программирования в реальном времени должны предоставлять следующие возможности:

– описание параллельных процессов;

– переключение процессов на основе динамических приоритетов, которые могут изменяться, в том числе и прикладными процессами;

– синхронизацию процессов;

– обмен данными между процессами;

– функции, связанные с часами и таймером, абсолютное и относительное время ожидания;

– прямой доступ к внешним аппаратным портам;

– обработку прерываний;

– обработку исключений.

Немногие языки обеспечивают все эти возможности. Большинство имеет лишь часть из них, хотя для определенных приложений этого оказывается достаточно. Некоторые компании разработали специальные языки для поддержки своих собственных аппаратных средств. Эти языки не претендуют на универсальность и ориентированы скорее на конкретные ЭВМ и их интерфейсы. Обычно они базируются на существующих языках - FORTRAN, BASIC - с расширениями, включающими функции реального времени, о чем свидетельствуют их названия типа "Process BASIC" и "Real-time

FORTRAN". Некоторые языки не поддерживают программирования в реальном времени в строгом смысле, но они легко расширяются, например C и C++.

В 1970-е годы широкую поддержку получила концепция единого переносимого многоцелевого языка программирования. В результате был разработан язык ADA. Его главная идея состоит в том, что среда программирования, то есть язык, должна быть полностью отделена от аппаратных средств. Программист не должен сталкиваться с деталями машинного уровня, а работать только в терминах абстрактных структур и типов данных.

Опыт показал не реалистичность такого подхода. Универсальные, сильно типизированные языки программирования гарантируют определенный уровень надежности программы, но в то же время ограничивают гибкость. Быстрое развитие технических средств, предъявляет новые требования, которые не могли быть предусмотрены в существующих языках, и многие программисты чувствуют ограничения, используя, не самые современные языки программирования. Цена надежности языка - сложность и громоздкость, а генерируемый при этом компилятором код - избыточен и малоэффективен. Открытый язык типа C, основанный на ограниченном количестве базовых идей, обладает большей гибкостью и предоставляет опытному программисту больше возможностей. Не существует наилучшего языка -для каждого приложения и среды необходимо подбирать свои средства и при этом учитывать квалификацию и предпочтения разработчиков.

2. Языки разработки для систем реального времени

Ассемблер. Обеспечивает получение наивысшей производительности, прямой доступ к оборудованию, возможность вызова любых процедур на других языках. Однако, приложения получаются не переносимыми, объектно-ориентированный подход отсутствует. Обычно ассемблер используется только для написания небольших и четко локализованных фрагментов приложения, таких, как обработчики прерываний, драйверы устройств, критические по времени исполнения секции.

Язык программирования ADA. Первым полным языком программирования в реальном времени является ADA. В середине 1970-х годов Министерство обороны США для сокращения расходов на разработку и сопровождение своих систем управления реального времени приняло решение ввести единый язык программирования в качестве альтернативы сотням использовавшихся тогда языков. В 1979 году министерство одобрило предложения, выдвинутые французской компанией Honeywell Bull. Язык назван в честь Августы Ады Байрон, графини Лавлейс (Augusta Ada Byron, Countess of Lovelace, 1815-1852), которую можно считать первым программистом в истории - она писала программы для аналитической машины (механического компьютера, который никогда не был построен), спроектированной английским изобретателем Чарльзом Бэббиджем (Charles Babbage).

Язык ADA является полной средой разработки программ с текстовым редактором, отладочными средствами, системой управлениями библиотеками и т.д. Спецификации ADA закреплены американским стандартом

ANSI/MIL-STD-1815A и включают средства контроля соответствия этому стандарту. Не допускаются диалекты языка - для сертификации компилятор должен правильно выполнить все эталонные тесты.

Структура языка ADA похожа на структуру языка Pascal, но его возможности значительно шире, в особенности применительно к системам реального времени. Процессу в ADA соответствует задача, которая выполняется независимо от других задач на выделенном виртуальном процессоре, то есть параллельно с другими задачами. Задачи могут быть связаны с отдельными прерываниями и исключениями, и работать как их обработчики.

Новым понятием, введенным в ADA, является пакет - модуль со своими собственными описаниями типов данных, переменных и подпрограмм, в котором явно указано, какие из программ и переменных доступны извне. Пакеты могут компилироваться отдельно с последующим объединением в один исполняемый модуль. Это средство поддерживает модульную разработку программ и создание прикладных библиотек. В начале 1990-х годов язык ADA был пополнен новыми функциями для объектно-ориентированного программирования и программирования в реальном времени.

Машинно-ориентированное программирование низкого уровня поддерживается ADA не достаточно эффективно - это следствие постулата, что все задачи должны решаться средствами высокого уровня. Например, для операций ввода/вывода в ADA используются прикладные пакеты с заранее определенными функциями для управления аппаратными интерфейсами и доступа к внешним данным.

Основным недостатком ADA является его сложность, которая делает язык трудным для изучения и применения. Существующие компиляторы являются дорогостоящими продуктами и требуют мощных процессоров. До сих пор ADA не получил ожидавшейся популярности, и сомнительно, что это когда-нибудь произойдет.

Языки C и C++. Язык программирования C, несмотря на отсутствие в нем многих средств, которые теоретики считают необходимыми для хорошего языка программирования, пользуется большим успехом, начиная с 1980-х годов и по настоящее время. Этот язык стал популярным для всех приложений, требующих высокой эффективности, в частности для программ реального времени. Для обычных микропроцессоров, используемых в системах управления, имеются C-компиляторы и системы разработки многих производителей. В промышленности существует явная тенденция к широкому применению языка C и операционной системы UNIX, которая сама написана на C, поскольку приложения, написанные на C, машинно-независимы и требуют не больших усилий для адаптации к работе в различной аппаратной среде.

Философией C является разбиение программ на функции. C - слаботипизированный язык и позволяет программисту делать почти все вплоть до манипуляции с регистрами и битами. Такая свобода делает язык небезопасным, поскольку компилятор не может проверить, являются ли подозрительные операции умышленными или нет. Небольшое количество заранее определенных функций и типов данных делает программы легко

переносимыми между разными системами. С поддерживает как хороший, структурированный, так и плохой стиль программирования, оставляя ответственность за качество разработки на программисте. Стиль программирования приобретает особое значения при сопровождении программ: плохо написанная и откомментированная программа на С - такая же загадка, как и ассемблерский код. Язык С регламентирован международным стандартом ISO 9899.

Язык С предпочтителен для написания программ с обращениями к функциям операционной системы, так как он обладает отличной совместимостью между логикой определения переменных и синтаксисом обращения к системе. Поскольку наиболее распространенные операционные системы в приложениях автоматического управления процессами основываются на UNIX, язык С является почти вынужденным выбором при разработке программ. Почти все примеры в современной технической литературе представлены на С.

С обеспечивает получение высокой производительности за счет хорошо разработанных оптимизирующих компиляторов, которые для современных процессоров часто дают код более эффективный, чем написанный на ассемблере. Язык С дает прямой доступ к оборудованию и возможность вызова процедур на других языках. Приложения получаются переносимыми (особенно, если ОСРВ поддерживают одинаковый стандарт, например POSIX), однако, объектно-ориентированный подход на уровне языковых конструкций отсутствует.

Язык С++ представляет собой значительно более мощный инструмент, чем С, на основе которого он создан. В С++ значительно улучшена абстракция данных с помощью понятия класса, похожего на абстрактный тип данных с четким разделением между данными и операциями. Классы С++ значительно легче использовать на практике, чем аналогичные понятия в других языках, поскольку С++ поддерживает объектно-ориентированное программирование и поэтапное уточнение типов данных.

Главным преимуществом языка С++ является его способность поддерживать разработку легко используемых библиотек программ. Программирование в реальном времени непосредственно в С++ не поддерживается, но может быть реализовано с помощью специально разработанных программных модулей и библиотек классов.

С++ включает язык С как подмножество и наследует все его положительные качества. С++ добавляет поддержку объектно-ориентированного подхода на уровне языковых конструкций.

Java. Как язык интерпретируемого типа, имеет очень низкую эффективность получаемого кода. Доступ к оборудованию и вызовы процедур на других языках - только посредством библиотечных функций (обычно написанных на С). Java обеспечивает наивысшую переносимость приложения на уровне двоичного кода и является объектно-ориентированным языком.

BASIC. Язык BASIC является простейшим среди языков программирования высокого уровня. Этот язык был создан в 1964 году для поддержки интерактивной разработки программ с удаленных терминалов. Из-за своей простоты BASIC часто критикуется опытными программистами, и несомненно, что этот язык не является хорошим средством для создания больших структурированных систем. С другой стороны, небольшие приложения на BASIC можно разработать значительно быстрее, чем на других языках. Кроме того, BASIC имеется почти на всех мини- и микрокомпьютерах.

Программа на BASIC может компилироваться, но чаще она интерпретируется, то есть каждая команда транслируется в машинные коды только в момент ее выполнения. BASIC удобен для разработки небольших прикладных задач в составе крупных систем, но его не следует использовать для приложений порядка 500-1000 строк или более. Тем не менее, BASIC является наилучшим средством для непрофессиональных программистов, которым требуется быстро решить частную задачу. Командные языки, основанные на BASIC, имеются во многих системах промышленной автоматики. Они применяются для написания простых программ управления без обращения к более сложным средствам программирования, требующим компиляции и загрузки.

FORTRAN. FORTRAN - это первый язык программирования высокого уровня, который, по-видимому, способствовал, более чем какой-либо другой язык, распространению и практическому применению ЭВМ. Выпущенный в 1957 году, он до сих пор широко используется, в особенности для математических вычислений. В целом FORTRAN имеет ограниченные возможности определения типа, весьма сложный способ работы с нечисловыми данными и не содержит многих важных функций языков реального времени, чтобы его серьезно рассматривать для этой цели. Новые версии FORTRAN заимствовали некоторые возможности из других языков и поддерживают более развитые структуры данных. В этом смысле различия между FORTRAN и другими языками сглаживаются.

Благодаря тому, что язык имеет устойчивое применение в научных приложениях, нередко данные в системах реального времени обрабатываются существующими FORTRAN-программами, а новые программы анализа и статистики пишутся на FORTRAN. В подобных случаях основной проблемой является координация передачи информации между базами данных реального времени и прикладными модулями, написанными на FORTRAN. Такая координация обычно выполняется операционной системой. FORTRAN не рекомендуется для написания драйверов устройств или модулей на уровне операционной системы, так как для этой цели лучше подходят другие языки.

Pascal и Modula-2. Pascal был разработан швейцарцем Николасом Вир-том (Niklaus Wirth) в 1971 году как дидактический язык для обучения хорошей технике программирования. Он быстро перерос свои первоначальные рамки и в настоящее время используется во множестве разнообразных приложений. Успех Pascal, как в случае BASIC, основан на распространении микро- и персональных компьютеров, на которых он

широко используется. Язык Modula-2 был разработан тем же автором в 1975 году специально для программирования встроенных промышленных и научных вычислительных систем реального времени. Pascal и Modula-2 весьма похожи по стилю и структуре, хотя Modula-2 обладает большим количеством функций и синтаксических конструкций.

В Pascal и Modula-2 предполагается, что программист постоянно остается в ограниченной среде, предоставляемой программой, что совсем не соответствует реальной практике. Гибкость их использования несколько выше, если некоторые программы для специальных приложений (драйверы устройств, обработчики прерываний) написаны на языке ассемблера. Оба языка поддерживают подключение внешних модулей на ассемблере. Pascal и Modula-2 являются хорошим средством для разработки встроенных систем, но не подходят для сложных приложений в распределенных компьютерных системах. Их ориентация на структуру делает программы хорошо читаемыми, что является существенным фактором для последующего сопровождения.

Языки четвертого поколения (CASE средства). Средства CASE (Computer Aided Software Engenering) получили широкое распространение при разработке приложений реального времени в силу большой сложности последних. Языки «четвертого поколения» представляют собой формализованный способ описания объектов, их свойств и взаимоотношений между собой. По этому формальному описанию «компилятор» строит текст приложения на языке более низкого уровня (обычно предоставляется выбор между C/C++/Java). Затем этот текст можно скомпилировать уже «обычным» компилятором. Поскольку можно добавлять фрагменты на языке более низкого уровня, то CASE средства наследуют все положительные свойства последнего.

Лекция 4.3. Программирование асинхронной и синхронной обработки данных

1. Обработка прерываний и исключений.
2. Программирование операций ожидания.
3. Внутренние подпрограммы операционной системы.
4. Приоритеты процессов и производительность системы.
5. Тестирование и отладка.

1. Обработка прерываний и исключений

Системы реального времени соединены с внешней средой (физический процесс) через аппаратные интерфейсы. Доступ к интерфейсам и внешним данным осуществляется либо по опросу, либо по прерыванию.

При опросе программа должна циклически последовательно проверять все входные порты на наличие у них новых данных, которые затем считываются и обрабатываются. Очередность и частота опроса определяют время реакции системы реального времени на входные сигналы. Опрос является простым, но неэффективным методом из-за повторяющихся проверок входных портов.

Получение данных по прерыванию происходит иначе. Интерфейсное устройство, получившее новые данные, привлекает внимание центрального процессора, посылая ему сигнал прерывания через системную шину. По отношению к текущему процессу прерывания являются асинхронными событиями, требующими немедленной реакции. Получив сигнал прерывания, процессор приостанавливает исполнение текущего процесса, сохраняет в стеке его контекст, считывает из таблицы адрес программы обработки прерывания и передает ей управление. Эта программа называется обработчиком прерывания. Другой вариант обработки прерываний заключается в том, что планировщик выбирает из очереди ожидания этого события или прерывания следующий процесс и переводит его в очередь готовых процессов.

Когда процессор передает управление обработчику прерываний, он обычно сохраняет только счетчик команд и указатель на стек текущего процесса. Обработчик прерываний должен сохранить во временных буферах или в стеке все регистры, которые он собирается использовать, и восстановить их в конце. Эта операция критична по времени и, как правило, требует запрета прерываний для того, чтобы избежать переключения процессов во время ее выполнения.

При управлении прерываниями время реакции должно быть как можно меньше. Оно представляет собой сумму времени, необходимого процессору, чтобы среагировать на прерывание (латентность прерывания), и времени, необходимого на переключение контекста до запуска обработчика прерываний. Типичная загрузка системы также играет определенную роль. Если система должна обслуживать много одновременных прерываний, вновь поступающие прерывания будут ждать в очереди, пока процессор не освободится.

Программы обработки прерывания должны быть предельно компактными (длина кода) и короткими (время выполнения). Если сложное действие, требующее большого расхода процессорного времени, например вычисления или доступ к базе данных, необходимо выполнить после возникновения прерывания, то его лучше вынести из обработчика прерывания в процесс. Программа обработки прерывания должна выполнять лишь минимально необходимые операции, например, считать входные данные, сформировать сообщение и передать другой программе, извещая ее, что произошло прерывание и требуется дальнейшая обработка. Хорошим стилем для обработчиков прерываний является использование реентерабельного кода. Это позволяет избежать конфликтов в случае, если прерывается сам обработчик и тот же код вызывается для обслуживания нового прерывания прежде, чем закончилась обработка предыдущего.

Реакция на исключения (exceptions) похожа на обработку прерываний. Исключениями называются нештатные ситуации, когда процессор не может правильно выполнить команду. Примером исключения является деление на ноль или обращение по несуществующему адресу. В англоязычной литературе для разных видов исключений применяются термины trap, fault, abort (не путать с "взаимным исключением" - mutual exclusion).

Обычно операционная система обрабатывает исключения, прекращая текущий процесс, и выводит сообщение, четко описывающее ситуацию, на устройство отображения, обычно монитор или принтер. Приемлемая при последовательной интерактивной многопользовательской обработке, внезапная остановка процесса в системах реального времени должна быть абсолютно исключена. Нельзя допустить, чтобы управляемые микропроцессором автопилот самолета или автоматическая тормозная система автомобиля (Automatic Braking System - ABS), внезапно прекратили работу из-за деления на ноль. В системах реального времени все возможные исключения должны анализироваться заранее с определением соответствующих процедур обработки.

Сложной проблемой при обработке исключений является проверка, что исключение не возникнет снова после того, как оно было обработано. Или иными словами, обработка исключений должна заниматься причиной, а не симптомами аномальной ситуации. Если исключение обработано некорректно, оно может возникнуть опять, заставляя процессор снова и снова переходить к модулю обработки. Например, обработчик деления на нуль должен проверять и изменять операнды, а не просто возобновлять исполнение с места, предшествующего ошибке, что приведет к бесконечному циклу.

Фактические адреса программных модулей известны только после их загрузки. При запуске системы в таблицу обработки прерываний записываются адреса памяти, куда загружаются обработчики, которые затем вызываются по ссылкам из этой таблицы.

2. Программирование операций ожидания

Процесс реального времени может явным образом ждать истечения некоторого интервала (относительное время) или наступления заданного момента (абсолютное время). Соответствующие функции обычно имеют следующий формат:

wait (n)

и

wait until (время)

где n - интервал в секундах или миллисекундах, а переменная "время" имеет формат часы, минуты, секунды, миллисекунды.

Когда выполняется одна из этих функций, операционная система помещает процесс в очередь ожидания. После истечения/наступления заданного времени процесс переводится в очередь готовых процессов.

Распространенный, но не лучший метод организации временной задержки - цикл, контроль системного времени в цикле занятого ожидания

repeat (*холостой ход*)

until (time = 12:00:00);

Как правило, подобные активные циклы ожидания представляют собой бесполезную трату процессорного времени, и их следует избегать. Однако имеются исключения. В системе, где аналого-цифровое преобразование занимает 20 мкс, а операция переключения процессов - 10

мкс, более экономно организовать ожидание на 20 мкс перед тем, как считать новые данные, чем начинать процедуру переключения процессов, неявно подразумеваемую "хорошей" операцией ожидания. Каждый случай требует индивидуального подхода - для этого обычно нужно хорошее знание системы и развитое чутье.

Важной особенностью процессов, запускаемых периодически, - например, фильтрация и алгоритмы регулирования, - является накопленная ошибка времени. Это связано с тем, что процесс из очереди ожидания события опять попадает в очередь, но уже готовых процессов и должен ждать некоторый случайный интервал времени прежде, чем получит управление (рис. 1а). Требуемое и фактическое время пробуждения процесса не совпадают. Ошибки ожидания накапливаются, если это время рассчитывается так новое время пробуждения = время начала ожидания + интервал По такому алгоритму работает холостой цикл "ждать 10 секунд". Накопленная временная ошибка представляет собой сумму времени, проведенного в очереди, и времени, необходимого для непосредственного исполнения. Правильное решение получается, если отсчет ведется от момента предыдущего пробуждения

новое время пробуждения = время предыдущего пробуждения + интервал

Таким образом, относительное время преобразуется в абсолютное. На практике необходимы две команды

wait until (ref_time);

ref_time := ref_time + 10 seconds;

Этот принцип проиллюстрирован на рис. 1б, где номинальное время отложено по горизонтальной оси. Когда абсолютное время принимается в качестве опорного, накопления ошибок времени удается избежать.

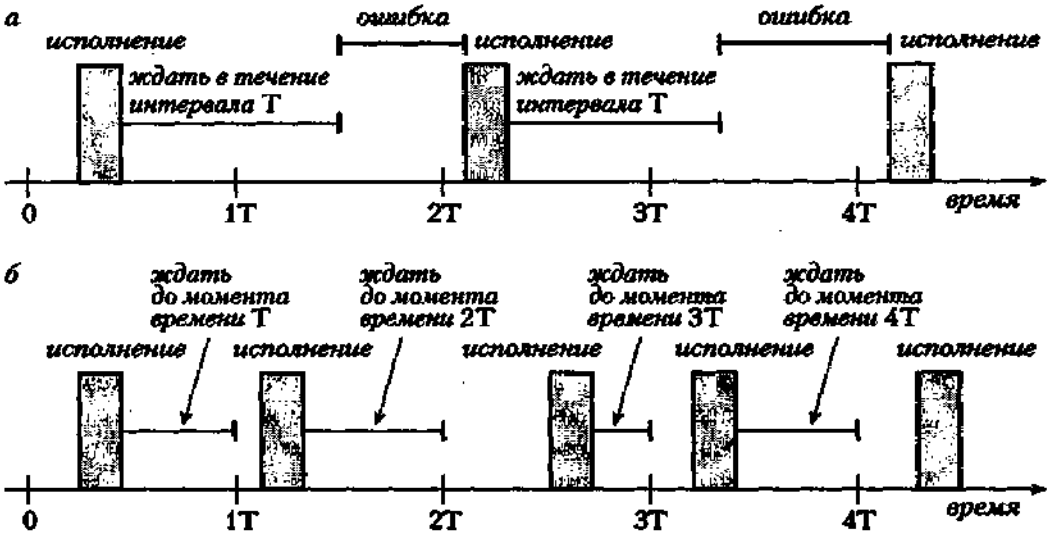


Рисунок 1. - (а) Неправильный способ определения момента очередного запуска периодических задач - ошибка времени накапливается; (б) правильное решение - ошибка времени не накапливается

3. Внутренние подпрограммы операционной системы

Типичной ситуацией при программировании в реальном времени является непосредственное обращение к подпрограммам операционной системы из-за того, что в используемом языке программирования отсутствует эквивалентное средство. Обращения к функциям операционной системы также необходимы при работе в сетевой и распределенной среде. Операционная система отвечает за все обслуживание прикладных задач, включая файловые и сетевые операции. Простое обращение к операционной системе может привести к сложной последовательности действий для доступа к удаленной базе данных, включая все сопутствующие проверки и операции управления, избавляющие прикладную программу от лишних деталей. Интерфейс операционной системы делает выполнение таких операций более прозрачным и упрощает написание сложных программ.

Многие языки программирования высокого уровня, например С, обеспечивают интерфейс с операционной системой для непосредственного вызова ее модулей из исполняемых процессов. Существуют различные виды программных интерфейсов с операционной системой непосредственные вызовы, примитивы и доступ через библиотечные модули.

Непосредственные (системные) вызовы осуществляются с помощью конструкции языка высокого уровня, которая передает управление подпрограмме, являющейся частью операционной системы. Необходимые параметры передаются списком, как при обычном обращении к подпрограмме. После завершения системной процедуры результат возвращается вызывающей программе.

Так как в многозадачной среде системные программы и примитивы могут вызываться одновременно разными процессами, их код всегда реентера-белен. Это позволяет избежать конфликтов при прерывании системной программы другим запросом, требующим ту же услугу из другого контекста.

В некоторых случаях для доступа к внутренним ресурсам операционной системы можно использовать библиотечные модули. Эти модули уже предварительно откомпилированы, и их остается только связать с основной программой. Необходимо проверить по документации системы требуемые параметры, а также механизмы их передачи и редактирования связей в языке высокого уровня.

4. Приоритеты процессов и производительность системы

Многозадачная операционная система реального времени должна допускать назначение приоритетов исполняемым процессам. Обычно приоритеты являются динамическими, что означает, что во время исполнения они могут изменяться как самими процессами, так и операционной системой. Обычно существуют определенные ограничения и механизмы контроля, которые определяют, кто и как может менять приоритеты. Назначение приоритетов оказывает серьезное влияние на работу системы в целом.

Наиболее важные процессы или процессы, время реакции которых жестко ограничено, получают более высокий приоритет. К последним относятся обработчики прерываний. Задачи, выполняющие менее важные действия,

например печать, получают более низкий приоритет. Очевидно, что необходимо обращать внимание на соглашения, используемые в системе относительно того, связан ли более высокий приоритет с большим или меньшим числом. Приоритеты имеют относительное значение и оказывают влияние только тогда, когда существуют процессы с разными приоритетами.

В системах реального времени реакция на прерывания отделена от вычислений, требующих значительных ресурсов процессора. Как только происходит событие или прерывание, его обработчик немедленно включается в очередь готовых процессов. Программы обработчиков прерываний обычно компактны, так как они должны обеспечивать быструю реакцию, например ввод новых данных, и передавать управление более сложным процессам, интенсивно потребляющим ресурсы процессора, которые исполняются с более низким приоритетом.

В примере системы управления манипулятором робота одна задача, которую можно построить как обработчик прерываний, ждет поступления от датчика новых данных о текущем положении манипулятора. Когда поступает прерывание от датчика - есть новые данные, - эта задача должна сразу получить управление. Затем она передает данные о положении программе их обработки, требующей больших вычислительных ресурсов. Эта программа не отвечает за обработку прерываний и может использовать больше времени для вычислений.

Производительность системы реального времени значительно труднее поддается оценке, чем систем, использующих обычные последовательные программы. Если обычная последовательная программа исполняется на конкретном процессоре с известной скоростью, то программа реального времени зависит от поведения окружающей среды, то есть управляемых технических процессов. Общая производительность системы должна быть достаточной для того, чтобы выполнять все операции и выдавать результаты за установленное время. Иными словами, система реального времени всегда должна быть готова к максимальной нагрузке, которую может создать технический процесс.

В развитых и сложных операционных системах, таких как UNIX, и в еще большей степени в распределенных операционных системах, доступ к большинству функций (ввод/вывод, сетевая поддержка и т.д.) происходит через системные вызовы или механизм удаленного вызова процедур. В прикладных программах для вызова системных функций используется довольно простая нотация, за которой, как правило, стоит длинная последовательность действий операционной системы. Если между двумя процессами, исполняющимися в разных узлах сети, организован программный канал, то считывание одного символа из этого канала требует целой серии операций в обоих узлах.

Поскольку на эти операции обычно наложены жесткие ограничения по времени, необходимо провести глубокий предварительный анализ прежде, чем принимать то или иное проектное решение. Если локальная сеть используется не только задачами реального времени, но и интерактивными

пользователями, то от количества и активности последних, в значительной мере зависит и ее общая нагрузка.

Многозадачные операционные системы имеют команды, показывающие в каждый момент все активные процессы, их текущий статус (например, ожидание ввода/вывода, ожидание прерывания) и долю в потреблении ресурсов процессора с момента последней перезагрузки системы или какого-либо иного события. Первый шаг по проверке характеристик системы - анализ ее работы с помощью подобной команды. Выявление процессов, занимающих слишком большую долю процессорного времени, может быть хорошей отправной точкой для поиска узких мест и оптимизации характеристик системы. Нет ничего плохого в том, если некоторые процессы загружают процессор больше, чем другие, однако разработчик системы должен иметь ясное представление о том, когда это происходит и почему.

5. Тестирование и отладка

Доказательство правильности работы программы является обязательным шагом в ее разработке. Необходимо проверить, что программа выполняет свои функции без ошибок. Визуальные и формальные методы позволяют выявить только ограниченное количество ошибок. На практике это означает, что формальная теория тестирования имеет мало смысла, а основную роль играет собственный опыт и "народные программистские" предания. Реальное тестирование проводится в "боевых" условиях.

Выявлять ошибки трудно - многие из них проявляются спорадически и их нельзя воспроизвести по желанию. Никакое доказательство не может гарантировать, что программа полностью свободна от ошибок, и никакие тесты не могут убедить, что выявлены все ошибки. Цель тестирования - найти как можно большее число ошибок и гарантировать, что программа работает с разумной надежностью. Один из создателей теории операционных систем, Эдсгер Дейкстра (Edsger Dijkstra), заметил: "Тестирование может доказать только наличие ошибок, но не их отсутствие".

Тщательный тест требует соответствующей разработки и подготовки; необходимо сочетание практических и аналитических тестов. Сначала тестовые процедуры и данные, ожидаемые результаты описываются в специальном документе. В процессе тестирования ведется журнал испытаний, который затем сравнивается со спецификацией тестов. Желательно, чтобы коллектив разработчиков системы отличался от того, который будет определять процедуры испытаний и проводить их.

При тестировании систем реального времени существует дополнительная сложность из-за большого количества возможных взаимосвязей между задачами. Вероятность внесения новой ошибки при исправлении старой очень велика - имеющийся опыт разработки программ размером свыше 10000 строк дает вероятность в пределах от 15 до 50%.

Существует два основных метода тестирования – исчерпывающий, и на примерах. При исчерпывающем тестировании проверяются все возможные комбинации входных и выходных данных. Очевидно, что этот

метод можно использовать лишь в случае, если число таких сочетаний невелико.

Метод испытаний на примерах используется наиболее часто. Производится выбор репрезентативного числа сочетаний входа и выхода. Тестовые данные должны также включать крайние значения, например находящиеся за пределами допустимого диапазона. Тестируемый модуль должен правильно распознать и обработать эти данные.

В многозадачных системах программные модули вначале тестируются отдельно. Во время такого тестирования должно быть проверено, что каждая строка программы выполняется хотя бы один раз. Иными словами, если программа содержит команды ветвления типа "if..then..else", то тестовые данные должны обеспечить выполнение обеих ветвей.

На этой фазе тестирования обычно полезны отладчики. Они позволяют непосредственно просматривать и изменять регистры процессора и области памяти при исполнении машинного кода. Отладчик вставляет в машинный код программы точки останова, в которых можно проверить состояние регистров и переменных и сравнить их со значениями, требуемыми логикой процесса. Однако с ростом сложности операционных систем и расширением функциональности системных вызовов, код которых обычно неизвестен программисту, использование отладчика может оказаться мало полезным. Обычные пошаговые отладчики не позволяют полностью оценить взаимодействие между несколькими параллельными процессами. Однако отладчики являются полезными и необходимыми средствами при разработке программ на ассемблере.

Только после того как все модули были проверены по отдельности и все обнаруженные ошибки исправлены, можно приступать к параллельному исполнению для отладки взаимодействия. Многочисленные взаимосвязи программных модулей могут привести к ошибкам в системе, даже если отдельные модули работают правильно. Общая работа системы - время обработки прерываний, производительность при разной нагрузке - проверяется на основе тестовой спецификации. Особое внимание следует обратить на функции, обеспечивающие надежность и безопасность системы.

Если система включает в себя обработку прерываний и исключений, то необходимо проверить корректность соответствующей реакции. Имитация ошибочных ситуаций позволяет оценить их последствия для системы и ее поведение в этом случае.

Результаты тестов отдельных модулей и комплексной отладки заносятся в протокол испытаний, и на его основе вносятся необходимые исправления. Ошибки тем труднее исправляются, чем позже они были обнаружены.

Расходы на тестирование - это инвестиции не только в качество системы, но и в ее общую экономическую эффективность, поскольку значительная часть расходов в течение жизненного цикла системы уходит на ее сопровождение, то есть, в конечном счете, на выявление и устранение ошибок.

ТЕМА 5. ПРОЕКТИРОВАНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Лекция 5.1. Методика комплексного проектирования и отладки систем реального времени

1. Этапы проектирования и отладки систем реального времени.

1. Основные этапы проектирования и отладки микропроцессорных и микроконтроллерных систем

Методики проектирования и отладки систем реального времени используют методологию создания микропроцессорных и микроконтроллерных систем и имеют определенную специфику. Микроконтроллерные системы ориентированы на выполнение задач управления определенными устройствами или их комплексами. Микропроцессорные системы можно условно разделить на два основных класса: универсальные, которые используются для решения широкого круга задач обработки информации, и управляющие, которые специализируются на решении задач управления процессами и объектами. Типичными примерами универсальных микропроцессорных систем являются персональные компьютеры и рабочие станции, которые применяются в самых различных сферах деятельности.

Управляющие микропроцессорные системы имеют много общего с микроконтроллерными. Они используются обычно для реализации сложных алгоритмов управления, требующих большой вычислительной мощности процессора, которую не могут обеспечить микроконтроллеры. При этом периферийные устройства, многие из которых располагаются на кристалле микроконтроллера, в микропроцессорных системах реализуются с помощью дополнительных микросхем, что повышает их стоимость и снижает надежность. Разработка интегрированных микропроцессоров, имеющих в своем составе ряд периферийных устройств, и сложно-функциональных микроконтроллеров, содержащих высокопроизводительное 32-разрядное процессорное ядро, приводит к размыванию границы применения управляющих микропроцессорных и микроконтроллерных систем, постепенному стиранию функциональных и структурных различий между ними.

Основной особенностью микроконтроллеров является наличие в их составе ПЗУ (ППЗУ, РППЗУ, ЭСППЗУ, флэш-памяти), в которое записывается резидентная рабочая программа системы. Разработка, отладка и запись в ПЗУ этой программы являются важнейшими стадиями проектирования микроконтроллерных систем. Записанная в ПЗУ рабочая программа становится составной частью системы, последующее изменение или коррекция которой обычно нежелательна или невозможна. При использовании внутреннего ПЗУ возможности внешнего контроля работы микроконтроллера в процессе отладки очень ограничены. Поэтому комплексная отладка программного и аппаратного обеспечения микроконтроллерных систем является достаточно сложной процедурой, требующей использования специализированных методов и средств контроля. Данный этап проектирования является наиболее ответственным, так как не выявленная ошибка может привести к весьма дорогостоящим последствиям.

Особенностью микроконтроллерных систем для ряда областей применения является необходимость строгого соблюдения определенных норм времени на выполнение программы или ее отдельных модулей.

В микропроцессорных схемах выполняемые модули рабочей программы загружаются в ОЗУ. Благодаря этому имеется возможность оперативной коррекции рабочей программы в случае необходимости. В процессе отладки проектировщик имеет доступ к общей шине, что облегчает текущий контроль за работой системы. Однако, наличие в большинстве современных микропроцессоров внутренней кэш-памяти ограничивает возможности внешнего контроля за ходом выполнения программы. Особенно возрастают сложности отладки при использовании микропроцессоров с суперскалярной структурой, в которых несколько команд выполняются одновременно и естественная очередность их выполнения может не соблюдаться.

Рассмотрим основные этапы проектирования и отладки систем реального времени и особенности их реализации.

Общая процедура проектирования и отладки систем реального времени включает этапы, показанные на рис.1.

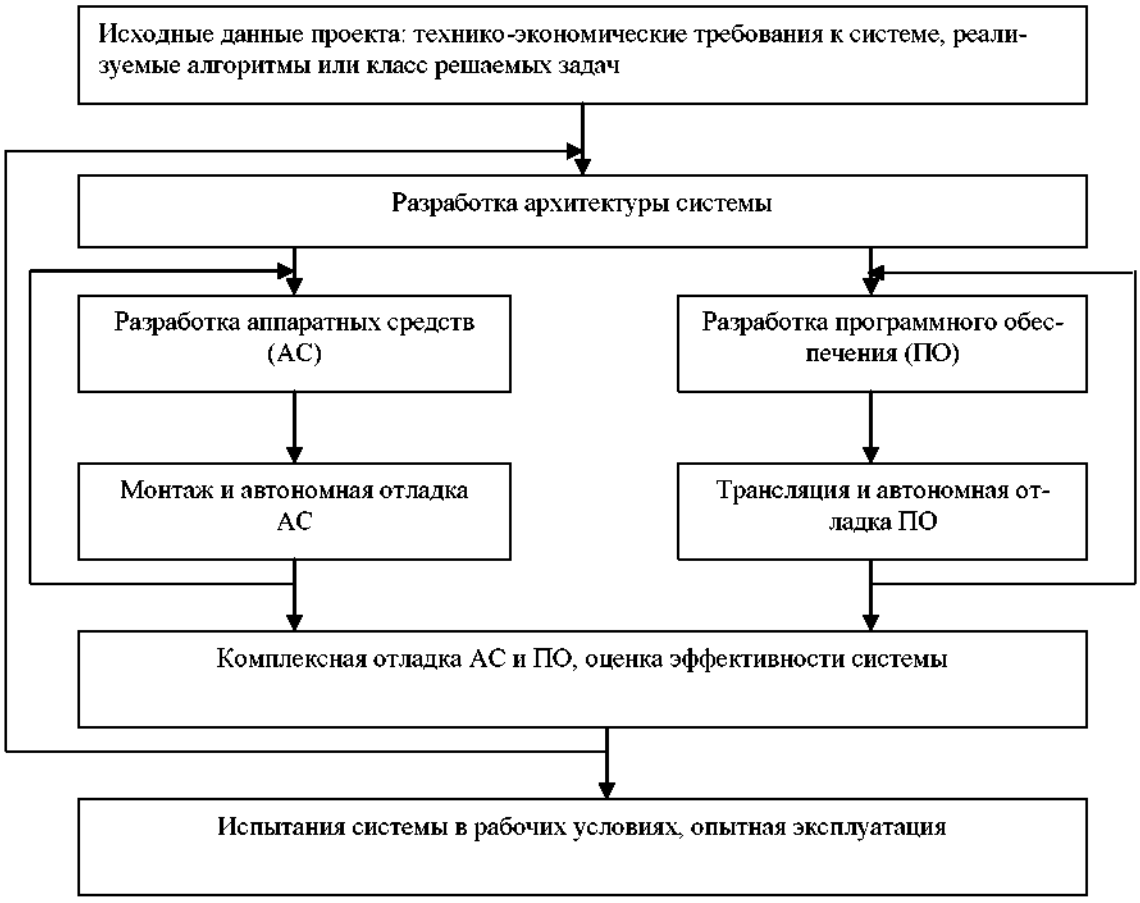


Рисунок 1. - Основные этапы проектирования и отладки систем реального времени

Исходные данные для проектирования содержат требования к основным технико-экономическим показателям: производительности, энергопо-

треблению, стоимости, надежности, конструктивным и другим параметрам. Кроме того, для управляющих систем должны быть определены реализуемые алгоритмы управления, для универсальных систем - классы выполняемых задач.

Разработка архитектуры системы подразумевает определение оптимального состава аппаратных и программных средств для решения поставленных задач. При этом разработчик решает, какие функции системы будут реализованы аппаратными средствами (АС), а какие - программным обеспечением (ПО). Определяется номенклатура АС - выбираются тип микропроцессора или микроконтроллера, объем и тип памяти, номенклатура периферийных устройств, протоколы обмена информацией и состав требуемых сигналов управления системой. Определяется также состав ПО - наличие операционной системы, ее тип и характеристики, номенклатура необходимых программных модулей, характер их взаимодействия, используемый язык программирования. Результатом выполнения этого этапа являются частные технические задания на проектирование АС и ПО.

Этап разработки АС может быть выполнен традиционными методами, с помощью которых проектируется и моделируется электрическая схема, разрабатывается печатная плата или комплект плат, после чего выполняется монтаж и отладка системы. Однако во многих случаях можно обеспечить сокращение сроков и повышение качества разработки АС путем использования "полуфабрикатов" или готовых изделий, выпускаемых рядом производителей.

Существует достаточно большая номенклатура таких изделий, которые носят, названия оценочных или целевых плат (evaluation board, target board), оценочных наборов или систем (evaluation kit, evaluation system), одноплатных компьютеров или контроллеров (SBC-single-board computer, single-board controller). Эти изделия в отечественной литературе называют платами развития. В их состав входит базовый микропроцессор или микроконтроллер, память (ОЗУ, флэш-память, служебное ПЗУ), ряд периферийных и вспомогательных схем. Обычно такие платы имеют соединитель для подключения к персональному компьютеру, с помощью которого производится комплексная отладка системы.

Если состав средств, имеющихся на плате развития, достаточен для реализации проектируемой системы, то ее разработка сводится к созданию ПО и выполнению комплексной отладки системы. Если имеющихся средств недостаточно, то они проектируются и размещаются на дополнительной плате, подключаемой к соединителю на плате развития непосредственно или с помощью кабеля. Так реализуется прототип проектируемой системы, на котором можно выполнить комплексную отладку программных и аппаратных средств, а в ряде случаев и провести проверку их функционирования в рабочих условиях. После этого разрабатывается рабочий вариант системы, объединяющий на одной плате используемые модули прототипной системы. Прототипная система может использоваться в качестве рабочей (целевой), если ее параметры и конструктивное оформление удовлетворяют требованиям

технического задания. В этом случае достигается сокращение сроков и стоимости проектирования системы.

Особенно следует отметить перспективность использования при разработке АС мезонинной технологии, которая унифицирует размеры и интерфейс базовой платы- носителя и размещаемых над ней небольших плат-мезонинов (типичный размер 45 х 99 мм). Одна плата- носитель несет от 2 до 12 мезонинов. Каждый мезонин соединяется с носителем двумя соединителями, которые выполняют также функции механических держателей. Один соединитель подключается к локальной шине платы-носителя, функциональное назначение контактов второго соединителя определяется типом мезонина, который может содержать многоканальную систему ввода-вывода, сетевые адаптеры и другие устройства. Используя серийно выпускаемые рядом производителей платы-носители и набор мезонинов, разработчик может быстро реализовать сложно-функциональные целевые системы для разнообразных применений.

Лидерами в этой области являются фирмы Green Spring Computers (США) и PEP Modular Computer (Германия), которые выпускают большую номенклатуру плат-носителей и мезонинов. Интеллектуальные платы-носители представляют собой одноплатные компьютеры или контроллеры, реализованные на базе высокопроизводительных микропроцессоров (MC68030, MC68040 и др.) или микроконтроллеров (MC68332, MC68360 и др.), которые имеют связь с персональным компьютером. Такие носители могут выполнять функции плат развития и использоваться в составе прототипных и целевых систем. Серийно выпускаемые мезонины (их около 300 типов) выполняют функции дополнительной памяти и периферийных различных устройств: параллельных и последовательных портов, таймеров-счетчиков, АЦП и ЦАП, сетевых и шинных контроллеров и др. При необходимости разработчик может самостоятельно спроектировать мезонин, выполняющий функции, которые необходимы для прототипной или целевой системы.

Таким образом, мезонинная технология является наиболее эффективным средством разработки современных электронных систем различного назначения, позволяя конфигурировать их из стандартных плат при минимальных затратах времени и средств на разработку дополнительных АС.

На этапе автономной отладки АС основными орудиями разработчика являются традиционные измерительные приборы - осциллографы, мультиметры, пробники и другие, а также логические анализаторы, которые обладают широкими возможностями контроля состояния различных узлов системы в заданные моменты времени. Весьма эффективным является использование на этом этапе средств тестирования по стандарту JTAG, которые имеются в составе многих современных моделей микропроцессоров и микроконтроллеров. С помощью размещенного на кристалле тест-Порта TAP и специальных выводов TDI, TDO, TCK, TMS, TRST# обеспечивается возможность подачи необходимых входных воздействий и считывания выходной реакции, запуск-останов процессора, изменение режима его работы. Вводом специальной команды можно установить выводы микропроцессора или микрокон-

троллера в отключенное состояние, чтобы отдельно протестировать другие устройства системы.

При разработке для универсальных микропроцессорных систем используется достаточно широкий набор языков высокого уровня, для которых имеются соответствующие компиляторы. Чаще всего используются языки C, C++, FORTRAN, Pascal, Forth. Для решения ряда задач применяются языки поддержки искусственного интеллекта Ada, Modula-2 и некоторые другие. При программировании управляющих систем чаще всего используются машинно-ориентированный язык Ассемблера или языки C, C++. Язык Ассемблера применяется в случаях, когда имеются жесткие ограничения на объем требуемой памяти или на время выполнения программных модулей. Такие случаи являются достаточно типичными при решении задач управления, поэтому Ассемблеры являются одним из основных средств создания ПО для микроконтроллерных систем. В тех случаях, когда указанные ограничения не очень жесткие, для создания ПО используются языки высокого уровня, обычно C, C++.

Автономная отладка ПО выполняется с помощью симулятора - программной модели используемого микропроцессора или микроконтроллера. На этом этапе разработчики используют широкий набор средств программирования - компиляторы, ассемблеры, дисассемблеры, отладчики, редакторы связей и другие, без которых практически невозможно создание работоспособного ПО в течение ограниченных сроков выполнения проекта.

Комплексная отладка АС и ПО является наиболее сложным и ответственным этапом создания системы. На этом этапе разработчик использует весь набор программных и аппаратных средств, применяющихся для автономной отладки АС и ПО, а также ряд специальных средств комплексной отладки. К числу таких средств относятся схемные эмуляторы - специализированные устройства, включаемые вместо микропроцессора или микроконтроллера прототипной системы и обеспечивающие возможность контроля ее работы с помощью персонального компьютера, связанного со схемным эмулятором. Схемные эмуляторы являются наиболее эффективным средством комплексной отладки систем.

Следует отметить, что многие модели микропроцессоров и микроконтроллеров имеют специальный режим отладки BDM, при котором реализуется ввод команд, ввод-вывод данных, управление режимом работы процессора с помощью последовательного специального порта. При его использовании микропроцессор или микроконтроллер может работать в режиме эмуляции под управлением подключаемого к этому порту персонального компьютера. Режим BDM позволяет существенно облегчить процедуру комплексной отладки и использовать при этом более простые и дешевые средства.

Одним из наиболее эффективных средств комплексной отладки микроконтроллерных систем являются эмуляторы ПЗУ. Эти устройства включаются вместо ПЗУ прототипной системы и работают под управлением подключенного к ним персонального компьютера. Так обеспечивается

текущий контроль за выполнением программы и ее оперативная коррекция, что значительно упрощает процесс отладки.

Для микроконтроллерных систем заключительной процедурой комплексной отладки является запись в ПЗУ объектных модулей отлаженной программы и завершающее испытание ее работоспособности. Запись программы в ПЗУ осуществляется с помощью специальных программаторов.

Для универсальных микропроцессорных систем после комплексной отладки производится оценка их производительности путем прогона специального набора тестовых программ (benchmarks).

После выполнения указанных этапов отлаженная прототипная система может быть испытана в рабочих условиях с подключением полного набора реальных периферийных устройств и объектов управления. В процессе опытной эксплуатации выявляются ошибки, не обнаруженные на этапе отладки, определяется реакция системы на возможные непредвиденные ситуации. В процессе разработки, при создании современных микропроцессорных и микроконтроллерных систем используется комплекс программно-аппаратных средств, которые помогают качественно и в ограниченные сроки выполнить их проектирование и отладку.

Лекция 5.2. Аппаратные средства поддержки проектирования и отладки систем реального времени

1. Логические анализаторы.
2. Схемные эмуляторы.
3. Эмуляторы ПЗУ.
4. Платы развития.

На различных этапах проектирования и отладки систем используются следующие специализированные аппаратные средства: логические анализаторы; различные виды плат развития; схемные эмуляторы, отладочные комплексы; эмуляторы ПЗУ; программаторы.

Рассмотрим основные особенности структуры и применения этих средств, используемых при разработке систем на базе микропроцессоров и микроконтроллеров.

1. Логические анализаторы

Логические анализаторы (ЛА) являются универсальными приборами для анализа функционирования цифровых систем. Они позволяют контролировать логическое состояние нескольких десятков точек системы в течение заданного промежутка времени и выдать информацию о состоянии в визуальном (на экране монитора) или печатном виде. Форма представления может быть символьная или графическая (временные диаграммы сигналов). Входные каналы ЛА подключаются к точкам контроля с помощью зондов-клипсов или соединителей. Число каналов в современных ЛА обычно составляет от 16 до 150. Запуск анализатора производится автоматически при поступлении на определенные каналы заданного кода (адреса, данных или комбинации управляющих сигналов) или последовательности кодов. После запуска в память ЛА записывается последовательность значений логических

сигналов временной оси (глубину контроля), которое, для большинства ЛА составляет от 2 К до 32 К. На экран выводятся несколько десятков точек для каждого канала с возможностью просмотра всей записанной в памяти последовательности состояний. Максимальная частота дискретизации временных интервалов для различных моделей ЛА имеет значение от 20 до 200 МГц.

ЛА реализуются в виде автономных измерительных приборов или плат расширения, подключаемых к базовому (host) персональному компьютеру. Эти приборы часто включают ряд дополнительных устройств, например программируемый генератор тестовых последовательностей. ЛА, реализованные в виде автономных приборов, выпускаются рядом ведущих производителей электронно-измерительной аппаратуры: Tektronix, Hewlett-Packard, John Fluke и др. Наиболее широко при отладке систем используются ЛА типов 16500B (Hewlett-Packard), 3001GPX и 3002GPX (Tektronix), PM3580 (Fluke), CLAS 4000 (Embedded Performance/Biomation). Их стоимость составляет несколько тысяч долларов.

Чтобы обеспечить разработчиков недорогими средствами контроля состояния системы, ряд производителей выпускает анализаторные платы, подключаемые к базовому персональному компьютеру, который программируется на выполнение значительной части функций ЛА. При этом для хранения последовательности состояний используется память базового компьютера, визуализация временных диаграмм в символьной или графической форме выполняется на дисплее его монитора, можно выполнить распечатку результатов измерений на принтере. Базовый компьютер управляет процессом измерения и производит обработку результатов. Благодаря этому анализаторная плата оказывается достаточно простой и на порядок более дешевой, чем автономный ЛА.

В качестве примера можно привести анализаторную плату, разработанную и поставляемую лабораторией "Моторола - Микропроцессорные системы" Московского государственного инженерно-физического института (МИФИ). Плата имеет следующие параметры:

размеры 200 х 110 мм; число входных каналов 8;

глубина буферной памяти 2 Кбайт/канал (до 2048 контрольных точек);

частота дискретизации до 40 МГц;

потребляемая мощность не более 1 Вт.

Анализаторная плата включается в соединитель расширения материнской платы базового компьютера типа IBM PC, внешние усилители-формирователи сигналов и зонды присоединяются к ней плоским кабелем. Последовательность логических состояний представляется на экране в виде диаграмм или шестнадцатеричных кодов. На экране отображается также панель управления анализатора. Предварительная настройка прибора (задание типа запуска, порогового уровня входного сигнала и др.) производится с клавиатуры базового компьютера. Плата служит дешевым и эффективным средством отладки 8-разрядных микропроцессорных и микроконтроллерных систем.

2. Схемные эмуляторы

Схемный эмулятор (СЭ) представляет собой программно-аппаратный комплекс, который в процессе отладки замещает в реализуемой системе микропроцессор или микроконтроллер. В результате такой замены функционирование отлаживаемой системы становится наблюдаемым и контролируемым. Разработчик получает возможность визуального контроля за работой системы на экране дисплея и управления ее работой путем установки определенных управляющих сигналов и модификации содержимого регистров и памяти. Благодаря наличию таких возможностей СЭ является наиболее универсальным и эффективным отладочным средством, используемым на этапе комплексной отладки системы.

Наиболее широкое применение получили СЭ, подключаемые к базовому управляющему компьютеру типа IBM PC или рабочей станции. Обычно такие СЭ конструктивно оформлены в виде прибора, размещенного в отдельном корпусе с автономным источником питания и соединенного с последовательным COM-портом базового компьютера. Некоторые типы эмуляторов для ускорения обмена связываются с компьютером через параллельный порт.

С помощью плоского кабеля к СЭ подключается эмуляторная головка, которая имеет вилку для включения в систему вместо эмулируемого микропроцессора или микроконтроллера. В головке размещается эмулирующий микропроцессор (микроконтроллер), который выполняет те же функции, что и эмулируемый, но работает под управлением компьютера. Большинство СЭ предназначено для работы с определенным семейством микропроцессоров (микроконтроллеров), причем для эмуляции каждой модели семейства используется соответствующая головка.

В структуру СЭ входят следующие блоки:

эмулятор микропроцессора или микроконтроллера (размещается в эму-ляторной головке);

память трассы, которая хранит значения сигналов, устанавливаемых на выводах микропроцессора (микроконтроллера) в процессе выполнения программы;

блок контрольных прерываний, который реализует остановы в контрольных точках, заданных пользователем с клавиатуры компьютера;

эмуляционная память (ОЗУ), которая заменяет в процессе отладки внутреннее ПЗУ микроконтроллеров или другие разделы памяти, внешний доступ к которым в процессе отладки ограничен;

таймер, используемый для контроля времени выполнения отлаживаемых фрагментов программы.

СЭ позволяет вводить в систему тестовую или рабочую программу и контролировать ее выполнение, обеспечивая прерывания в контрольных точках. Условиями прерывания могут быть различные комбинации значений адреса, данных и управляющих сигналов, поступающих на выводы эмулирующего микропроцессора или микроконтроллера. Эти комбинации задаются пользователем с клавиатуры управляющего компьютера. После останова пользователь может получить на экране полную информацию о текущем со-

стоянии любых регистров и ячеек памяти системы. С помощью памяти трассы можно просмотреть состояния системной шины для определенного количества предыдущих циклов выполнения программы. Дисассемблер дает возможность анализировать выполнение программы в соответствии с ее исходным текстом на языке Ассемблера.

Память трассы работает почти аналогично памяти ЛА, поэтому СЭ может выполнять также его функции. Число устанавливаемых контрольных точек обычно составляет несколько десятков, хотя некоторые модели современных СЭ обеспечивают существенно большие возможности (см. табл. 1). Объем памяти трассы в различных СЭ позволяет контролировать от 4 К до 512 К программных циклов. Таймер служит для определения времени выполнения фрагментов программы с учетом реальной тактовой частоты системы.

Программное обеспечение СЭ состоит из монитора - служебной программы, обеспечивающей работу всех блоков под управлением базового компьютера, компилятора или ассемблера, позволяющих программировать работу системы на языке высокого уровня или ассемблера и отладчика. Данные программные средства обычно функционируют в составе интегрированной среды проектирования-отладки. Большинство современных СЭ используют символьные отладчики и дисассемблеры, применение которых делает процесс отладки более простым и наглядным. Программное обеспечение СЭ реализует в процессе отладки выдачу данных на экран монитора в удобном для пользователя многооконном формате.

Многие типы СЭ содержат эмуляционное ОЗУ, которое заменяет ПЗУ отлаживаемой системы. Благодаря такой замене можно в процессе отладки производить оперативное изменение содержимого этой памяти. После отладки содержимое эмуляционного ОЗУ переносится в рабочее ПЗУ системы.

Некоторые модели СЭ предоставляют возможности анализа эффективности выполняемой программы, обеспечивая информацию о частоте обращения к определенным ее фрагментам, и позволяют производить отладку мультипроцессорных систем с помощью организации многоэмуляторных комплексов.

СЭ, реализующие набор перечисленных выше функций, называют отладочными комплексами или системами развития (development system). Такие комплексы выпускаются для различных семейств фирмой Motorola (MMDS05, MMDS11, CDS32, см. табл. 1) и рядом других производителей.

В табл. 1 приведены основные характеристики наиболее распространенных зарубежных моделей СЭ, которые используются для комплексной отладки систем, реализуемых на базе микропроцессоров и микроконтроллеров фирмы Motorola. Данные СЭ обладают широким набором функциональных возможностей и имеют стоимость порядка нескольких тысяч долларов. Например, эмулятор типа Flex-ICE фирмы Noral Micrologics имеет цену 10 000 долл.

Таблица 1.1. Схемные эмуляторы для отладки систем на базе микропроцессоров и микроконтроллеров фирмы Motorola

Производитель, тип	Семейство МП или МК	Объем памяти (трассы, эмуляции)	Число контрольных точек	Базовый компьютер
Motorola MMDS 05 MMDS 11 CDS 32	68HC05 68HC11 683xx	64 К (эм.) 64 К (эм.) 8 К (тр.), 1 М (эм)	64 128 4	IBM PC, PS/2 IBM PC, PS/2 IBM PC, PS/2
Hevlett-Packard HP 6470 (серия)	680x0 683xx	До 256 К (тр.) До 8М (эм.)	Не ограниченно	IBM PC, PS/2 SPARC HP9000
Applied Microsstems EL 1600 EL 3200 Code ICE Code TAP-XA	68000, 68302 68330, 340, 360 68030, 040, 060 68331, 332	-	-	IBM PC, PS/2 SPARC
Huntsvill Microsystems HMI-200-xx (серия) LITE-68HC16 LITE-68300	68HC11, 68HC16 68x0, 683xx MPC860, MPC805 68HC16 653 xx	До 8 К (эм.) До 4М (эм.) 4 К (тр.), 256 К (эм.) 4 К (тр.), 256 К (эм.)	4 4 4	IBM PC SPARC IBM PC SPARC
Microtek International MICE-III MICEpack PowerPack	680x0, 68302 68306, 307, 328 683xx, 68HC16	1М (эм.) 128 К (тр.), до 8М (эм.) 128 К (тр.), до 8М (эм.)	8+8 8+8	IBM PC IBM PC IBM PC
Lauterbach Datentechnik TRACE-32	Все типы	До 4М (тр.) До 16М (эм.)	Не ограниченно	IBM PC SPARC DEC VAX, DEC ALPHA, HP 9000
EST TRACE 16 TRACE 32 vision ISE	68HC16 683xx 683xx, MPC860 MCF5xxx	32 К (тр.) 32 К (тр.) До 128 К (тр.)	64	IBM PC SPARC HP 9000
Pentica Systems MIME 600 MIME 700 MIME 800	68HC16 68HC16, 683xx 683xx	8 К (тр.) 32 К (тр.), до 2 М (эм.) 512 К (тр.), до 5 М (эм.)	Не ограничено Не ограничено Не ограничено	IBM PC IBM PC IBM PC, SPARC **IBM PC**
Hitex teletest 32 AX68300 AX 6811	68000, 6808, 68100, 683xxx 683xxx 68HC11	32 К (тр.), до 52 М (эм.) До 8К (тр.), до 152 М (эм.) 32 К (тр.)	20 256	
Nohau EMUL68 – PC EMUL68 – PC EMUL68 - PC	68HC11 68HC16 683xx	16 К (тр.) 512 К (тр.), до 4 М (эм.) 512 К (тр.), до 4 М (эм.)		IBM PC SPARC HP 9000
Meta link ice Master-68HC05 ice	68HC05 68HC11	4 К (тр.), 56 К (эм) 4 К (тр.),	56 К 64 К	IBM PC

Master-68HC11		64 К (эм)		
Wytec WICE 68HC11XX (серия)	68HC11	64 К (эм)	64 К	IBM PC
Noral Micrologics	68020,030	32 К (тр.), 512 - К (эм.)		IBM PC

Кроме описанных сложно-функциональных и дорогих моделей СЭ рядом производителей выпускаются их упрощенные варианты, реализованные на одной печатной плате. Такие СЭ обладают ограниченными возможностями: имеют существенно меньший объем памяти трассы, не реализуют функции ЛА, не обеспечивают символьной отладки. Однако они позволяют выполнять отладку систем малой и средней сложности, имеют на порядок более низкую стоимость, поэтому находят достаточно широкое практическое применение. Некоторые типы плат развития, также выполняют часть функций СЭ.

Уникальной особенностью ряда моделей микропроцессоров и микроконтроллеров фирмы Motorola является реализация специального режима отладки BDM, который позволяет производить комплексную отладку систем без использования дорогостоящих СЭ. Для этого разработан ряд устройств, которые называются BDM- портами, BDM-отладчиками или эмуляторами. Фирма Motorola выпускает BDM-отладчики типа M68ICD16 для семейства M68HC16 и M68ICD32 для семейства M683xx. Фирма Nihau производит BDM-эмуляторы EMUL300-PC/BDM для семейства M683xx. Аналогичные BDM - эмуляторы серии 300 выпускаются фирмой EST. Для микропроцессоров MCF5202, MCF203 семейства Cold Fire разработаны BDM-эмуляторы MICE pack-J фирмы Microtel International, которые имеют память трассы емкостью 32 Кбайт и обеспечивают обслуживание 256 контрольных точек останова программы. BDM-порты BDMPort 68HC16 и BDMPort CPU32, выпускаемые фирмой EST, позволяют подключаться к соответствующим выводам микроконтроллеров семейств M68HC16 и M683xx и выполнять отладку систем, реализованных на их базе. Средства комплексной отладки, использующие режим BDM, реализуют значительную часть функций СЭ при значительно меньшей стоимости.

Следует отметить, что ряд российских организаций имеет опыт разработки и выпуска оригинальных моделей СЭ, обеспечивающих достаточный набор функциональных возможностей при низкой стоимости. Такие СЭ производятся для ряда типов микропроцессоров и микроконтроллеров, выпускаемых фирмами Intel, Zion, Microchip. Если будет реализовано производство аналогичных СЭ для систем на базе микропроцессоров и микроконтроллеров фирмы Motorola, то российские разработчики аппаратуры будут иметь широкий выбор средств отладки, имеющих различную стоимость и функциональные возможности.

3. Эмуляторы ПЗУ

Данные устройства используются при отладке систем, рабочая программа которых размещается в ПЗУ. Эмулятор ПЗУ содержит ОЗУ, которое подключается к системе вместо управляющего ПЗУ, и работает под управлением подключенного к эмулятору базового компьютера. В простейшем случае эмулятор ПЗУ позволяет в процессе отладки выполнять многократное оперативное изменение рабочей программы. Окончательный вариант рабочей программы заносится в ПЗУ системы после отладки.

Типичным примером этого класса приборов является эмулятор ПЗУ типа EMD-256, выпускаемый Консультационно-техническим центром по микроконтроллерам (КТЦ-МК, Москва). Он предназначен для эмуляции микросхем ПЗУ типа 2756 емкостью 32 Кбайт, широко используемых в 8-разрядных микроконтроллерных системах. Эмулятор имеет размеры 77 х 46 мм и включается непосредственно в панельку ПЗУ, смонтированного в системе, с помощью жесткого переходника или переходника, соединяемого с эмулятором плоским кабелем. Подключение к управляющему компьютеру производится через последовательный COM-порт. При обмене используется стандарт RS-232, скорость обмена 9600 бод. На плате эмулятора располагается светодиодный индикатор, указывающий режим его работы (прием файла от компьютера или работа в составе системы), наличие ошибок при записи файла.

Эмулятор принимает файлы в формате Motorola S-Record, что позволяет использовать его при отладке систем на базе микроконтроллеров семейств M68HC05, M68HC08, M68HC11, и в Intel HEX-формате. Распознавание формата файлов производится автоматически. Управляющая программа эмулятора, загружаемая в компьютер с дискеты, обеспечивает запуск ассемблера, установку режима работы Сом-Порта, просмотр и редактирование содержимого файла, и его загрузку. Питание эмулятора производится напряжением 5 В, поступающим от непосредственно от отлаживаемой системы, потребляемый ток 23 мА. Стоимость эмулятора 75 долл.

Более сложные "интеллектуальные" эмуляторы ПЗУ имеют более широкие функциональные возможности. Используя один из входов прерывания системы, они позволяют останавливать ее работу в заданных контрольных точках, аналогично схемному эмулятору. При этом на дисплее базового компьютера может быть представлено содержимое эмулирующей памяти. В случае использования в эмуляторе памяти трассы можно обеспечить просмотр предыдущих шагов обращения к ПЗУ, т.е. проверить последовательность выбиравшихся команд. Во многих случаях такая информация является достаточной для выполнения отладки микроконтроллерных систем. В качестве примера эмуляторов ПЗУ этого класса можно привести IDS/LC, выпускаемый компанией Cactus Logic (США), который предназначен для отладки микроконтроллерных систем на базе семейства M68HC11.

В лаборатории "Моторола - Микропроцессорные системы" МИФИ разработан эмулятор ПЗУ типа RET, обеспечивающий эмуляцию памяти емкостью до 128 Кбайт (ЭППЗУ от 2 К х 8 до 128 К х 8, ППЗУ от 256 х 4 до 2

К х 4, ОЗУ от 2 К х 8 до 128 К х 8, ПЛИС 82S100/101) с временем выборки до 50 нс. Этот эмулятор ориентирован на отладку 8-разрядных микроконтроллерных систем, в том числе систем на базе семейств M68HC05, M68HC08, M68HC11. Эмулятор RET реализован на плате с размерами 60 х 80 мм, подключаемой к управляющему IBM-PC компьютеру через параллельный LPT-порт. В качестве дополнительных функций отладки систем обеспечивается пошаговый режим микроконтроллера и работа эмулятора в качестве логического анализатора с памятью трассы 64 К.

Таким образом, эмуляторы ПЗУ могут выполнить значительную часть функций схемных эмуляторов. При этом их реализация оказывается проще и дешевле, так как они не эмулируют функции микроконтроллера, который в процессе отладки продолжает работать в составе системы. Вследствие этого эмуляторы ПЗУ являются универсальными средствами, которые могут использоваться для отладки систем с различными моделями микроконтроллеров.

4. Платы развития

Этот класс средств проектирования микропроцессорных и микроконтроллерных систем является наиболее многочисленным. Условно их можно разделить на следующие типы:

системные комплекты (evaluation kit) - набор размещенных на плате аппаратных средств, достаточных для реализации несложных систем;

отладочные платы и системы (evaluation board, system) - размещенные на плате программно-аппаратные комплексы, обеспечивающие моделирование и отладку систем различного назначения на базе определенных моделей микропроцессоров или микроконтроллеров;

целевые платы (target board) - программно-аппаратные комплексы, ориентированные на использование после отладки в качестве прототипной системы;

одноплатные компьютеры и контроллеры (single-board computer, controller) - конструктивные комплексы, предназначенные для использования в качестве базовых модулей при реализации целевых систем промышленного применения.

Эти средства могут использоваться для следующих целей:

изучение функционирования определенных моделей микропроцессоров и микроконтроллеров, получение навыков их практического применения;

тестирование и отладка программного обеспечения систем на реальных образцах микропроцессоров (микроконтроллеров);

комплексная отладка макета системы, используемого затем в качестве образца для реализации прототипной системы;

сборка и отладка прототипной или целевой системы, в состав которой входят платы развития в качестве базовых модулей.

Практически все типы плат развития содержат в своем составе порты для подключения, управляющего персонального компьютера. Чаще всего для этой цели используется последовательный обмен по стандарту RS-232. Ряд

типов отладочных и целевых плат имеют также отдельное поле для макетирования пользователем дополнительных устройств с помощью проводного монтажа.

Ввиду большого разнообразия областей и способов применения номенклатура выпускаемых плат развития очень широка и четкие границы между их типами отсутствуют. Во многих случаях отладочные платы могут использоваться в качестве целевых, а одноплатные компьютеры часто служат средствами отладки прототипных систем.

Рассмотрим отдельных типичных представителей этого класса средств проектирования - отладки.

Отладочные платы серии M68HC05EVM, производимые фирмой Motorola, служат для проектирования - отладки систем на базе семейства M68HC05. Ввиду большой номенклатуры этого семейства выпускается несколько типов таких плат для различных серий микроконтроллеров. Платы комплектуются соответствующими типами микроконтроллеров, эмуляционной памятью емкостью 8 или 16 Кбайт, содержат резидентный отладчик, позволяющий выполнять отладку программ без использования, управляющего компьютера. Порты микроконтроллера выведены на внешние соединители платы, что дает возможность подключать к ней периферийные различные устройства. На плате расположен также программатор, который позволяет переписывать отлаженную программу в ПЗУ микроконтроллера, используемого в прототипной системе. Два отдельных последовательных порта типа RS-232 обеспечивают подключение к ней управляющего компьютера и монитора.

В режиме автономной отладки плата M68HC05EVM работает совместно с внешним монитором под управлением резидентного отладчика. При этом управление осуществляется с клавиатуры монитора, информация о состоянии системы выводится на экран его дисплея, отладка программы реализуется с помощью однострочного ассемблера-дисассемблера. При работе под управлением персонального компьютера может быть использован полный комплект программных средств проектирования-отладки.

Платы серии M68HC05EVM являются эффективным средством для практического освоения методов проектирования систем на базе семейства M68HC05. Эти средства позволяют изучить функционирование данных микроконтроллеров, получить навыки их программирования, освоить способы их применения в системах управления различными устройствами и процессами.

Для отладки систем на базе семейства M68HC11 фирма Motorola выпускает несколько серий отладочных плат, отличающихся набором функциональных возможностей. Эти платы содержат резидентный монитор-отладчик BUFFALLO, который позволяет отлаживать системы с помощью внешнего монитора без участия, управляющего компьютера. На плате размещается ОЗУ пользователя емкостью 8 или 16 Кбайт. Имеется также последовательный порт для подключения управляющего компьютера с возможностью использования большого набора программных средств проектирования-отладки, разработанных для этого семейства, например, отладочного пакета

68S11SIMAB фирмы Motorola. Серия M68HC11EVM реализует функции, которые для семейства M68HC05 обеспечивались описанными выше платами M68HC05EVM. Серия M68HC11EVB выполняет аналогичные функции, но не содержит программатора. Серия M68HC11EVBU имеет на плате макетное поле для проводного монтажа дополнительных устройств, необходимых в прототипной системе. Платы SHC11EVB2 позволяют подключать к системе дополнительную плату, реализующую функции ЛА с памятью трассы на 8 К точек. Плата M68EBLP11 имеет батарейное питание, макетное поле для монтажа дополнительных устройств и двухстрочный 14-позиционный жидкокристаллический дисплей, что позволяет на его основе реализовать автономные портативные системы управления.

Российскими специалистами разработана отладочная плата TET-HC11EVB для макетирования и отладки систем на базе семейства M88HC11 (производитель - внедренческое предприятие ТЕТ, Зеленоград). Плата содержит микроконтроллер MC68HC711E9, эмуляционную память, последовательный порт стандарта RS-232 для подключения к управляющему компьютеру, макетное поле для реализации устройств пользователя. Обеспечивается работа системы в однокристальном или расширенном режиме. Плата используется вместе с комплексом средств программирования (редактор, транслятор с языка Ассемблера, отладчик), работающих в составе интегрированной среды разработки. Чтобы обеспечить отладку систем на базе различных моделей микроконтроллеров семейств M68HC05, M68HC11, фирма Motorola выпускает двухпалатные модульные отладочные системы M68HC05xxEVS и M68HC11xxEVS. Эти системы имеют базовую плату PFB, используемую для всех моделей семейства, и платы эмуляционных модулей EM, которые определяют модель используемого микроконтроллера. Путем смены платы EM, система настраивается на применение определенной модели. Данные системы содержат также программатор, эмуляционное ОЗУ емкостью 16 или 64 Кбайт, и выполняют ряд функций СЭ, обеспечивая остановы в контрольных точках программы.

Аналогичные отладочные платы производятся фирмой Motorola для микроконтроллеров семейств M68HC16 (платы M68HC16Z1EVB) и M683xx (платы M68331EVK, M68332EVK). Кроме того, можно использовать для проектирования- отладки систем на базе этих семейств двухпалатную модульную отладочную систему MEVB1632, которая настраивается на эмуляцию различных моделей путем установки на базовую плату PFB соответствующего модуля EM.

Для разработки систем на базе микропроцессоров семейства M680xO фирма Motorola выпускает интегрированную платформу разработки M68ECOXOIDP, которая содержит отладочную систему и комплект программных средств. Отладочная система состоит из базовой платы и набора подключаемых к ней процессорных модулей, содержащих различные модели микропроцессоров данного семейства. Базовая плата содержит динамическое ОЗУ емкостью 2 Мбайт, ПЗУ емкостью 1 Мбайт с резидентным программным обеспечением, панельку для подключения ПЗУ пользователя (I Мбайт),

два последовательных порта DUART (микросхема MC68681), параллельный интерфейс-таймер (микросхема MC68230), параллельный интерфейс Censorings, пять соединителей расширения для подключения устройств пользователя. Резидентное программное обеспечение содержит ассемблер- дисассемблер и отладчик, которые позволяют создавать прикладные программы средней сложности. Для реализации более сложных программных комплексов используется кросс-система программирования, установленная на базовом компьютере.

Для проектирования систем на базе микропроцессоров PowerPC фирмой Motorola разработана двухпалатная отладочная система CogeX, которая состоит из базовой платы MPC60X-CB и подключаемого к ней процессорного модуля. Набор из трех процессорных модулей MPC60X-PB позволяет реализовывать системы на базе микропроцессора MPC601, MPC603 или MPC604. Управляющий компьютер подключается к системе через последовательный порт стандарта RS-232. Система содержит динамическое ОЗУ емкостью 8 Мбайт, два соединителя для подключения микросхем ЭСППЗУ емкостью 256 Кбайт, соединители расширения для подключения внешних устройств к системной шине, модуль ввода-вывода, реализованный на базе коммуникационного контроллера MC68302. Этот модуль обеспечивает различные виды интерфейса для связи с внешними устройствами непосредственно (последовательный RS-232, SPI, параллельный Censorings), или через коммуникационные сети (ISDN и др.). Для микропроцессоров MPC602, MPC603 разработана отладочная система Big Bend, подключаемая к шине PCI персонального компьютера. Система содержит ОЗУ емкостью 3.2 Мбайт, контроллеры дисководов, жесткого диска и CD-ROM, соединители для подключения устройств, использующих стандарт шины ISA, плату для подключения к сети Ethernet и ряд других устройств.

Различные типы плат развития для 16- и 32- разрядных микропроцессоров и микроконтроллеров фирмы Motorola выпускаются также рядом других производителей отладочных средств. За последние годы в их числе появились и российские разработчики. В лаборатории "Моторола - Микропроцессорные Системы" МИФИ разработаны целевые платы для проектирования-отладки систем на базе микроконтроллеров MC68HC16Z1 и MC68332. В процессе отладки микроконтроллер на плате работает в отладочном режиме BDM, обеспечивая функции эмулятора.

Плата M68332-DK содержит, кроме микроконтроллера, три панельки для включения микросхем памяти емкостью по 32 Кбайт, порт последовательного обмена по стандарту RS-232, соединители расширения для подключения периферийных устройств. Память целевой платы может быть конфигурирована как 64 Кбайт ОЗУ и 32 Кбайт ПЗУ или наоборот. В режиме отладки плата работает под управлением персонального компьютера IBM PC, соединенного с ней через параллельный LPT-порт. Реализуется пошаговое выполнение программы или ее прогон с остановами в контрольных точках, при этом текущая информация представляется на дисплее компьютера с помощью многооконного интерфейса. Для отладки используется разработан-

ный в лаборатории отладчик DEBUG-32, который обеспечивает выдачу информации о состоянии ресурсов системы, изменение содержимого регистров и ячеек памяти, представление выполняемой программы в мнемонической форме и ее модификацию. После отладки плата может отключаться от управляющего компьютера и использоваться как автономный контроллер или в составе более сложной прототипной системы. Серийное производство этих плат производится Консультационно-техническим центром по микроконтроллерам (КТЦ-МК, Москва).

Плата M68HC16Z1-DK выполняет аналогичные функции, реализуя отладку систем на базе микроконтроллеров MC68HC16Z1 с помощью программного отладчика DEBUG-16 с последующей работой в качестве автономного контроллера.

Для отладки систем на базе коммуникационных контроллеров фирма Motorola производит платы развития M68302FADS (для MC68302), M68360QUADS (для MC68360). Платы содержат ПЗУ с резидентным отладчиком, ОЗУ емкостью до 1 Мбайт, флэш-память емкостью до 1 Мбайт, последовательный порт RS-232, имеют соединители для подключения параллельного порта управляющего компьютера (IBM-PC или SUN) и логического анализатора. Плата M68360QUADS имеет дополнительный ведомый контроллер MC68EN360, обеспечивающий подключение к сети Ethernet, а также специальные соединители, которые соединены с выводами отладочных BDM-портов, имеющихся в составе контроллеров. Платы используются в качестве отладочных или целевых плат при разработке систем на базе этих контроллеров.

Достаточно большая номенклатура одноплатных компьютеров и контроллеров - SBC (single-board computer, controller) выпускается рядом фирм. Одной из ведущих в этой области является фирма Arenas (США), которая производит ряд типов SBC на базе семейств M680x0 и M683xx. Типичным примером является SBC68000 этой фирмы, который реализован на базе микропроцессора MC68000 и содержит на плате: ОЗУ емкостью 64 или 128 Кбайт, служебное ПЗУ с записанным в нем резидентным отладчиком Tutor, последовательный двойной порт (DUART) типа MC68681, программируемый интерфейс-таймер (PIT) типа MC68230, контроллер дисковода, генератор тактовых импульсов и монитор шины. Один из последовательных портов в режиме отладки используется для подключения монитора или персонального компьютера. Стоимость платы около 400 долл.

В режиме автономной отладки SBC68000 работает под управлением резидентного отладчика Tutor, используя дисплей и клавиатуру подключенного монитора. С помощью монитора осуществляется ввод команд и данных, представляется текущее состояние ресурсов системы, обеспечивается работа однострочного ассемблера - дисассемблера. При подключении к SBC68000 управляющего персонального компьютера можно использовать более эффективные средства программирования и отладки с применением макроассемблера или языка высокого уровня. После отладки программы SBC68000 может быть отключен от монитора или компьютера и работать в качестве автоном-

ной прототипной системы или выполнять функции локального контроллера при сохранении общего управления системой со стороны персонального компьютера.

Фирмой Arnos производятся также SBC на базе микропроцессоров MC68306, MCF5204 и контроллера MC68302. Эти платы имеют расширенный объем памяти (до 1...4 Мбайт). Их стоимость составляет 400...600 долл. Плата SBC360 содержит коммуникационный контроллер MC68EN360 и средства подключения к сети Ethernet. В платах SBC360EC и SBC603 для управления MC68EN360 используются микропроцессоры MC68EC040 и MPC603. Такие платы содержат память емкостью 4 Мбайт и выше, их стоимость от 1000 до 2500 долл.

Различные серии SBC выпускаются также фирмами EST на базе MC68360, MC68341/349, MPC860, MCF5202), General Microsystems (на базе MC68030, MC68040, MC68060 совместно с MC68360, MCF5102), New Micros (на базе MC68HC11, MC68HC16, MC68332) и рядом других производителей.

При разработке современных микропроцессорных и микроконтроллерных систем перспективным является использование мезонинной технологии. Для ее реализации рядом производителей выпускаются SBC, которые имеют несколько соединителей для включения мезонинных плат, обеспечивающих получение заданной целевой системы. Эти SBC выпускаются в модульном промышленном исполнении, что позволяет легко собирать различные конфигурации систем, монтируя необходимые модули. Для унификации интерфейса системные модули ориентированы на подключение к одной из стандартных общих шин, наиболее распространенной из которых является VMEbus. Отладка системы производится под управлением персонального компьютера, подключаемого к последовательному порту SBC, или с помощью резидентного монитора-отладчика. Отлаженная система может функционировать автономно или с участием управляющего компьютера. Рассмотрим основные характеристики типичных SBC-модулей для реализации мезонинной технологии создания целевых систем, которые выпускаются рядом ведущих зарубежных фирм.

Несколько серий SBC-модулей выпускается фирмой Motorola. Наиболее распространенными из них являются SBC серии MVME162, реализованные на базе микропроцессоров MC68040 или MC68EC040. Различные типы SBC этой серии имеют объем динамического ОЗУ от 1 до 64 Мбайт, статического ОЗУ от 256 до 512 Кбайт (с батарейным питанием), ЭППЗУ и флэш-память объемом до 2 Мбайт, от двух до четырех последовательных портов RS-232, шесть 32-разрядных таймеров. Кроме того, SBC различаются наличием или отсутствием интерфейсов VMEbus, SCSI, Ethernet. На плате могут устанавливаться до четырех мезонинных модулей стандарта IP, которые содержат различные периферийные, коммуникационные и другие устройства. SBC содержат резидентный монитор-отладчик 162Bug и могут функционировать под управлением ряда ОС реального времени: OS-9, VxWorks, LynxOS и др. SBC-модули MVME1603, MVME1604, реализованные на базе PowerPC микропроцессоров MPC603, MPC604, содержат на плате динамиче-

ское ОЗУ емкостью от 8 до 128 Мбайт, два последовательных порта, интерфейс с шиной PCI, контроллер шины SCSI-2, интерфейс с сетью Ethernet, графический адаптер SVGA, контроллеры клавиатуры и манипулятора "мышь", а также позицию для мезонинного модуля стандарта PMC. Новые семейства SBC-модулей MVME2600, MVME3600, использующие RISC-микропроцессоры MPC603 и MPC604, имеют ОЗУ емкостью до 1 Гбайт и расширенные возможности подключения периферийных устройств с различными интерфейсами.

Большую номенклатуру SBC-модулей с интерфейсом VMEbus выпускает фирма Green Spring Computers (США). Модули VIPC64, VIPC65, VIPC75 реализованы на базе микропроцессоров MC68020, MC68ECO3O, содержат двухпортовое динамическое ОЗУ емкостью от 1 до 4 Мбайт, флэш-память до 1 Мбайт, два последовательных порта RS-232, один 16-разрядный таймер. На их платах имеются две позиции для IP-мезонинов. Модули семейства SBC1-5 этой фирмы не содержат позиций для мезонинов. Они используют в качестве базовых микропроцессоры MC68HCOO, MC68020, MC68ECO3O, имеют объем ОЗУ от 256 Кбайт до 4 Мбайт, объем ЭППЗУ от 128 Кбайт до 2 Мбайт. Фирма Green Spring производит также большой набор мезонинных IP-модулей. Среди них коммуникационные IP-модули IP-Comm 302-PGA и IP-Comm 360, реализованные на базе контроллеров MC68302 и MC68360, которые подключаются при необходимости организации обмена по различным каналам связи.

Фирма BVM (Великобритания) выпускает ряд типов SBC-модулей с интерфейсом VMEbus (рис. 1). Модули BVME4000 и BVME6000, реализованные на базе микропроцессоров MC68040 и MC68060, содержат на плате шесть позиций для IP-мезонинов, динамическую память емкостью до 32 Мбайт, флэш-память емкостью до 16 Мбайт, имеют интерфейсы SCSI и Ethernet. Модуль BVME3000 содержит коммуникационный контроллер MC68EN360, два дополнительных последовательных порта, до 16 Мбайт динамического ОЗУ, 1 или 2 Мбайт флэш-памяти, 512 Кбайт статическое ОЗУ с батарейным питанием, четыре позиции для IP-мезонинов. Аналогичные характеристики имеет модуль RP3000, но без интерфейса VMEbus. Модули BVME390/395 (на базе MC68EC040/MC68040) и BVME370/380 (на базе MC68ECO3O/ MC68030) имеют двухпортовое динамическое ОЗУ емкостью до 32 Мбайт, 128 Кбайт ЭСППЗУ, параллельный принтерный порт, два последовательных порта. Эти модули не содержат позиций для мезонинов. В коммуникационном SBC - модуле RP2000 (на базе MC68302) отсутствует интерфейс VMEbus. На его плате располагаются статическое ОЗУ емкостью до 512 Кбайт, ЭППЗУ емкостью 512 Кбайт, два последовательных порта с оптоизолированными выходами, две позиции для мезонинных IP-модулей. В числе других мезонинных модулей фирмой BVM выпускается коммуникационный модуль IP-302 на базе контроллера MC68302.

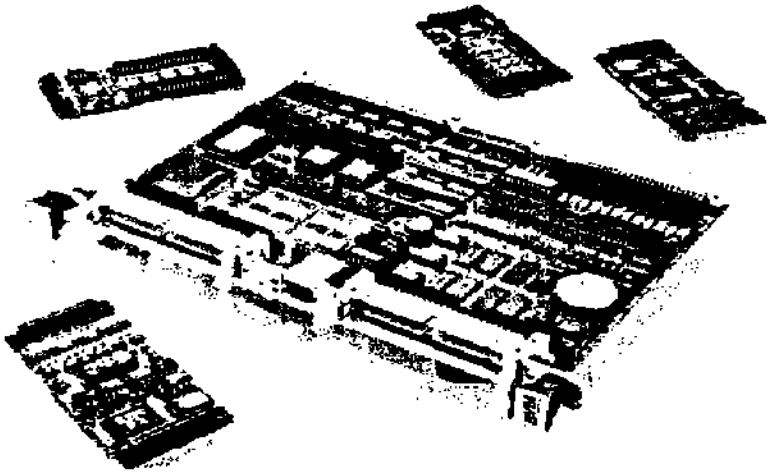


Рисунок 1. Плата-носитель SBC и мезонинные IP-модули

SBC-модули VM42 (на базе MC68040) и VM62 (на базе MC68060) фирмы PEP Modular Computers (Германия) отличаются тем, что содержат на плате также коммуникационный контроллер MC68360. На плате модулей размещаются статическое ОЗУ емкостью 256 Кбайт или 1 Мбайт, двухпортовое ОЗУ емкостью 128 Кбайт, интерфейс с VMEbus, позиции для подключения мезонинных модулей, которые обеспечивают увеличение объема ОЗУ до 16 Мбайт, введение флэш-памяти емкостью до 4 Мбайт, реализацию подключения к сети Ethernet и ряд других функций. SBC - модуль ШС-32 этой фирмы реализован на базе контроллера MC68360 или MC68EN360 и содержит также статическое ОЗУ емкостью 256 Кбайт или 1 Мбайт, шесть последовательных портов (четыре обеспечиваются с помощью мезонинов), позиции для включения мезонинов. Модуль VSBC-32 отличается от, ШС-32 наличием интерфейса VMEbus.

Имеющийся набор разнообразных плат развития помогает разработчику спроектировать и отладить макет или опытный образец системы, а в ряде случаев позволяет собрать рабочую систему из готовых модулей. При проектировании сложно-функциональных систем целесообразно использовать серийно выпускаемые SBC и периферийные различные модули, ориентированные на мезонинную технологию. Стандартизация этих изделий, их широкая номенклатура и высокие технические характеристики позволяют достаточно быстро собирать на их основе системы различного назначения. Для таких SBC и модулей имеется достаточно развитое программное обеспечение, что также упрощает и ускоряет создание системы, готовой для применения в реальных условиях. Поэтому платы развития, реализующие мезонинную технологию, наиболее перспективны для построения сложно-функциональных целевых систем.