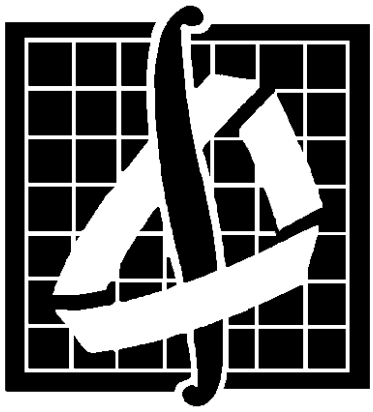
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА



Механико-математический факультет
Кафедра вычислительной математики

К.Ю. Богачев

Операционные системы реального времени

(предварительные материалы лекций)

Москва 2001 год

Настоящий документ представляет собой предварительные материалы лекций, читавшихся автором на механико-математическом факультете МГУ в 2000, 2001 гг.. Из-за недостатка времени у автора здесь практически отсутствует необходимый иллюстративный материал (диаграммы, таблицы и т.д.), приводимый на лекциях (хотя в ряде случаев присутствуют объяснения к этим иллюстрациям). Автор надеется исправить эту ситуацию в следующих версиях Документа.

Последняя версия Документа доступна по адресу `ftp://compsci.math.msu.su/pub/rtos`. по `anonymous ftp`. Поскольку сервис на указанной машине включен не круглосуточно, Документ также можно получить с `http://mars.math.msu.su`, однако, организационные проблемы на этой машине не всегда дают возможность обновить версию Документа вовремя.

Содержание

Предисловие

Настоящее пособие представляет собой введение в операционные системы реального времени (ОСРВ). Описываются их назначение, сферы применения, главные особенности и принципы внутреннего устройства. Рассматриваются основные возникающие задачи и методы их разрешения. Приводятся несколько стандартов на ОСРВ, а также описываются более десяти конкретных ОСРВ разных производителей. В пособии дан также обзор основных аппаратных компонент и их влияние на ОСРВ. Рассматриваются процессоры семейств Motorola 68xxx, Intel 80x86, PowerPC, SPARC, Intel 80960, ARM, на которых построено подавляющее большинство промышленных компьютеров, контроллеров и встраиваемых систем, а также архитектуры шин VME и PCI, играющих доминирующую роль в подобных системах.

1. Определение и основные особенности операционных систем реального времени (ОСРВ)

В этом разделе мы введем понятие "операционной системы реального времени" (ОСРВ) и рассмотрим основные отличия ОСРВ от других операционных систем.

1.1. Определение операционных систем реального времени (ОСРВ)

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

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

   Таким образом, временны́е ограничения должны быть *гарантированно* удовлетворены. Это требует от системы быть *предсказуемой*, т.е. 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.

   Хорошим примером задачи, где требуется ОСРВ, является управление роботом, берущим деталь с ленты конвейера. Деталь движется, и робот имеет лишь маленькое временно́е 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 самолет, 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 самолета и информацией о нем в управляющей программе.

   Иногда 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. Если же это ведет только к потере производительности, то говорят о "слабом" реальном времени. Большинство программного обеспечения ориентировано на "слабое" реальное время, а задача хорошей ОСРВ – обеспечить уровень безопасного функционирования системы, даже если управляющая программа никогда не закончит своей работы.

2. Стандарт POSIX 1003.1 определяет ОСРВ 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 по маркетинговым соображениям. Действительно, если интерактивную программу называют работающей в "реальном времени", то это просто означает что она успевает обрабатывать запросы от человека, для которого задержка в сотни миллисекунд даже незаметна.

4. Иногда понятие "система реального времени" отождествляют с "быстрая система". Это не всегда правильно. Время задержки реакции ОСРВ на событие не так уж важно (оно может достигать нескольких секунд). Главное, чтобы это время было *достаточно* для рассматриваемого приложения и *гарантированно*. Очень 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 ОСРВ вводят свои понятия "реального времени". Например, процесс цифровой обработки сигнала (digital signal processing, DSP) 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 секунд для анализа 2.00 секунд звука, то это не процесс реального времени. Если же требуется 1.99 секунд, то это процесс реального времени.

1.2. Типичные времена реакции на внешние события в управляемых ОСРВ процессах

Согласно определению, ОСРВ должна "обеспечить требуемый уровень сервиса в заданный промежуток времени" Этот промежуток времени обычно задается периодичностью и скоростью процессов, которыми управляет система. Приблизительное время реакции в зависимости от области применения ОСРВ может быть следующим:

- 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 очень разнятся и накладывают различные **требования на вычислительную установку**, на которой работает ОСРВ.

1. В зависимости от сложности программы управления "реальное время" накладывает различные условия на вычислительную мощность процессора для ОСРВ.

2. Внешние события становятся известны системе посредством прерываний (interrupt requests (IRQ)) (т.е. запросов на обслуживание со стороны внешних устройств). Поэтому зачастую 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 прерывние (т.е. как можно меньшее время между поступлением запроса на обслуживание и началом выполнения обслуживающей программы).

3. ОСРВ часто сама является инициатором периодических процессов, которыми управляет (например, движением пресса или луча радара). Поэтому является необходимым наличие одного или нескольких таймеров (аппаратных устройств, выдающих прерывание через заданные промежутки времени), которые могут работать в периодическом или ждущем режиме.

4. ОСРВ часто управляет очень ответственными промышленными процессами, что выдвигает очень жесткие требования на надежность используемого оборудования.

1.3. Основные области применения ОСРВ

В течение длительного времени основными потребителями ОСРВ были военная и космическая области. Сейчас ситуация кардинально изменилась и ОСРВ можно встретить даже в товарах широго потребления.

Основные области применения ОСРВ:

- **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) (computer-aided manufacturing (CAM)), автоматические системы управления технологическим процессом (АСУТП);
  - *автомобилестроение*: симуляторы, системы управления мотором, автоматическое сцепление, системы антиблокировки колес...
  - *энергетика*: сбор информации, управление данными и оборудованием...
  - *телекоммуникации*: коммуникационное оборудование, сетевые коммутаторы, телефонные станции...
  - банковское оборудование (например, во многих банкоматах работает ОСРВ QNX).
- **Товары широкого потребления**:
  - мобильные телефоны, например, в телефонах стандарта GSM работает ОСРВ pSOS;
  - цифровые телевизионные декодеры;
  - цифровое телевидение (мультимедиа, видеосерверы...);
  - компьютерное и офисное оборудование (принтеры, копиры), например, в факсах применяется ОСРВ VxWorks, в устройствах чтения компакт-дисков – ОСРВ VRTX32.

Отметим, что часто ОСРВ существуют в нескольких вариантах: полном и сокращенном, когда объем системы составляет несколько килобайтов.

1.4. Особенности оборудования, на котором работают ОСРВ

Вычислительные установки, на которых применяются ОСРВ, 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 техничекой 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 жидкокристалические 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 или иным 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. Сотоят 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 ОСРВ; типичная емкость – 500Kb;

   – 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 – 16Mb;

   – 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 – 2Mb; типичное время сохранения данных – 5 лет;

   – флеш-память (flash RAM) (электрически программируемое ПЗУ), которое играет роль диска для ОСРВ; типичная емкость – 4Mb;

   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.

   Отметим, что несмотря на наличие контроллера SCSI, обычно ОСРВ работает без дисковых накопителей, поскольку последние 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 тот же корпус вставляются платы аналого-цифровых и/или цифро-аналоговых преобразователей (АЦП и/или ЦАП)

(analog to digital and/or digital to analog converters, ADC and/or DAC), через которые осуществляется ввод/вывод управляющей информации, платы управления электромоторами и т.п. В тот же корпус могут вставляться другие такие же (или иные) промышленные компьютеры, образуя многопроцессорную систему.

Среди процессоров промышленных компьютеров доминируют процессоры семейств PowerPC (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, локомотив или самолет) 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 мы 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 часто хранится в сжатом виде и загружается 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 по-возможности должна поддерживать как можно более широкий ряд процессоров, что дает возможность потребителю выбрать процессор подходящей мощности.
- Система по-возможности должна 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 не достаточно 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, и полученная программа может работать на многих компьютерах.

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

2. Основные положения

В этом разделе мы введем основные понятия, используемые при рассмотрении операционных систем реального времени.

2.1. Основные определения

Определение. Программа — это описание на некотором формализованном языке алгоритма, решающего поставленную задачу. Программа является статической единицей, т.е. неизменяемой с точки зрения операционной системы, ее выполняющей.

Определение. Процессор — это устройство, выполняющее определенный набор инструкций. Для того, чтобы быть выполненной, программа должна быть прежде всего переведена с алгоритмического языка на язык этих инструкций (скомпилирована).

Определение. Процесс — это динамическая сущность программы, ее код в процессе своего выполнения. Имеет

- собственные области памяти под код и данные,

- собственный стек,

- (в системах с виртуальной памятью) собственное отображение виртуальной памяти на физическую,

- собственное **состояние**.

Процесс может находится 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 межпроцессного 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”.

Определение. 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.

Определение. 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 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 куски равного размера, называемые страницами (типичный размер – 4Kb), 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.

Определение. Межпроцессное взаимодействие – это тот или иной способ передачи информации из одного процесса в другой. Наиболее распространенными формами взаимодействия 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.
2. *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 по его поступлении. Требует поддержки со стороны операционной системы.

Определение. 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). Очень 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 еще тратим дополнительное время на обеспечение их взаимодействия. Улучшает ситуацию введение задач.

Определение. 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,
- (в системах с виртуальной памятью) разделяет с процессом отображение виртуальной памяти на физическую.
- 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 (например, область памяти), имеющее свое значение в каждом из них.

Определение. Ресурс – это объект, необходимый для работы процессу или задаче.

Определение. 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. Как отмечалось выше, часто в ОСРВ задача с меньшим приоритетом может вообще не получить управления при наличии в состоянии готовности задачи с большим приоритетом.

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**. Отметим, что 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/задачами, давая каждому из них квант времени, пропорциональный его приоритету. Этот квант времени часто не зависит от специфики решаемой задачи реального времени, поэтому такой подход обычно не используется в ОСРВ. Обычно в ОСРВ в состоянии выполнения может быть только один процесс. В хорошей ОСРВ это можно изменить программным путем.

Определение. Связывание (линковка, linkage) – это процесс превращения скомпилированного кода (объектных модулей) в загрузочный модуль (т.е. 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 обычно для избежания недетерминированной задержки на загрузку программы на исполнение все необходимые процессы реального времени запускают при старте системы (т.е. заранее, а не по требованию).

2.2. Типы задач

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.

2.3. Виды программирования

В зависимости от архитектуры целевого компьютера и назначения, подходы к написанию программ можно разделить на 3 группы.

1. **Последовательное программирование**. Программа исполняется на одном процессоре в виде одного процесса, состоящего из одной задачи. Поведение программы детерминировано. Результат работы не зависит от временны́х характеристик компьютера, таких, как производительность процессора, скорость переключения задач, время реакции на внешние события.
2. **Параллельное программирование**. Различают квази-параллельные программы, состоящие из нескольких независимых процессов, и истинно-параллельные программы, состоящие из нескольких связанных между собой параллельно работающих процессов или/и задач. Конечное состояние программы может зависеть от временны́х характеристик компьютера, когда процессы или/и задачи могут завершаться в разном порядке.
3. **Программирование для систем реального времени**. Является по-необходимости параллельным, поскольку на одном компьютере обычно работают несколько процессов и задач, управляющих оборудованием, находящимся в одной технологической цепочке. По определению “реального времени” необходимо гарантированно соблюсти временны́е ограничения и при этом обеспечить максимально высокую скорость выполнения.

2.4. Виды ресурсов

По своей природе ресурсы можно разделить на

- **аппаратные**:
  - процессор,
  - область памяти,
  - периферийные устройства,
  - прерывания,
- **программные**:
  - программа,
  - данные,
  - файлы,
  - сообщения.

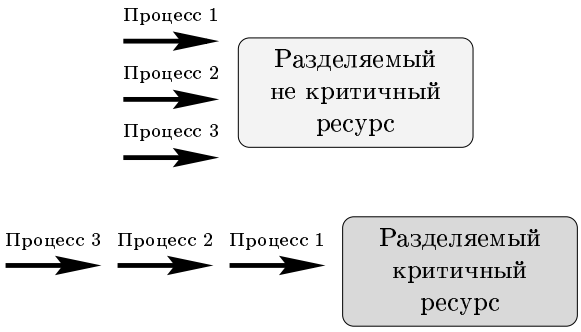


Рис. 1. Виды разделяемых ресурсов

По своим характеристикам ресурсы разделяют на:

- **активные**:
  - способны изменять информацию (процессор),
- **пассивные**:
  - способны хранить информацию,
- **локальные**:
  - принадлежат одному процессу; время жизни совпадает с временем жизни процесса,
- **разделяемые**:
  - могут быть использованы несколькими процессами; существуют, пока есть хоть один процесс, который их использует,
- **постоянные**:
  - используются посредством операций “захватить” и “освободить”,
- **временные**
  - используются посредством операций “создать” и “удалить”.

Разделяемые ресурсы бывают:

- **не критичные**:
  - могут быть использованы *одновременно* несколькими процессами (например, жесткий диск или канал Ethernet)
- **критичные**:
  - могут быть использованы *только одним* процессом, и пока этот процесс не завершит работу с ресурсом, последний не доступен другим процессам (например, разделяемая память, доступная на запись).

2.5. Типы взаимодействия процессов

По типу взаимодействия различают

- **сотрудничающие процессы**:
  - процессы, разделяющие только коммуникационный канал, по которому один передает данные, а другой их получает;
  - процессы, осуществляющие взаимную синхронизацию: когда работает один, другой ждет окончания его работы (типично для программ, управляющих рядом технологических процессов);
- **конкурирующие процессы**:
  - процессы, использующие совместно разделяемый ресурс;
  - процессы, использующие критические секции;
  - процессы, использующие взаимные исключения.

Определение. Критическая секция — это участок программы, на котором запрещается переключение задач для обеспечения исключительного использования ресурсов текущим процессом (задачей). Все ОСРВ предоставляют системные вызовы "войти в критическую секцию" и "выйти из критической секции". Отметим, что "обычные" операционные системы такой возможности пользовательским программам не дают. Время пребывания процесса (задачи) в критической секции должно быть как можно меньше, так как иначе можно нарушить временны́е ограничения на скорость реакции на внешние события. Хорошая ОСРВ должна даже при нахождении в критической секции собирать прерывания в очередь "отложенных" прерываний и обработать их при выходе из критической секции.

Определение. Взаимное исключение (mutual exclusion, mutex) — это способ синхронизации параллельно работающих процессов (задач), использующих разделяемый постоянный критичный ресурс. Если ресурс занят, то системный вызов "захватить ресурс" переводит процесс (задачу) из состояния выполнения в состояние ожидания. Когда ресурс будет освобожден посредством системного вызова "освободить ресурс", то этот процесс (задача) вернется в состояние выполнения и продолжит свою работу. Ресурс при этом перейдет в состояние "занят".

Если процессы независимы (не имеют совместно используемых ресурсов), то синхронизация их работы не требуется. Если же процессы используют разделяемый ресурс, то их деятельность необходимо синхронизировать. Например, при использовании общего блока памяти проблемы могут возникнуть даже если один процесс (задача) только читает данные, а другой – только пишет.

При синхронизации задач необходимо бороться с 3-мя проблемами:

1. "блокировка" ("lockout"):
   - процесс (задача) ожидает ресурс, который никогда не освободится,
2. "тупик" ("deadlock"):
   - два процесса (задачи) владеют каждый по ресурсу и ожидают освобождения ресурса, которым владеет другой процесс (задача),
3. "застой" ("starvation"):
   - процесс (задача) монополизировал процессор.

Для минимизации этих проблем используются следующие идеи.

- Количество ресурсов ограничено, поэтому нельзя допускать создания задач, для которых недостаточно ресурсов для выполнения.

- Задачи делятся на группы:
  - **неактивные** задачи, которым не хватило даже пассивных ресурсов, и **ожидающие** событий задачи; таким задачам активный ресурс (процессор) не дается вообще;
  - **готовые** задачи, у которых есть все необходимые пассивные ресурсы, но нет процессора; являются кандидатами на получение процессора в случае его освобождения;
  - **выполняющиеся** задачи, у которых есть все необходимые пассивные ресурсы, и процессор.

2.6. Состояния процесса

Рассмотрим более детально состояния процесса и переходы из одного состояния в другое. Состояния:

1. не существует,
2. не обслуживается,
3. готов,
4. выполняется,
5. ожидает ресурс,
6. ожидает назначенное время,
7. ожидает события.

Переходы из состояния в состояние:

1. переход 1 – 2: создание процесса
2. переход 2 – 1: уничтожение процесса
3. переход 2 – 3: активизация процесса диспетчером
4. переход 3 – 2: дезактивизация процесса
5. переход 3 – 4: загрузка на выполнение процесса диспетчером
6. переход 4 – 3: требование обслуживания от процессора другим процессом
7. переход 4 – 2: завершение процесса
8. переход 4 – 5: блокировка процесса до освобождения требуемого ресурса
9. переход 4 – 6: блокировка процесса до истечения заданного времени
10. переход 4 – 7: блокировка процесса до прихода события
11. переход 2 – 6: активизация процесса приводит к ожиданию временно́й задержки
12. переход 2 – 7: активизация процесса приводит к ожиданию события
13. переход 2 – 5: активизация процесса приводит к ожиданию освобождения ресурса
14. переход 5 – 3: активизация процесса из-за освобождения ожидавшегося ресурса
15. переход 6 – 3: активизация процесса по истечение заданного времени

16. переход 7 – 3: активизация процесса из-за прихода ожидавшегося события

Переход 4 – 3 в системах реального времени происходит сразу, как только в состояние готовности перейдет процесс с большим приоритетом. Такой механизм называют приоритетным переключением (preemption, буквально: приоритетное право на покупку). Система реального времени **должна** осуществлять переключение задач в соответствии с этим принципом.

3. Стандарты на операционные системы реального времени

В этом разделе мы рассмотрим существующие стандарты на ОСРВ. Как и во многих других областях, стандарты стали появляться лишь *после* того, как уже был создан ряд ОСРВ. Основной целью введения стандартов является облегчение переноса программного обеспечения из одной системы в другую. Отметим, что разработчики систем реального времени часто ставят задачу обеспечения максимальной скорости работы и компактности ОСРВ выше задачи ее стандартизации. Поэтому, с одной стороны, среди ОСРВ преобладают системы с уникальным устройством, а с другой стороны, многие стандарты носят весьма общий характер. При этом даже системы, декларирующие свою совместимость с некоторым стандартом, обычно содержат ряд расширений, выходящих за его рамки. Тем не менее, важность стандартов состоит в том, что они фактически выступают в качестве аксиоматической базы, задающей определения рассматриваемых объектов и понятий.

Мы кратко рассмотрим несколько стандартов. Подробно будет рассмотрен только наиболее старый и завершенный из них — европейский стандарт SCEPTRE.

3.1. Нормы ESSE консорциума VITA

Нормы ESSE консорциума VITA (VMEbus International Trade Association) находятся в стадии разработки, но уже оказывают влияние на разработчиков ОСРВ в силу весомости организации VITA (напомним, шина VME занимает лидирующее положение среди шин для промышленных компьютеров и встраиваемых систем). Нормы прежде всего ориентированы на унификацию приложений для встраиваемых систем в области телекоммуникаций, автомобилестроения и промышленности. Основной целью норм ESSE также является стандартизация ОСРВ (ядра и подсистемы ввода/вывода).

3.2. Стандарт POSIX 1003.1b

Стандарт POSIX (Portable Operating System Interface) 1003.1b, ранее существовавший под рабочим именем POSIX 1003.4 и разработанный IEEE (Institute of Electrical and Electronical Engineers), определяет расширения стандарта POSIX 1001 на операционные системы UNIX, которые позволяют использовать последние в качестве ОСРВ. Большинство приложений UNIX могут быть перенесены в такие системы, поскольку стандарт POSIX 1003.1b обеспечивает единый с системами UNIX программный интерфейс (API, application interface). Стандарт POSIX 1003 состоит следующих частей.

1. POSIX 1003.1 — определяет стандарт на основные компоненты операционной системы, API для процессов, файловой системы, устройств и т.д.
2. POSIX 1003.2 — определяет стандарт на основные утилиты.
3. POSIX 1003.1b — определяет стандарт на основные расширения “реального времени”.
4. POSIX 1003.1c — определяет стандарт на задачи (threads).
5. POSIX 1003.1d — определяет стандарт на дополнительные расширения “реального времени” (такие, как, например, поддержка обработчиков прерываний); этот стандарт еще

официально не утвержден, но некоторые ОСРВ заявляют о своей поддержке некоторых его частей.

Стандарту POSIX 1003 с расширением 1003.1b удовлетворяют такие системы, как Lynx, VxWorks, QNX. Некоторые системы, например, CHORUS, обеспечивают поддержку стандарта 1003.1b при загрузке поставляемого программного обеспечения, т.е. имеют как бы два типа API: оригинальный собственный и стандартный.

3.3. Стандарт SCEPTRE

Стандарт SCEPTRE (Standartisation du Cœur des Executifs des Produits Temps Réel Européens) (европейский стандарт на основы систем реального времени, здесь игра слов: sceptre по-французски означает "скипетр") разрабатывался в 1980–90 годы. За время его создания появились новые концепции в ОСРВ, не все из которых успели найти отражение в стандарте. В стандарте

- объединены усилия инженеров и исследователей в разработке групп спецификаций для промышленных приложений;
- даны определения и описания набора методов и подходов, используемых в ОСРВ;
- определены семь основных целей, которые должна преследовать ОСРВ; это:
  1. адекватность поставленной задаче,
  2. безопасность (система должна быть максимально устойчивой к аппаратным и программным сбоям),
  3. минимальная стоимость,
  4. максимальная производительность,
  5. переносимость (возможность реализовать систему на другом типе процессора, адекватном поставленной задаче),
  6. адаптивность (способность системы приспосабливаться к новому управляемому ею оборудованию и/или задачам),
  7. модульность (система должна состоять из достаточно независимых компонент, из которых можно построить систему, достаточную для решения поставленной задачи на имеющемся оборудовании).

Весь сервис, предоставляемый операционной системой, разделен в стандарте на следующие группы:

- коммуникации (межпроцессное взаимодействие),
- синхронизация (синхронизация процессов),
- контроль и планирование задач,
- управление памятью,
- управление прерываниями и оборудованием ввода/вывода,
- высокоуровневый интерфейс ввода/вывода и управления периферийными устройствами,
- управление файлами,
- управление транзакциями (сообщениями и передачами данных),
- обработка ошибок и исключений,

- управление временем.

Основные функции операционной системы разделены стандартом на следующие группы:

- выделение памяти и адресация объектов,
- создание и удаление объектов,
- доступ к другим машинам.

Стандарт разделяет задачи на два класса.

- **Прямые** (непосредственные) задачи – это задачи, обеспечивающие интерфейс между приложением и его внешним окружением, например, задачи ввода/вывода, задачи, обрабатывающие события таймера. Такие задачи активизируются непосредственно сигналом, приходящим с управляемого интерфейса (из внешнего мира). Эта активизация реализуется посредством механизма прерываний.
- **Косвенные** (отложенные) задачи – это задачи, активизируемые другими задачами. Такие задачи оперируют с данными, полученными от прямых задач или от других косвенных задач.

Стандарт определяет основные виды взаимоотношений между задачами:

- создание,
- удаление,
- активизация,
- остановка,
- коммуникация,
- синхронизация,
- обмен сигналами,
- взаимное исключение.

Стандарт определяет основные виды состояния задач.

- **Не существует**: нет связанного с задачей дескриптора (структуры данных, характеризующей задачу с точки зрения операционной системы).
- **Существует**: задача обладает определенным дескриптором.
- **Не исполнима**: задача существует, но не может ни активизироваться, ни продолжиться.
- **Исполнима**: задача существует и ее можно исполнить.
- **Не обслуживается**: задача исполнима, но требует активизации или успешно завершена.
- **Обслуживается**: задача исполнима, начинает исполнение и не терминирована.
- **Ожидает**: задача обслуживается и ожидает выполнения определенного условия для продолжения своего исполнения.
- **Активна**: задача обслуживается и не ожидает выполнения какого-либо условия, а только ожидает когда освободится процессор от других задач.

- **Готова**: задача активна и ожидает процессора.
- **Выполняется**: задача использует процессор.

Стандарт определяет основные виды межзадачного взаимодействия:

- **Обмен сигналами (событиями)**. Хотя событие предполагает наличие хотя бы двух задач, стандарт SCEPTRE ассоциирует событие с единственной задачей. Сделано это по следующим причинам:
  - хотя событие может ожидаться несколькими задачами, с ним необходимо связана единственная очередь ожидания;
  - ассоциация события только с одной задачей минимизирует расходы на адресацию при обмене событиями (вся необходимая информация, которой обмениваются задачи должна быть "заложена" в самом событии).

  Хотя событие связано с одной задачей, обмен событиями между двумя задачами является операцией, в которой участвуют обе задачи: одна задача ждет сигнала, а вторая его посылает. **Событие** с точки зрения SCEPTRE – это объект, принимающий два состояния: "прибыл" или "не прибыл". Над событиями определены элементарные операции:
  - послать,
  - ожидать,
  - очистить (удалить поступившее событие),
  - проверить (поступление).
- **Коммуникация**. Стратегии коммуникации между задачами, работающими на одном компьютере, реализованы в виде одной структуры *очереди*:
  - очереди ожидающих задач,
  - очереди сообщений.

  Над очередями (организованными по принципу FIFO, First In - First Out) определены следующие элементарные операции:
  - положить элемент (задачу, сообщение) в очередь,
  - взять элемент (задачу, сообщение) из очереди,
  - очистить очередь,
  - проверить наличие элемента (задачи, сообщения).
- **Исключения** – это неожиданная ситуация, в результате которой задача не может далее исполняться. Детектирование исключений может быть программным или аппаратным. Типы исключений:
  - *синхронные* (например, при подсчете времени),
  - *асинхронные* (все остальные, например, ситуация деления на 0).

  Для обслуживания исключения необходимы:
  - обслуживающая программа,
  - соответствующая задача (в состоянии ожидания исключения),
  - механизм переключения от текущей задачи к задаче обработки исключения,
  - механизм возврата из задачи обработки исключения к текущей задаче.
- **Семафоры** – это высокоуровневый механизм синхронизации задач. Различают:

– *Двоичные (булевские) семафоры* – это механизм взаимного исключения для защиты критичного разделяемого ресурса; определены следующие элементарные операции:

  * взять (если семафор уже "взят" другой задачей, то эта операция переводит задачу в состояние ожидания освобождения семафора),
  * вернуть (если семафор ожидается другой задачей, то она может быть активизирована и может вытеснить текущую задачу, например, если ее приоритет выше),
  * попробовать взять (если семафор свободен, то взять его, иначе вернуть признак занятости семафора без перевода задачи в состояние ожидания; используется, если задача может продолжить работу без затребованного ресурса, например, если можно использовать другой ресурс).

  Один семафор может ожидаться несколькими задачами, поэтому система организует *очередь задач*, ожидающих семафор. В этой очереди могут быть задачи с разным приоритетом, поэтому должна быть возможность управления порядком активизации задач из очереди (в порядке поступления (FIFO) или в порядке поступления, начиная с самых приоритетных (такая очередь называется приоритетной)).

– *Счетные семафоры* – это механизм взаимного исключения для защиты ресурса, который может быть одновременно использован не более, чем ограниченным фиксированным числом задач (например, 4-х канальный порт ввода/вывода может быть использован не более чем четырьмя задачами, требующими по одному каналу каждая). Семафор представляет собой счетчик, уменьшаемый при каждой выдаче ресурса и увеличиваемый при каждом его возвращении. Если счетчик не находится в заданном диапазоне (ресурса больше нет), то при затребовании ресурса задача перейдет в состояние ожидания. Над семафором определены следующие элементарные операции:

  * взять k единиц из семафора, т.е. уменьшить счетчик на k (если в счетчике нет k единиц, то эта операция переводит задачу в состояние ожидания наличия как минимум k единиц в семафоре),
  * вернуть k единиц в семафор, т.е. увеличить счетчик на k (если семафор ожидается другой задачей и ей требуется не более k единиц, то она может быть активизирована и может вытеснить текущую задачу, например, если ее приоритет выше),
  * попробовать взять k единиц из семафора (если семафор свободен, то взять k единиц из его, иначе вернуть признак занятости семафора без перевода задачи в состояние ожидания).

  Как и для двоичных семафоров система организует *приоритетную очередь задач*, ожидающих семафор.

- **Модель клиент – сервер** – это высокоуровневый комбинированный механизм коммуникации задач. Состоит из очереди передаваемой информации (сообщений) и события, посылаемого в случае, если очередь не пуста.

В настоящий момент стандарт развивается для поддержания:

- требований "жесткого" реального времени,
- независимости от окружения (типа процессора, вида компьютера, языка разработки приложений),
- распределенных и многопроцессорных вычислений,
- требований повышенной надежности.

4. Типы архитектур операционных систем реального времени

В этом разделе мы рассмотрим базисные принципы внутреннего устройства операционных систем реального времени. Мы рассмотрим классический и объектно-ориентированный подходы к построению ОСРВ, а также три основных типа архитектур ОСРВ:

1. монолитный,
2. модульный,
3. объектный.

4.1. Объектно-ориентированный подход в программировании

Рассмотрим кратко, на уровне идей, очень популярный в последнее время "объектно-ориентированный подход" к разработке программного обеспечения.

Всякая программа, работающая (управляющая, анализирующая и т.д.) с некой системой (программной, аппаратной и т.д.), фактически работает с неким формализованным представлением об этой системе, называемом **моделью**. **Объектно-ориентированный подход** — это техника построения и описания этой модели. В этой технике мы моделируем систему как некоторое количество взаимодействующих объектов. Таким образом, вне зависимости от типа моделируемой системы мы представляем ее содержание как набор объектов с теми или иными связями между ними. Какие объекты будут включены в модель и какие связи между ними будут учтены в модели, зависит от поставленной задачи.

Человеческое мышление воспринимает окружающий мир в терминах объектов. Поэтому при использовании объектно-ориентированного подхода человеку легче создавать модель, и сама модель получается **понятнее** и **более соотносится с реальностью**. Для внесения уточнений в модель и/или ее модификации часто достаточно внести **локальные изменения** в один объект.

4.1.1. Объекты

Объект характеризуется набором операций и состоянием, запоминающим эффект от этих операций. Снаружи объекта (т.е. используя его в модели) мы видим только эти операции, но не то, как они устроены и как работают. Внутренность объекта (его внутренняя информация) скрыта от других объектов. Другими словами, объект характеризуется тем, **что** он выполняет, но не тем, **как** он это делает. Внутри объекта мы можем видеть его информационную структуру и детали реализации его операций. В информационной структуре выделяют три компоненты:

- **атрибуты** объекта – основные характеристики, которые надо помнить об объекте;
- **части** объекта (если присутствуют) – это другие объекты, из которых составлен данный объект, который в этом случае называется **агрегатным** (составным) (например, объект "человек" может быть представлен как состоящий из объектов "рука", "нога" и т.д.),
- **поведение** объекта – это набор операций с объектом.

Отношения между объектами в модели разделяют на

– **статические**, существующие в течение длительного времени и подразумевающие, что взаимодействующие объекты знают о существовании друг друга;

– **динамические**, когда объекты устанавливают связь между собой в процессе работы. Эти отношения между объектами A и B реализуются в виде посылки объектом A **запроса** на выполнение той или иной операции (объекта B) объекту B. С программной точки зрения под запросом обычно понимают посылку сообщения.

Вся информация в объектно-ориентированной системе хранится в ее объектах, и выполнить с ней какое-либо действие можно, только послав объекту(ам) запрос на выполнение этого действия. Поведение и информация **инкапсулированы** в объекте. Объекты, таким образом, обеспечивают **сокрытие информации**.

Понятие объекта тесно связано с понятием "абстрактного типа данных". **Абстрактный тип данных** – это модель (структура данных) с определенным набором операций, воздействующих на нее. Преимущества абстрактных типов данных:

- они могут быть использованы независимо от их реализации (механизма сокрытия информации),
- они просты: их пользователь не может быть вовлечен в их логическую структуру, поскольку может оперировать лишь с их спецификацией.

Преимущества использования объектов:

1. **Двойная защита**:
   - Внутри объекта. Атрибуты объекта могут быть изменены только внутренними методами объекта. Пользователю не надо знать реализацию этих методов.
   - Вне объекта. Объект не знает своего окружения и не может его случайно модифицировать. Только сам объект отвечает за свое поведение.
2. **Модульность**: приложение состоит из объектов, обменивающихся сообщениями.
3. **Легкость отладки и сопровождения** является следствием защищенности и модульности.
4. **Повторная используемость**: хороший объект не зависит от своего окружения и может быть без проблем использован на другой архитектуре процессора или в другой задаче.

4.1.2. Классы и представители

Для описания объектов, имеющих одинаковое поведение и информационную структуру, вводят понятие класса объектов. **Класс** представляет собой шаблон для создания объектов и определяет внутреннее устройство этих объектов. Объекты одного класса имеют одинаковое определение для своих операций и информационных структур. Не следует путать класс объекта и его тип. **Тип** определяется набором операций, которые с ним можно производить. Класс же включает в себя, помимо этого, еще информационную структуру. Поэтому можно рассматривать класс как одну из возможных реализаций типа.

В объектно-ориентированных системах каждый объект принадлежит некоторому классу. Объект, принадлежащий некоторому классу, называется **представителем** этого класса. Часто термины "объект" и "представитель класса" используются как синонимы. Таким образом, **представитель** – это объект, созданный по классу (как по шаблону). Класс описывает (поведенческую и информационную) структуру представителя, а текущее состояние представителя определяется операциями, выполняемыми с ним.

4.1.3. Полиморфизм

Динамическое взаимодействие представителей, созданных из классов, определяет поведение модели. Взаимодействие представителей осуществляется путем посылки запросов между ними. Если представитель посылает запрос и ему не важно, к какому классу принадлежит представитель–получатель, то это называют **полиморфизмом**. Другими словами, **полиморфизм** означает, что отправителю запроса не требуется знать класс представителя–получателя; представитель–получатель может принадлежать любому классу.

Однако, обычно используют понятие **ограниченного полиморфизма**, когда на возможные классы представителя–получателя накладывают некоторые ограничения. В качестве таких ограничений обычно выступают требования наследования (см. ниже) от некоторого класса.

Полиморфизм позволяет динамически изменить управляемый объект без изменения поведения управляющего объекта. Например, замена объекта "электрический двигатель" на объект "двигатель внутреннего сгорания" не повлияет на объект "управляющее устройство", использующий абстрактный объект "двигатель".

4.1.4. Наследование

При описании классов можно заметить, что многие из них имеют общие характеристики (поведенческие и информационные структуры). Мы можем собрать общие характеристики в один специальный класс и сделать остальные классы его **наследниками**. В этом случае при описании этих классов мы должны будем лишь описать добавочные компоненты, специфичные для каждого из классов. Говорят, что класс B является **наследником** класса A, если операции и информационные структуры, описанные в классе A, являются частью (подмножеством) класса B. При этом класс B называют **потомком** класса A, а класс A – **предком** для класса B. Если класс B является **непосредственным потомком** класса A, то его называют **дочерним**, а класс A – **родительским** для класса B.

Наследование позволяет повторно использовать общие описания. С программной точки зрения это означает возможность повторного использования существующего кода, что часто рассматривается как основное преимущество объектно-ориентированного подхода.

При наследовании мы также имеем еще одно преимущество. Если мы хотим изменить характеристики класса предка, то это надо сделать только в одном месте – в его описании. При этом эти изменения будут автоматически внесены во все его потомки.

При наследовании некоторые поведенческие и/или информационные структуры могут **переопределять** соответствующие структуры предка. Переопределения является достаточно легким и гибким способом построения новых классов, но оно затрудняет понимание иерархической структуры классов, поскольку потомок наследует только часть характеристик предка, а другую часть переопределяет.

Наследование может использоваться для следующих целей.

1. **Повторное использование кода** с помощью наследования может проявляться двумя путями:

   а) у двух классов находится общая часть, она выделяется в абстрактный класс, который наследуется этими классами;

   б) находится подходящий класс в библиотеке классов и строится его потомок, обладающий нужными свойствами.

2. **Построение подтипа**. Если потомка класса A можно использовать во всех местах, где использован класс A, то говорят, что классы **поведенчески совместимы**, а потомок представляет подкласс класса-предка. На практике это означает, что потомок имеет тот же программный интерфейс, что и предок. Это обычно получается, если наследование образовано только расширением, а не переопределением, структур в предке, причем расширение не наложило никаких дополнительных ограничений на структуры предка.

3. **Специализация класса**. Если потомок модифицирован при наследовании так, что он более не является поведенчески совместимым с предком, то говорят, что класс был специализирован. Обычно это означает, что часть операций и/или информационных структур, была переопределена или удалена.

4. **Концептуальная**. Использование идеи наследования проявляет причинно-следственные связи и является интуитивно понятной.

4.2. Классический и объектно-ориентированный подходы к построению ОСРВ

В силу преимуществ объектно-ориентированного подхода приложения создаются на его основе, используя тот или иной язык 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 ОСРВ, используя объектно-ориентированный подход. При этом

- как приложение, так и операционная система полностью объектно-ориентированы 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++.

4.3. Монолитная архитектура

ОСРВ с монолитной архитектурой можно представить в виде

- прикладного уровня: состоит из работающих прикладных процессов;
- системного уровня: состоит из монолитного ядра операционной системы, в котором можно выделить следующие части:
  - интерфейс между приложениями и ядром (API),
  - собственно ядро системы,
  - интерфейс между ядром и оборудованием (драйверы устройств).

API в таких системах играет двойную роль:

1. управление взаимодействием прикладных процессов и системы,
2. обеспечение непрерывности выполнения кода системы (т.е. 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 API как прерывания или ловушки (специальный тип исключений). Это сильно увеличивает время их работы.

2. Ядро не может быть прервано пользовательской задачей (non-preemptable). Это может приводить к тому, что высокоприоритетная задача может не получить управления из-за работы низкоприоритетной. 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.

4.4. 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 (т.е. отсутствие переключения задач во время исполнения микроядра).

Недостатки модульной архитектуры фактически те же, что и у монолитной. 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), но остались.

4.5. 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, т.е. система полностью preemptible.

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 (т.е., 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. Все эти приемы позволяют 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 (последнее обеспечить значительно сложнее)). 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 (напомним, 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 (т.е. образовывать с ним связанный загрузочный образ еще до загрузки) и должны работать в привилегированном (суперпользовательском) режиме. Далее, 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.

4.6. Строение 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 килобайтов.

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).

Критерии выбора ОСРВ:

- производительность,

- надежность, круглосуточная готовность,

- поддержка различных типов процессоров,

- поддержка многопроцессорности,
- наличие средств разработки на требуемом языке,
- наличие механизмов реального времени,
- поддержка файловой системы.

Роль управляющей системы ОСРВ:

- управляет взаимным исключением и взаимодействием задач для оптимизации времени использования процессора; управление основывается на ведущихся ОСРВ таблицах дескрипторов задач;
- предоставляет приложению основные возможности по управлению временем, периферийными устройствами, взаимодействию с оператором;
- предоставляет набор библиотечных функций для удобного доступа к возможностям системы, например, почтовые ящики, семафоры и т.д.
- занимается планированием задач.

5. Синхронизация и взаимодействие процессов

Доступ процессов (задач) к различным ресурсам (особенно разделяемым) в многозадачных системах требует синхронизации действий этих процессов (задач). Способы осуществления взаимодействия подразделяют на:

- **безопасное взаимодействие**, когда обмен данными осуществляется посредством "объектов" взаимодействия, предоставляемых системой; при этом целостность информации и неделимость операций с нею (т.е. отсутствие нежелательного переключения задач) неявно обеспечиваются системой; примерами таких "объектов" взаимодействия являются семафоры, сигналы и почтовые ящики;
- **небезопасное взаимодействие**, когда обмен данными осуществляется посредством разделяемых ресурсов (например, общих переменных), не зависимых от системных объектов взаимодействия; при этом целостность информации и неделимость явно обеспечивается самим приложением (в подавляющем большинстве случаев – посредством того или иного системного объекта синхронизации и взаимодействия).

Поскольку для любого типа взаимодействия требуются системные объекты синхронизации, то все имеющиеся ОСРВ предоставляют приложениям некоторый набор таких объектов. Ниже мы рассмотрим самые распространенные из них.

5.1. Разделяемая память

Определение. Разделяемая память – это область памяти, к которой имеют доступ несколько процессов. Взаимодействие через разделяемую память является базовым механизмом взаимодействия процессов, к которому сводятся все остальные. Оно, с одной стороны, является самым быстрым видом взаимодействия, поскольку процессы напрямую (т.е. без участия ОСРВ) передают данные друг другу. С другой стороны, оно является небезопасным, и для обеспечения правильности передачи информации используются те или иные объекты синхронизации.

В системах с виртуальной памятью существуют два подхода к логической организации разделяемой памяти.

1. Разделяемая память находится в адресном пространстве операционной системы, а виртуальные адресные пространства процессов отображаются на нее. В зависимости от реализации операционной системы это может приводить к переключению задач при работе с разделяемой памятью (поскольку она принадлежит ОСРВ, а не процессу) и фиксации разделяемой памяти в физической памяти (поскольку сама ОСРВ не участвует в страничном обмене).

2. Разделяемая память логически представляется как файл, отображенный на память (т.е. файл, рассматриваемый как массив байтов в памяти). При этом разделяемая память полностью находится в пользовательском адресном пространстве и отсутствуют дополнительные задержки при доступе к ней.

В силу большей эффективности рекомендуется использовать второй способ работы с разделяемой памятью.

В системах с виртуальной памятью над разделяемой памятью определены следующие элементарные операции.

- создать (или отрыть) разделяемую память, при этом разделяемая память появляется в процессе как объект, но доступ к ее содержимому еще невозможен;

- подсоединить разделяемую память к адресному пространству процесса, при этом происходит отображение разделяемой памяти на виртуальное адресное пространство процесса; после этой операции разделяемая память доступна для использования;

- отсоединить разделяемую память от адресного пространства процесса, после этой операции доступ к содержимому разделяемой памяти невозможен;

- удалить (или закрыть) разделяемую память, реально разделяемая память будет удалена, когда с ней закончит работать последний из процессов.

В системах без виртуальной памяти эти операции тривиальны.

Отметим, что в разных процессах разделяемая память может быть отображена в разные адреса виртуальной памяти. Следовательно, в разделяемой памяти нельзя хранить указатели на другие элементы разделяемой памяти (например, классическую реализацию однонаправленного списка нельзя без изменений хранить в разделяемой памяти).

5.2. Семафоры

Определение. Семафор – это объект синхронизации, задающий количество пользователей (задач, процессов), имеющих одновременный доступ к некоторому ресурсу. С каждым семафором связаны счетчик (значение семафора) и очередь ожидания (процессов, задач, ожидающих принятие счетчиком определенного значения). Различают:

- *двоичные (булевские) семафоры* – это механизм взаимного исключения для защиты критичного разделяемого ресурса; начальное значение счетчика такого семафора равно 1;

- *счетные семафоры* – это механизм взаимного исключения для защиты ресурса, который может быть одновременно использован не более, чем ограниченным фиксированным числом задач n; начальное значение счетчика такого семафора равно n.

Над семафорами определены следующие элементарные операции (ниже $k = 1$ для булевских семафоров)

- взять k единиц из семафора, т.е. уменьшить счетчик на k (если в счетчике нет k единиц, то эта операция переводит задачу в состояние ожидания наличия как минимум k единиц в семафоре, и добавляет ее в конец очереди ожидания этого семафора);

- вернуть k единиц в семафор, т.е. увеличить счетчик на k (если семафор ожидается другой задачей и ей требуется не более, чем новое текущее значение счетчика единиц, то она может быть активизирована, удалена из очереди ожидания и может вытеснить текущую задачу, например, если ее приоритет выше);

- попробовать взять k единиц из семафора (если в счетчике $\geq k$ единиц, то взять k единиц из его, иначе вернуть признак занятости семафора без перевода задачи в состояние ожидания);

- проверить семафор, т.е. получить значение счетчика;

- блокировать семафор, т.е. взять из него столько единиц, сколько в нем есть (при этом иногда бывают две разновидности этой операции: взять столько, сколько есть в данный момент, или взять столько, сколько есть в начальный момент, т.е. максимально возможное количество, именно последнее обычно называют блокировкой, поскольку такая задача будет монопольно владеть ресурсом);

- разблокировать семафор, т.е. вернуть столько единиц, сколько всего было взято данной задачей по команде блокировать.

Логическая структура двоичных семафоров особенно проста. Счетчик семафора s инициализируется 1 при создании. Для доступа к нему определены две примитивные операции:

- $Get(s)$ – взять (или закрыть) семафор s, т.е. запросить ресурс; эта операция вычитает из счетчика 1;

- $Put(s)$ – вернуть (или открыть) семафор, т.е. освободить ресурс; эта операция прибавляет к счетчику 1.

Эти операции **неделимы**, т.е. переключение задач во время их исполнения запрещено. В процессе работы состояние счетчика может быть:

- 1 – ресурс свободен,

- 0 – ресурс занят, очередь ожидания пуста,

- $m < 0$ – ресурс занят, в очереди ожидания находятся $|m|$ задач.

Рассмотрим пример.

1. $A.Get$ – задача A завладела ресурсом,

2. $C.Get$ – задача C запросила ресурс, который занят, следовательно, задача C заблокирована и помещена в очередь ожидания,

3. $B.Get$ – задача B запросила ресурс, который занят, следовательно, задача B заблокирована и помещена в очередь ожидания,

4. $A.Put$ – задача A освободила ресурс, который был передан первой задаче в очереди ожидания, т.е. C (которая в свою очередь активизирована),

5. $C.Put$ – задача C освободила ресурс, который был передан первой задаче в очереди ожидания, т.е. B (которая в свою очередь активизирована),

6. $B.Put$ – задача B освободила ресурс, который будет передан первой задаче, которая вызовет Get.

Иногда рассматривают **личные** или **приватные** семафоры. Такой семафор создается задачей T сразу в закрытом состоянии. При этом задача T имеет право вызывать только функцию Get, а все остальные задачи – только функцию Put. С помощью таких семафоров легко организовать обмен между задачами по типу клиент – сервер. Задача T_1 (сервер) сразу после старта создает личный семафор S_1 и вызывает функцию $S_1.Get()$. Тем самым она блокируется до освобождения семафора S_1. Задача T_2 (клиент) сразу после старта создает личный семафор S_2, подготавливает данные в разделяемой с задачей T_1 памяти и вызывает функцию $S_1.Put()$. Это приводит к активизации задачи T_1. Задача T_2 продолжает работу и вызывает функцию $S_2.Get()$, блокируясь до освобождения семафора S_2. Задача T_1 (сервер) по окончании обработки данных вызывает функции $S_2.Put()$, $S_1.Get()$, активизируя задачу T_2 (клиента) и блокируя себя. Ниже мы рассмотрим другие способы реализации взаимодействия клиент – сервер.

5.3. События

Определение. Событие – это логический сигнал (оповещение), приходящий асинхронно по отношению к течению процесса. С каждым событием связаны булевская переменная E, принимающая два значения 0 – событие не пришло, и 1 – событие пришло, и очередь ожидания (процессов, задач, ожидающих прихода события). Над событиями определены следующие элементарные операции:

- $Send(E)$ – послать событие, т.е. установить переменную E в 1 (по традиции UNIX систем эту функцию обычно называют $Kill$, поскольку среди событий определено такое, единственной реакцией на которое является немедленное аварийное завершение получившей его задачи), при этом все задачи из очереди ожидания активизируются;

- $Wait(E)$ – ожидать события, (если события нет, т.е. переменная E равна 0, то эта операция переводит задачу в состояние ожидания прихода события, и добавляет ее в конец очереди ожидания этого события, как только событие придет, задача будет активизирована);

- $Reset(E)$ – очистить (удалить поступившее событие), т.е. установить переменную E в 0;

- $Test(E)$ – проверить (поступление) – получить значение переменной E.

Рассмотрим пример.

1. $B.Wait$ – задача B вызвала $Wait$; поскольку $E = 1$, то это не имеет никакого эффекта, B продолжает исполнение;

2. $Reset$ – была вызвана функция установки $E = 0$ (одной задач A, B или C, или другой задачей);

3. $B.Wait$ – поскольку $E = 0$, то задача B заблокирована и помещена в очередь ожидания;

4. $C.Wait$ – поскольку $E = 0$, то задача C заблокирована и помещена в очередь ожидания;

5. $A.Send$ – установить $E = 1$, активизировать все задачи из очереди ожидания, т.е. активизировать B и C.

С помощью событий легко организовать обмен между двумя задачами по типу клиент – сервер. Пусть есть два события E_1 и E_2. Задача T_1 (сервер) сразу после старта вызывает функцию $E_1.Wait()$. Тем самым она блокируется до получения события E_1. Задача T_2 (клиент) сразу после старта подготавливает данные в разделяемой с задачей T_1 памяти и

вызывает функцию $E_1.Send()$. Это приводит к активизации задачи T_1. Задача T_2 продолжает работу и вызывает функцию $E_2.Wait()$, блокируясь до получения события E_2. Задача T_1 (сервер) по окончании обработки данных вызывает функции $E_2.Send()$, $E_1.Wait()$, активизируя задачу T_2 (клиента) и блокируя себя. Ниже мы рассмотрим и другие способы реализации взаимодействия клиент – сервер.

5.4. Почтовые ящики

Определение. Почтовые ящики – это объект обмена данными между задачами, устроенный в виде очереди FIFO. С каждым почтовым ящиком связаны:

1. очередь сообщений (образующая содержимое почтового ящика),
2. очередь задач, ожидающих сообщений в почтовом ящике,
3. очередь задач, ожидающих освобождения места в почтовом ящике,
4. механизм взаимного исключения, обеспечивающий правильный доступ нескольких задач к одним и тем же сообщениям в почтовом ящике.

Количество задач, ожидающих сообщения в почтовом ящике, обычно не ограничено. Количество же сообщений в ящике обычно ограничено параметром, указанным при создании ящика. Это связано с тем, что размер каждого сообщения указывается при создании ящика и может быть значительным. Если ящик полон и задача пытается поместить в него новое сообщение, то она блокируется до тех пор, пока в ящике не появится свободное место. Над почтовыми ящиками определены следующие элементарные операции

- положить сообщение в почтовый ящик, при этом задача, вызвавшая эту операцию может быть блокирована и помещена в очередь задач, ожидающих освобождения места в ящике, если в ящике нет свободного места (активизация наступит сразу после взятия первого же сообщения из очереди сообщений); если очередь ожидающих сообщения задач непуста, то активизируются все задачи из этой очереди;
- попробовать положить сообщение в почтовый ящик – если в ящике есть свободное место, то эта операция эквивалентна положить, иначе вернуть признак отсутствия места;
- положить в начало – то же, что положить, только сообщение помещается в голову очереди;
- попробовать положить в начало – то же, что попробовать положить, только сообщение помещается в голову очереди;
- взять сообщение из почтового ящика, при этом задача, вызвавшая эту операцию может быть блокирована и помещена в очередь задач, ожидающих сообщения, если в ящике нет сообщений; при поступлении сообщения она будет активизирована; при этом, если в очереди ожидающих сообщения задач находится более одной задачи, то при выполнении операции взять обеспечивается механизм взаимного исключения задач (т.е. пока выполняется эта операция одной задачей все другие задачи, запросившие ту же операцию, заблокированы);
- попробовать взять сообщение из почтового ящика – если в ящике есть сообщения, то эта операция эквивалентна взять, иначе вернуть признак отсутствия сообщений;
- очистить ящик – удалить все сообщения из очереди.

Фактически описанный механизм представляет собой взаимодействие клиент–сервер, когда задачи–клиенты помещают запросы в почтовый ящик, а задачи–серверы их оттуда забирают.

5.5. Взаимодействие клиент – сервер

Определение. Взаимодействием клиент – сервер называется способ передачи информации от одной задачи к другой.

Ранее мы рассмотрели несколько способов обеспечения такого взаимодействия (см. семафоры, сигналы и почтовые ящики).

5.6. Очереди задач

Для обеспечения рассмотренных выше видов взаимодействия и синхронизации задач операционная система строит ряд очередей задач.

- **Очередь выполняемых задач** – это очередь задач, находящихся в состоянии исполнения (получающих процессорное время в соответствии с тем или иным алгоритмом разделения времени).
- **Очередь готовых задач** – это очередь задач, находящихся в состоянии готовности.
- **Очередь задач, ожидающих семафора** – строится для каждого созданного семафора.
- **Очередь задач, ожидающих события** – строится для каждого события.

Это базовые очереди, на основе которых строятся другие. Например, почтовые ящики часто реализуются на базе семафоров и событий, и с каждым из почтовых ящиков связаны по две очереди задач (ожидающих сообщений и ожидающих свободного места в ящике).

Эти очереди ведутся операционной системой, поэтому использование любого "объекта" взаимодействия и синхронизации, приводит к системным вызовам. При любом таком вызове исполнение проходит по ядру системы и менеджерам задач и памяти, и, следовательно, любой вызов сопряжен с дополнительными расходами.

5.7. Объекты синхронизации POSIX

Стандарт POSIX 1003.1b (см. раздел 3.2) определяет ряд объектов синхронизации, которые должны присутствовать в системах реального времени:

Семафоры:
: логически совпадают с описанными выше булевскими семафорами. Идентификатором семафора является имя, синтаксически устроенное как имя файла.

Очереди сообщений:
: логически совпадают с описанными выше почтовыми ящиками. Идентификатором очереди является имя, синтаксически устроенное как имя файла.

Разделяемая память:
: ее идентификатором является имя, синтаксически устроенное как имя файла.

Стандарт POSIX 1003.1c (см. раздел 3.2) определяет ряд объектов синхронизации, которые должны присутствовать в системах, использующих задачи (threads):

Объекты mutex:
: обеспечивают взаимное исключение (MUTual EXclusion) работающих задач (см. раздел 5.7.1).

Объекты condvar:
: обеспечивают взаимное исключение работающих задач при принятии условной переменной (CONDitional VARiable) определенного значения (см. раздел 5.7.2).

5.7.1. Объекты синхронизации типа mutex

Объекты синхронизации типа mutex фактически представляют собой некоторое развитие булевских семафоров в плане повышения безопасности и эффективности работы программы.

Типичный цикл работы с разделяемым ресурсом следующий: взять семафор, работать с ресурсом, вернуть семафор. Однако, если в результате ошибки в программе она вначале вызовет функцию вернуть семафор (не взяв его!), а затем выполнит приведенный выше цикл работы с разделяемым ресурсом, то функция взять семафор не блокирует задачу, если ресурс занят.

Другой проблемой при работе с семафорами является необходимость переключения задач при каждом вызове функций, работающих с семафором. Дело в том, что сам счетчик семафора s (фактически являющийся разделяемой между процессами памятью) находится в области данных операционной системы, и любая функция, работающая с семафором, является системным вызовом. Это особенно плохо при синхронизации между задачами (threads), поскольку в этом случае сам обмен данными не требует переключения задач (так как вся память у задач общая).

Для борьбы с этими явлениями вводится объект mutex, который фактически состоит из пары: булевского семафора и идентификатора задачи – текущего владельца семафора (т.е. той задачи, которая успешно вызвала функцию взять и стала владельцем разделяемого ресурса). При этом сама эта пара хранится в разделяемой между задачами памяти (в случае threads – в любом месте их общей памяти). Для доступа к объекту mutex m определены три примитивные операции:

- $Lock(m)$ – блокировать mutex m, если m уже заблокирован другой задачей, то эта операция переводит задачу в состояние ожидания разблокирования m;
- $Unlock(m)$ – разблокировать mutex m, (если m ожидается другой задачей, то она может быть активизирована, удалена из очереди ожидания и может вытеснить текущую задачу, например, если ее приоритет выше); если вызвавшая эту операцию задача не является владельцем m, то операция не имеет никакого эффекта;
- $TryLock(m)$ – попробовать блокировать mutex m, если m не блокирован, то эта операция эквивалентна $Lock(m)$, иначе возвращается признак неудачи.

Эти операции **неделимы**, т.е. переключение задач во время их исполнения запрещено.

Объекты mutex бывают:

- **локальными** – доступны для синхронизации между задачами (threads) одного процесса; размещаются в любом месте их общей памяти;
- **глобальными** – доступны для синхронизации между задачами (threads) разных процессов; размещаются в разделяемой между процессами памяти.

Отметим, что глобальные mutex предоставляются не всеми операционными системами.

Некоторые операционные системы предоставляют объекты mutex со специальными свойствами, полезными в ряде случаев. Рассмотрим следующую ситуацию. Функции `f1` и `f2` работают с разделяемыми ресурсами и используют mutex `m` для синхронизации между задачами:

```
f1()
{
  Lock(m);
  ...
  Unlock(m);
}

f2()
```

```
{
  Lock(m);
  ...
  Unlock(m);
}
```

В результате развития программы нам потребовалось вызвать функцию `f1` из `f2`:

```
f2()
{
  Lock(m);
  <операторы 1>
  f1();
  <операторы 2>
  Unlock(m);
}
```

Однако это приводит к ситуации **deadlock**: задача переходит в вечный цикл ожидания освобождения mutex `m` в функции `f1`, поскольку она уже является владельцем этого mutex. Поэтому некоторые операционные системы предоставляют объекты mutex дополнительных типов:

- **error check** – вызов *Lock* владельцем mutex, а также *Unlock* не владельцем не производят никакого действия; отметим, что mutex этого типа решит проблему, описанную выше, но появляется новая: `<операторы 2>` (может быть, работающие с разделяемым ресурсом) выполняются, когда задача уже не является владельцем mutex;
- **recurcive** – вызов *Lock* владельцем mutex увеличивает счетчик таких вызовов, вызов *Unlock* владельцем – уменьшает счетчик; реально разблокирование mutex происходит при значении счетчика, равном 0.

5.7.2. Объекты синхронизации типа condvar

Объект синхронизации типа condvar дает возможность задаче ожидать выполнения некоторых условий. Фактически он состоит из объекта – события E (см. раздел 5.3), с одним отличием: при поступлении события (посредством функции *Send*) только одна из очереди ожидающих события задач активизируется. Как и vutex, объект condvar явно размещается в разделяемой между задачами (процессами) памяти, что позволяет обойтись без системных вызовов при синхронизации между threads. Для доступа к объекту condvar E определены три примитивные операции:

- $Wait(E, m)$ (где m типа mutex) – производит следующие действия (первые два из них **неделимы**, т.е. переключение задач во время их исполнения запрещено):
  - вызвать $Unlock(m)$ для текущей задачи (т.е. вызвавшей эту операцию),
  - вызвать $Wait(E)$,
  - вызвать $Lock(m)$;
- $Signal(E)$ – вызвать $Send(E)$;
- $Broadcast(E)$ – вызвать $Send(E)$ и активизировать **все** ожидающие задачи (т.е. тот же эффект, что и для функции *Send*, описанной в разделе 5.3.

Рассмотрим пример использования объектов condvar. Пусть есть задачи $T_0, T_1, \ldots, T_n$, разделяющие общую область памяти X. Для синхронизации доступа к X используется mutex m. Пусть задача T_0 активизируется, только если выполнено некоторое условие $P(X)$. Для обеспечения этого взаимодействия используем объект condvar E. Тогда алгоритм работы T_0 может быть схематически записан так:

```
T_0 ()
{
  for (;;)
    {
      Lock (m);
      Wait (E, m);
      /* Можем работать с разделяемой памятью X, поскольку
         являемся владельцем mutex m */
      if (P(X))
        {
          /* Условие выполнено: исполняем необходимые действия */
          ...
        }
      Unlock (m);
    }
}
```

Тогда алгоритм работы T_i $(i = 1, \ldots, n$ может быть схематически записан так:

```
T_i ()
{
  for (;;)
    {
      Lock (m);
      /* Можем работать с разделяемой памятью X, поскольку
         являемся владельцем mutex m */
      /* Записываем данные в X */
      ...

      Signal (E);
      Unlock (m);
    }
}
```

5.8. Пример использования объектов синхронизации POSIX

Приводимая ниже программа считывает строку со стандартного ввода и выводит ее на стандартный вывод. Для чтения строки и для ее вывода создаются две задачи (thread), кодом для которых являются функции `reader` и `writer` соответственно. Поскольку задачи работают в том же адресном пространстве, что и создавший их процесс, то они автоматически разделяют все переменные, включая буфер сообщений `msgbuf`. Для организации взаимного исключения при доступе к критическим разделяемым ресурсам (переменным `done`, `msglen`, `msgbuf`) используется объект синхронизации mutex `mutx`.

```
#include <stdio.h>
#include <pthread.h>

/* Компиляция в POSIX-совместимых системах (например, Linux):  */
/* cc <имя_файла> -lpthread                                    */

#define BUF_LEN 256

pthread_mutex_t mutx;   /* Объект синхронизации типа mutex     */
int done;               /* Признак окончания работы            */
int msglen;             /* Длина сообщения                     */
```

```
char msgbuf[BUF_LEN];   /* Буфер для сообщения                  */

/* Ждать сообщения, по его поступлении вывести его на экран.
   Функция работает как независимая задача, вызываемая
   операционной системой, поэтому прототип фиксирован.
   Аргумент argp не используется. */
void * writer (void * argp)
{
  for (;;)
    {
      pthread_mutex_lock (&mutx);       /* "захватить" mutex   */
      if (done)
        {
          /* Напечатать идентификатор текущей задачи          */
          printf ("Thread %x exits\n", (int)pthread_self ());
          pthread_mutex_unlock (&mutx); /* "освободить" mutex  */
          pthread_exit (0);             /* завершить задачу    */
          return 0;                     /* никогда             */
        }
      if (msglen)
        {
          printf ("*> %s\n", msgbuf);   /* вывести на экран    */
          msglen = 0;
        }
      pthread_mutex_unlock (&mutx);     /* "освободить" mutex  */
      /* Поместить задачу в конец очереди готовых задач с тем  */
      /* же приоритетом                                       */
      sched_yield ();
    }
}

/* Считать сообщение и поместить его в буфер.
   Функция работает как независимая задача, вызываемая
   операционной системой, поэтому прототип фиксирован.
   Аргумент argp не используется. */
void * reader (void *argp)
{
  for (;;)
    {
      pthread_mutex_lock (&mutx);       /* "захватить" mutex   */
      if (!msglen)
        {
          /* Считать сообщение. Выход по Ctrl+D               */
          if (!fgets (msgbuf, BUF_LEN, stdin))
            break;
          msglen = strlen (msgbuf) + 1;
        }
      pthread_mutex_unlock (&mutx);     /* "освободить" mutex  */
      /* Поместить задачу в конец очереди готовых задач с тем  */
      /* же приоритетом                                       */
      sched_yield ();
    }
  /* Напечатать идентификатор текущей задачи                  */
  printf ("Thread %x exits\n", (int)pthread_self ());
```

```
  pthread_mutex_unlock (&mutx);         /* "освободить" mutex  */
  pthread_exit (0);                     /* завершить задачу    */
  return 0;                             /* никогда             */
}

int main ()
{
  pthread_t wtid, rtid;                 /* дескрипторы задач   */

  if (pthread_mutex_init (&mutx, 0))    /* создать mutex       */
    perror ("pthread_mutex_init");

  if (pthread_create (&wtid, 0, writer, 0))  /* создать задачу */
    perror ("pthread_create");
  if (pthread_create (&rtid, 0, reader, 0))  /* создать задачу */
    perror ("pthread_create");

  if (!pthread_join (rtid, 0))  /* ждать окончания задачи rtid */
    done = 1;

  pthread_join (wtid, 0);       /* ждать окончания задачи wtid */

  if (pthread_mutex_destroy (&mutx))    /* удалить mutex       */
    perror ("pthread_mutex_destroy");

  return 0;
}
```

Если во время работы этой программы посмотреть список задач в системе, то мы увидим, что этой программе их соответствует четыре (!):

1. запущенный процесс,
2. “главная” задача, соответствующая функции `main`,
3. задача, соответствующая функции `reader`,
4. задача, соответствующая функции `writer`.

Приведенная выше программа является не совсем корректной в следующем смысле. Каждая из функций `reader`/`writer` делает предположение о том, что вторая функция `writer`/`reader` в момент вызова функции `sched_yield` ждет только освобождения mutex `mutx`. Для данной программы это предположение верно, но в общем случае требуется обеспечить, чтобы после выполнения функции `reader`/`writer` следующей выполнялась `writer`/`reader`, если же последняя занята (не обработала предыдущее сообщение), то текущая задача должна перейти в состояние ожидания, а не потреблять процессорные ресурсы в бесполезном цикле опроса. Ниже приводится модифицированный вариант этой программы, в котором помимо объекта mutex для взаимного исключения при доступе к разделяемым данным используется объект condvar для синхронизации самих задач. Функцией `pthread_cond_wait` задача переходит в режим ожидания данных от другой задачи, а функцией `pthread_cond_signal` она сообщает о готовности данных.

```
#include <stdio.h>
#include <pthread.h>

/* Компиляция в POSIX-совместимых системах (например, Linux):  */
```

```
/* cc <имя_файла> -lpthread                                        */

#define BUF_LEN 256

pthread_mutex_t mutx;   /* Объект синхронизации типа mutex      */
pthread_cond_t  condx;  /* Объект синхронизации типа condvar    */
int done;               /* Признак окончания работы             */
int msglen;             /* Длина сообщения                      */
char msgbuf[BUF_LEN];   /* Буфер для сообщения                  */

/* Ждать сообщения, по его поступлении вывести его на экран.
   Функция работает как независимая задача, вызываемая
   операционной системой, поэтому прототип фиксирован.
   Аргумент argp не используется. */
void * writer (void * argp)
{
  for (;;)
    {
      pthread_mutex_lock (&mutx);       /* "захватить" mutex   */
      while (!msglen)
        {
          /* Освободить mutex и ждать сигнала от condvar,      */
          /* затем "захватить" mutex опять                     */
          pthread_cond_wait (&condx, &mutx);
          if (done)
            {
              /* Напечатать идентификатор текущей задачи       */
              printf ("Thread %x exits\n", (int)pthread_self ());
              pthread_mutex_unlock (&mutx);/*"освободить" mutex*/
              pthread_exit (0);            /* завершить задачу */
              return 0;                    /* никогда          */
            }
        }
      printf ("*> %s\n", msgbuf);       /* вывести на экран    */
      msglen = 0;
      /* Послать сигнал condvar                                */
      pthread_cond_signal (&condx);
      pthread_mutex_unlock (&mutx);     /* "освободить" mutex  */
    }
}

/* Считать сообщение и поместить его в буфер.
   Функция работает как независимая задача, вызываемая
   операционной системой, поэтому прототип фиксирован.
   Аргумент argp не используется. */
void * reader (void *argp)
{
  for (;;)
    {
      pthread_mutex_lock (&mutx);       /* "захватить" mutex   */
      if (!msglen)
        {
          /* Считать сообщение. Выход по Ctrl+D                */
          if (!fgets (msgbuf, BUF_LEN, stdin))
```

```
            break;
          msglen = strlen (msgbuf) + 1;
          /* Послать сигнал condvar                              */
          pthread_cond_signal (&condx);
        }
      while (msglen)
        {
          /* Освободить mutex и ждать сигнала от condvar,       */
          /* затем "захватить" mutex опять                      */
          pthread_cond_wait (&condx, &mutx);
        }
      pthread_mutex_unlock (&mutx);     /* "освободить" mutex  */
    }
  /* Напечатать идентификатор текущей задачи                    */
  printf ("Thread %x exits\n", (int)pthread_self ());
  pthread_mutex_unlock (&mutx);         /* "освободить" mutex  */
  pthread_exit (0);                     /* завершить задачу    */
  return 0;                             /* никогда             */
}

int main ()
{
  pthread_t wtid, rtid;                 /* дескрипторы задач   */

  if (pthread_mutex_init (&mutx, 0))    /* создать mutex       */
    perror ("pthread_mutex_init");
  if (pthread_cond_init (&condx, 0))    /* создать condvar     */
    perror ("pthread_cond_init");

  if (pthread_create (&wtid, 0, writer, 0))  /* создать задачу */
    perror ("pthread_create");
  if (pthread_create (&rtid, 0, reader, 0))  /* создать задачу */
    perror ("pthread_create");

  if (!pthread_join (rtid, 0))  /* ждать окончания задачи rtid */
    {
      done = 1;
      pthread_cond_signal (&condx);
    }

  pthread_join (wtid, 0);       /* ждать окончания задачи wtid */

  if (pthread_mutex_destroy (&mutx))    /* удалить mutex       */
    perror ("pthread_mutex_destroy");
  if (pthread_cond_destroy (&condx))
    perror ("pthread_cond_destroy");

  return 0;
}
```

6. Управление задачами

Управление задачами (процессами, процедурами обработки прерываний) является важнейшей функцией любой операционной системы. Именно специфический механизм планирования задач и процедур обработки прерываний делает операционную систему системой реального времени.

6.1. Планирование задач

Необходимость планирования задач появляется, как только в очереди активных (готовых) задач появляются более одной задачи (в многопроцессорных системах – более числа имеющихся процессоров). Алгоритм планирования задач является основным отличием систем реального времени от "обычных" операционных систем. В последних целью планирования является обеспечение выполнения *всех* задач из очереди готовых задач, обеспечивая иллюзию их параллельной работы и не допуская монополизацию процессора какой-либо из задач. В ОСРВ же целью планирования является обеспечение выполнения каждой готовой задачи **к определенному моменту времени**, при этом часто "параллельность" работы задач *не допускается*, поскольку тогда время исполнения задачи будет зависеть от наличия других задач.

Важнейшим требованием при планировании задач в ОСРВ является **предсказуемость** времени работы задачи. Это время не должно зависеть от текущей загруженности системы, количества задач в очередях ожидания (процессора, семафора, события, ...) и т.д. При этом желательно, чтобы длина этих очередей не была бы ограничена (т.е. ограничена только объемом памяти, доступной системе).

Определение. Планировщик задач (scheduler) – это модуль (программа), отвечающий за разделение времени имеющихся процессоров между выполняющимися задачами. Отвечает за коммутацию задач из состояния блокировки в состояние готовности, и за выбор задачи (задач – по числу процессоров) из числа готовых для исполнения процессором(ами).

В многопроцессорных системах количество очередей готовых задач (очередей ожидания) может зависеть от типа архитектуры системы. В симметричных многопроцессорных системах (SMP, Symmetric MultiProcessor system) обычно есть одна очередь ожидания для всех процессоров. В других системах может быть по одной очереди на процессор (или по одной очереди на группу процессоров, образующих SMP систему).

6.1.1. Приоритеты

Напомним, что приоритетом называется число, приписанное операционной системой (а именно, планировщиком задач) каждому процессу и задаче. Существуют несколько схем назначения приоритетов.

- **Фиксированные** приоритеты – приоритет задаче назначается при ее создании и не меняется в течение ее жизни. Эта схема с различными дополнениями применяется в большинстве систем реального времени. В схемах планирования ОСРВ часто требуется, чтобы приоритет каждой задачи был уникальным, поэтому часто ОСРВ имеют большое число приоритетов (обычно 255 и более).

- **Турнирное** определение приоритета – приоритет последней исполнявшейся задачи понижается.

- Определение приоритета по алгоритму **round robin** – приоритет задачи определяется ее начальным приоритетом и временем ее обслуживания. Чем больше задача обслуживается процессором, тем меньше ее приоритет (но не опускается ниже некоторого порогового значения). Эта схема в том или ином виде применяется в большинстве UNIX систем.

Отметим, что в разных системах различные алгоритмы планирования задач могут вводить новые схемы изменения приоритетов. Например, в системе OS-9 приоритеты ожидающих задач увеличиваются для избежания слишком больших времен ожидания.

Пример UNIX системы с фиксированными приоритетами: пусть готовая задача T_1 имеет приоритет 1, готовая задача T_2 имеет приоритет 5. Тогда задача T_1 получит 5/6 процессорного времени, а задача T_2 – 1/6,

Определение. Ситуацию, когда более приоритетная задача блокирована менее приоритетной, владеющей разделяемым ресурсом, требуемым приоритетной задаче, называют **инверсией приоритетов**. Все ОСРВ, осуществляющие планирование задач на основе их приоритетов, используют те или иные механизмы борьбы с этим явлением. Наиболее часто используют механизм т.н. **наследования приоритетов**, когда задача, владеющая разделяемым ресурсом временно получает приоритет более приоритетной задачи, ожидающей этот ресурс. Приоритет возвращается к прежнему значению, когда задача освобождает разделяемый ресурс.

6.1.2. Стратегии планирования задач

Рассмотрим способы, которыми планировщик выбирает очередную задачу для передачи на выполнение процессору. Ниже мы вначале рассмотрим "чистые" способы, обычно применяемые только в сочетании с другими.

- **Очередь ожидания типа FIFO**: первой будет исполнена первой поступившая задача.
- **Очередь ожидания, отсортированная по времени исполнения задач**: первой будет исполнена задача, исполнявшаяся до этого самое короткое время.
- **Несколько очередей ожидания** (одного из приведенных выше типов): поступившая задача попадает в одну из очередей в зависимости от приоритета задачи, выборка производится из головы первой очереди, если последняя пуста, то выбирается голова второй очереди и т.д.; часто задачи в разных очередях получают разный квант времени в зависимости от номера очереди (первая очередь – меньший квант, так как задачи в ней получают доступ к процессору чаще).
- **Очередь ожидания, отсортированная по приоритету задач**: первой будет исполнена задача, имеющая наивысший приоритет.

В UNIX системах применяется следующий способ планирования. Все процессорное время разбито на кванты фиксированной длины. Все готовые задачи получат свой квант времени, но частота этого зависит от общего количества готовых задач и их приоритета. Изначально задача получает высокий приоритет. Если в течение своего кванта задача использует процессор (не блокируется с самого начала), то ее приоритет не меняется или даже повышается. С другой стороны, если задача использует свой квант полностью (не блокируется до его окончания), то ее приоритет понижается. Такая стратегия обеспечивает высокую среднюю производительность системы и быстрое время реакции для интерактивных программ (типа текстовых редакторов), но абсолютно не годится для систем реального времени, поскольку процессорное время, получаемое задачей, зависит от количества других задач и их активности.

В системах реального времени обычно применяется следующий алгоритм планирования. Задачи имеют фиксированные приоритеты, которые могут динамически меняться самой задачей или планировщиком. Процессор получает задача, имеющая самый высокий приоритет. Если такая задача не одна (например, не все задачи имеют разный приоритет), то среди задач с самым высоким приоритетом организуется планирование с изменением приоритета по схеме round robin (так называемая схема RRS, round robin scheduling).

Подобная схема автоматически обеспечивает preemtion – задача с любым приоритетом (включая ядро системы) может быть прервана задачей с более высоким приоритетом.

6.1.3. Планирование периодических задач

Большинство задач в системах реального времени – периодические, активизируемые сигналом таймера или датчика. Для таких задач разработаны специальные приемы разделения времени.

Очевидно, что в однопроцессорной системе должно быть выполнено соотношение

$$\sum_{i=1}^{n} \frac{R_i}{T_i} < 1,$$

где T_i и R_i – соответственно период и максимальное время работы задачи i, n – количество задач в системе. Любая схема планирования должна обеспечивать выполнение этого соотношения.

В 1973г. Liu и Layland предложили метод анализа периодических задач в ОСРВ, названный RMA (Rate Monotonic Analysis), и соответствующую схему планирования, названную RMS (Rate Monotonic Scheduling). В исходном варианте в этой схеме предполагается, что все задачи в системе периодические и между ними отсутствует взаимодействие. Приоритет задачи в RMS обратно пропорционален периоду, т.е. задача имеет тем выше приоритет, чем короче ее период. При этом для обеспечения стабильности работы системы должно быть выполнено соотношение

$$\sum_{i=1}^{n} \frac{R_i}{T_i} < n(2^{1/n} - 1).$$

Отметим, что для реализации схемы RMS требуется, чтобы приоритеты у всех задач были различными.

В исходном варианте RMA используется нереалистичное предположение о независимости всех задач. Поэтому Liu и Layland расширили RMA на общий случай.

Если между задачами есть зависимость, то необходимо принимать меры для борьбы с инверсией приоритетов. Liu и Layland предложили использовать PCP (Priority Celling Protocol). Основные составляющие протокола:

- каждому разделяемому ресурсу и каждому приложению (набору задач) приписан уровень приоритетности (не путать с приоритетом);
- блокировка ресурса невозможна, если уровень его приоритетности выше, чем уровень приоритетности запросившего блокировку приложения;
- приоритет блокирующих (т.е. владеющих ресурсом) задач временно увеличивается.

Отметим, что чаще используется простой механизм наследования приоритетов (см. выше).

При условии применения этого протокола Liu и Layland показали, что планирование синхронных задач должно удовлетворять для каждого k следующему условию

$$\sum_{i=1}^{k} \frac{R_i}{T_i} + \frac{A_k}{T_k} < n(2^{1/k} - 1),$$

где A_k – это максимальное время ожидания задачи k, которое является суммой времен, проведенных задачами с меньшим приоритетом в критических секциях.

6.1.4. Разработка хорошо планируемых задач

Каким бы "умным" не был алгоритм назначения приоритетов и планирования неверно написанная задача может испортить общую загруженность системы. Рассмотрим, например, следующую систему. Пусть в системе есть два входа A_1, B_1 и два выхода A_2, B_2. Две задачи T_1, T_2 занимаются получением данных с входа, их преобразованием и передачей на соответствующий выход. Как будет распределено время между этими двумя задачами?

При использовании классического полинга (polling), когда задачи постоянно опрашивают вход на наличие данных, все процессорное время разделяется между ними.

При этом во время обработки данных для первой задачи данные второй могут быть потеряны, поскольку они не были взяты вовремя. Если же требуется запустить еще одну задачу T_3, то время для нее должно зависеть от того, были ли данные на входе задач T_1 или T_2 (поскольку от этого зависит время их работы и, соответственно, время, оставшееся до следующего запроса данных).

Однако, мы все равно имеем три постоянно работающие задачи. Наилучшее решение для планирования: 3 процессора, но это слишком дорого. Если же разделять время между этими задачами, то система будет использовать $2/3$ процессорного времени, даже если входных данных нет.

С точки зрения системы реального времени наилучшее решение – это активизация задач T_1 и T_2 прерыванием по готовности данных.

6.2. Переключение контекста

Контекст задачи – это набор данных, задающих состояние процессора при выполнении задачи. Обычно совпадает с набором регистров, доступных для изменения прикладной задаче. 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 зачастую термины “переключение задач” и “переключение контекста” употребляют как синонимы.

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

При переключении задач диспетчеру необходимо:

1. корректно остановить работающую задачу; для этого
   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 весьма 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.

Детерминированность особенно важна при обработке прерываний, поскольку их может быть несколько в очереди прерываний, и обработчик должен обслужить их все. Приложение должно знать, сколько времени это займет в наихудшем случае.

6.3. Прерывания

Прерывания являются основным источником сообщения внешним устройством 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.

7. Управление памятью

Использование виртуальной памяти является настолько удобным средством разделения задач и обеспечения им непрерывного адресного пространства, начинающегося с фиксированного адреса, что оно используется во многих системах.

Однако, наличие виртуальной памяти противоречит основным принципам ОСРВ.

- Даже если все страницы виртуальной памяти находятся в физической памяти, задержка, вносимая при трансляции адреса, не детерминирована. Она зависит от того, были ли уже обращения к данной странице (т.е. находится ли она в кэше трансляции страниц, называемом TLB, Translation Look-aside Buffers).

- Если же не все страницы виртуальной памяти находятся в физической памяти, то задержка, вносимая при трансляции адреса, не только не детерминирована, но и может быть очень большой (включает время загрузки страницы с диска). Для уменьшения этой проблемы, ОСРВ предоставляют специальные системные вызовы, "закрепляющие" указанные страницы в физической памяти (т.е. запрещающие переносить их на диск).

8. Обзор операционных систем реального времени

Рассмотрим некоторые из систем реального времени. По способу разработки программного обеспечения их разделяют на следующие категории:

- **Self-Hosted ОСРВ** – это системы, в которых пользователи могут разрабатывать приложения, работая в самой ОСРВ. Обычно это предполагает, что ОСРВ поддерживает файловую систему, средства ввода-вывода, пользовательский интерфейс, имеются компиляторы, отладчик, средства анализа программ, текстовые редакторы, работающие под управлением ОСРВ.

  Достоинством таких систем является более простой и наглядный механизм создания и запуска приложений, которые работают на той же машине, что и пользователь. Недостатком является то, что промышленному компьютеру во время его реальной эксплуатации часто вообще не требуется пользовательский интерфейс и возможность запуска

тяжеловесных программ вроде компилятора. Следовательно, большинство из описанных выше возможностей ОСРВ просто не используются и только зря занимают память и другие ресурсы компьютера.

Обычно self-hosted ОСРВ применяются на "обычных" компьютерах промышленного исполнения (см. описание оборудования на с. 9).

- **Host/Target ОСРВ** – это системы, в которых операционная система и(или) компьютер, на котором разрабатываются приложения (host), и операционная система и(или) компьютер, на котором запускаются приложения (target), различны. Связь между компьютерами осуществляется с помощью последовательного соединения (COM порта), ethernet, общей шины VME или compact PCI. В качестве host системы обычно выступают компьютер под управлением UNIX или Windows NT, в качестве target системы – промышленный или встраиваемый компьютер под управлением ОСРВ. Бывают системы, в которых на одном компьютере работают две операционных системы: "обычная" и реального времени.

  Достоинством таких систем является использование всех ресурсов "обычной" системы (таких, как графический интерфейс, файловая система, быстрый процессор и большой объем оперативной памяти) для создания приложений и уменьшение размеров ОСРВ за счет включения только нужных приложению компонент. Недостатком является относительная сложность программных компонент: кросс-компилятора, удаленного загрузчика и отладчика, и т.д.

Отметим, что, с одной стороны, рост мощности промышленных компьютеров позволяет использовать self-hosted системы на большем числе вычислительных систем. С другой стороны, увеличивающееся распространение встраиваемых систем (в разнообразном промышленном и бытовом оборудовании), расширяет сферу применения host/target систем (поскольку при больших объемах выпуска цена системы является определяющим фактором).

В зависимости от происхождения, ОСРВ разделяют на следующие группы.

- **Обычные ОС**, используемые в качестве ОСРВ. Часто к обычным ОС добавляют дополнительные модули, реализующие поддержку специфического оборудования (например, шины VME), а также планирование задач и обработку прерываний в соответствие с требованиями к ОСРВ и сглаживающие невозможность прервать ядро системы. Все такие системы относятся к разряду self-hosted.

- **Собственно ОСРВ** – специализированные операционные системы для применения в задачах реального времени. Бывают как self-hosted, так и host/target (большинство), некоторые ОСРВ поддерживают обе модели.

- **Специализированные (частные) ОСРВ** – это ОСРВ, разработанные для конкретного микроконтроллера его производителем. Часто не являются полноценными ОС, а представляют единый модуль с приложением и обеспечивают только необходимый минимум функциональности. Все такие системы относятся к разряду host/target.

По внутреннему строению различают "классические" и объектно-ориентированные системы.

8.1. "Классические" системы

Рассмотрим системы, основанные на классическом процедурном подходе к программированию.

8.1.1. CHORUS

Система CHORUS выпускается фирмой Chorus Systéme (Saint Quentin Yvelines, France). В ноябре 1997г. фирма приобретена компанией Sun Microsystems (Menlo Park, CA, USA). Основные характеристики:

1. *Тип*: host-target (CHORUS/Micro и CHORUS/ClassiX) и self-hosted (CHORUS/MiX)
2. *Архитектура*: на основе микроядра
3. *Стандарт*: собственный и POSIX 1003
4. *Свойства как ОСРВ*:
   - *Многозадачность*: POSIX 1003 (многопроцессность и многозадачность)
   - *Многопроцессорность*: да
   - *Уровней приоритетов*:
   - *Планирование*: приоритетное, FIFO; preemptive ядро
5. *ОС разработки (host)*: CHORUS/UNIX/Windows
6. *Процессоры (target)*: Intel 80x86, Motorola 68xxx, Motorola 88xxx, SPARC, PowerPC, T805, MIPS, PA-RISC, YMP (Cray), DEC Alpha
7. *Линии связи host-target*: последовательный канал и ethernet
8. *Минимальный размер*: 10Kb для CHORUS/Micro, 50Kb для CHORUS/ClassiX
9. *Средства синхронизации и взаимодействия*: POSIX 1003 (семафоры, mutex, condvar)
10. *Средства разработки*:
    - CHORUS/Harmony – интегрированная среда разработки C/C++, включающая компиляторы, отладчик, анализатор
    - CHORUS/JaZZ – виртуальная машина Java

8.1.2. LynxOS

Система LynxOS выпускается фирмой Lynx Real Time Systems (Los Gatos, USA). Основные характеристики:

1. *Тип*: self-hosted
2. *Архитектура*: на основе микроядра
3. *Стандарт*: POSIX 1003
4. *Свойства как ОСРВ*:
   - *Многозадачность*: POSIX 1003 (многопроцессность и многозадачность)
   - *Многопроцессорность*: да
   - *Уровней приоритетов*: 255
   - *Планирование*: FIFO, round robin, Quantum; preemptive ядро
5. *ОС разработки (host)*: –
6. *Процессоры (target)*: Intel 80x86, Motorola 68xxx, SPARC, PowerPC
7. *Линии связи host-target*: –
8. *Минимальный размер*:
   - полной системы: 256Kb
   - усеченной системы: 124Kb
   - только ядра: 33Kb

Систему можно записать в ROM.

9. *Средства синхронизации и взаимодействия*: POSIX 1003 (семафоры, mutex, condvar)
10. *Средства разработки*:
    - Комплекты разработчика, включающие компилятор C/C++, отладчик, анализатор
    - X Windows/Motif для Lynx
    - TotalView – многопроцессный отладчик

8.1.3. QNX

Система QNX выпускается фирмой QNX SoftWare Systems (USA). Основные характеристики:

1. *Тип*: self-hosted
2. *Архитектура*: на основе микроядра
3. *Стандарт*: POSIX 1003
4. *Свойства как ОСРВ*:
    - *Многозадачность*: POSIX 1003 (многопроцессность и многозадачность)
    - *Многопроцессорность*: да
    - *Уровней приоритетов*: 32
    - *Планирование*: FIFO, round robin, адаптивное; preemptive ядро
5. *ОС разработки (host)*: –
6. *Процессоры (target)*: Intel 80x86
7. *Линии связи host-target*: –
8. *Минимальный размер*: 60Kb
9. *Средства синхронизации и взаимодействия*: POSIX 1003 (семафоры, mutex, condvar)
10. *Средства разработки*:
    - Комплекты разработчика, включающие компилятор C/C++, отладчик, анализатор от QNX и независимых поставщиков (например, Watcom/SyBase);
    - X Windows/Motif для QNX.

8.1.4. OS-9

Система OS-9 выпускается фирмой Microware (USA). Основные характеристики:

1. *Тип*: host-target
2. *Архитектура*: на основе микроядра
3. *Стандарт*: собственный, вызовы похожи на UNIX
4. *Свойства как ОСРВ*:
    - *Многозадачность*: многопроцессность
    - *Многопроцессорность*:

- *Уровней приоритетов*:
- *Планирование*: приоритетное, FIFO, специальный механизм планирования (см. раздел 6.1.1); preemptive ядро

5. *ОС разработки (host)*: UNIX/Windows
6. *Процессоры (target)*: Motorola 68xxx, Intel 80x86, ARM, MIPS, PowerPC
7. *Линии связи host-target*: последовательный канал и ethernet
8. *Минимальный размер*: 16Kb
9. *Средства синхронизации и взаимодействия*: разделяемая память, сигналы, семафоры, события, ...
10. *Средства разработки*:
    - Hawk – интегрированная среда разработки на C/C++
    - PersonalJava – виртуальная машина Java

8.1.5. pSOS

Система pSOS выпускается Integrated Systems (Santa Clara, USA). В феврале 2000г. фирма приобретена компанией Wind River Systems (Alameda, CA, USA). Основные характеристики:

1. *Тип*: host-target
2. *Архитектура*: на основе микроядра
3. *Стандарт*: собственный
4. *Свойства как ОСРВ*:
    - *Многозадачность*: многопроцессность и многозадачность
    - *Многопроцессорность*: да
    - *Уровней приоритетов*: 255
    - *Планирование*: приоритетное; preemptive ядро
5. *ОС разработки (host)*: UNIX/Windows
6. *Процессоры (target)*: Motorola 68xxx, Intel 80x86, Intel 80960, ARM, MIPS, PowerPC
7. *Линии связи host-target*:
8. *Минимальный размер*: 15Kb
9. *Средства синхронизации и взаимодействия*: семафоры, mutex, события, ...
10. *Средства разработки*:
    - Интегрированная среда разработки на C/C++/Ada

8.1.6. RTC

Система RTC (Real Time Craft) выпускается фирмой GSI-TECSI (Paris, France). Основные характеристики:

1. *Тип*: host-target (RTC) и self-hosted (RTC/PC)
2. *Архитектура*: монолитная
3. *Стандарт*: SCEPTRE (RTC) и собственный (RTC/PC)
4. *Свойства как ОСРВ*:
   - *Многозадачность*: многопроцессность и многозадачность
   - *Многопроцессорность*: да (RTC); нет (RTC/PC)
   - *Уровней приоритетов*: 65532
   - *Планирование*: приоритетное, FIFO; preemptive ядро
5. *ОС разработки (host)*: UNIX/Windows
6. *Процессоры (target)*: Intel 80x86, Motorola 68xxx, Intel 80C166, Am39K (для RTC/PC – только Intel 80x86)
7. *Линии связи host-target*: ethernet
8. *Минимальный размер*: 1.5Kb (RTC); 4Kb (RTC/PC)
9. *Средства синхронизации и взаимодействия*: SCEPTRE (семафоры, сигналы, почтовые ящики)
10. *Средства разработки*:
    - Компилятор C, отладчик Watcom TRC.

8.1.7. VRTX

Система VRTX выпускается фирмой Ready Systems (Sunnyvale, USA). Основные характеристики:

1. *Тип*: host-target
2. *Архитектура*:
3. *Стандарт*: собственный
4. *Свойства как ОСРВ*:
   - *Многозадачность*: многопроцессность и многозадачность
   - *Многопроцессорность*: нет
   - *Уровней приоритетов*: 255
   - *Планирование*: приоритетное; preemptive ядро
5. *ОС разработки (host)*: UNIX/Windows
6. *Процессоры (target)*: Motorola 68xxx, Intel 80x86, Intel 80960, PowerPC
7. *Линии связи host-target*: последовательный канал, ethernet, шина VME
8. *Минимальный размер*: 16Kb

9. *Средства синхронизации и взаимодействия*: семафоры, очереди, сигналы, ...

10. *Средства разработки*:

    - MasterWorks – интегрированная среда разработки
    - Xray – специализированный отладчик
    - Simulator Xray – эмулятор ядра

8.1.8. VxWorks

Система VxWorks выпускается фирмой Wind River Systems (Alameda, CA, USA). Основные характеристики:

1. *Тип*: host-target

2. *Архитектура*: монолитная

3. *Стандарт*: собственный и POSIX 1003

4. *Свойства как ОСРВ*:

    - *Многозадачность*: многопроцессность и многозадачность
    - *Многопроцессорность*: да
    - *Уровней приоритетов*: 256
    - *Планирование*: приоритетное; preemptive ядро

5. *ОС разработки (host)*: UNIX/Windows

6. *Процессоры (target)*: Motorola 68xxx, Intel 80x86, Intel 80960, PowerPC, SPARC, Alpha, MIPS, ARM

7. *Линии связи host-target*: последовательный канал, ethernet, шина VME

8. *Минимальный размер*: 5–8Kb

9. *Средства синхронизации и взаимодействия*: семафоры POSIX 1003, очереди, сигналы, ...

10. *Средства разработки*:

    - TORNADO – интегрированная среда разработки C/C++
    - VxSim – эмулятор для UNIX
    - WindView – графический визуализатор состояния задач

8.2. Объектно-ориентированные системы

Рассмотрим системы, основанные на объектно-ориентированном подходе к разработке програмного обеспечения (см. раздел 4.1). К их числу мы будем относить системы, не только предоставляющие средства разработки и наборы библиотек для объектно-ориентированных языков, но и сами написанные на таких языках.

8.2.1. SoftKernel

Система SoftKernel выпускается фирмой Microprocess (Courbevoie, France). Основные характеристики:

1. *Тип*: host-target
2. *Архитектура*: на основе объектов-микроядер
3. *Стандарт*: собственный
4. *Свойства как ОСРВ*:
   - *Многозадачность*: многопроцессность и многозадачность
   - *Многопроцессорность*: да
   - *Уровней приоритетов*: 255
   - *Планирование*: приоритетное, FIFO; preemptive ядро
5. *ОС разработки (host)*: UNIX/Windows
6. *Процессоры (target)*: Motorola 68xxx, PowerPC, Intel 80960, ARM, SPARC
7. *Линии связи host-target*: последовательный канал и ethernet
8. *Минимальный размер*: 40Kb
9. *Средства синхронизации и взаимодействия*: семафоры, сигналы, события, почтовые ящики
10. *Средства разработки*:
    - SoftWorks – интегрированная среда разработки C/C++

8.3. Специализированные (частные) ОСРВ

Системы, проектируемые под конкретную модель микроконтроллера или конкретную задачу, обладают определенными преимуществами:

- наивысшая производительность,
- наилучший учет особенностей оборудования,
- наибольшая компактность.

Недостатки

- большое время разработки,
- высокая стоимость,
- непереносимость.

Примерами таких систем являются ОСРВ, разработанные многими производителями электронной техники (Sony, Sagem, ...), а также системы, разработанные под конкретную большую задачу (например, управление железными дорогами TGV во Франции).

8.4. Системы на основе Linux

Рассмотрим системы на основе Linux – свободно распространяемой версии системы UNIX. Она получила значительное распространение на настольных компьютерах в силу своей бесплатности и качества. Появившись на машинах с процессорами Intel 80x86, сейчас она поддерживает процессоры Alpha, SPARC, PowerPC, ARM, Motorola 68xxx, MIPS. Открытость исходных текстов системы позволяет реализовывать на ее основе специализированные системы и обеспечивать поддержку нового оборудования.

Приспособление системы Linux к требованиям "реального времени" происходит по следующим трем направлениям.

1. **Поддержка стандартов POSIX**, касающихся систем реального времени. Стандарт POSIX 1003.1c (thread – работа с задачами) уже поддержан, стандарт POSIX 1003.1b (расширения реального времени) поддержан лишь частично: реализованы механизмы управления памятью и механизмы планирования задач, механизмы же работы с таймерами, сигналы, POSIX семафоры и очереди сообщений пока не реализованы.

2. **Поддержка специального оборудования**, важнейшим из которых является шина **VME**. Уже существует поддержка моста VME–PCI. Ведутся разработки по обеспечению выполнения Linux из ПЗУ. Также для систем реального времени важным является повышение разрешения таймера системы.

3. **Реализация механизма preemption для ядра системы**. Этот механизм, с одной стороны, является необходимым для того, чтобы систему можно было называть системой реального времени, а с другой стороны, он является очень сложным для реализации. Linux, как и все UNIX системы, надолго запрещает прерывания при входе в ядро системы, и является существенно не preemptive.

Существует несколько проектов реализации premption для ядра Linux. По способу решения задачи их можно разделить на две группы.

1. Механизм preemption реализуется путем переписывания ядра системы (благо оно доступно в исходных текстах). На этом пути можно достичь самых качественных результатов, но на данный момент значительных успехов в этом плане нет по следующим причинам:

   а) слишком большой объем работы, связанный с большим объемом ядра;

   б) слишком высокая скорость изменения ядра, причем изменения вносятся, не учитывая интересы реального времени.

2. Механизм preemption реализуется путем написания микроядра, отвечающего за диспетчеризацию прерываний и задач. Ядро Linux работает как задача с низким приоритетом. Само ядро лишь незначительно изменено, для предотвращения блокирования им аппаратных прерываний. Задачи в такой системе разделены на две группы:

   а) процессы, работающие под управлением только микроядра (не использующие функции ядра Linux); эти процессы удовлетворяют требованиям реального времени, поскольку могут прерывать ядро Linux;

   б) процессы, работающие под управлением Linux (обычные приложения, которые в данной системе рассматриваются как подзадачи процесса – ядра Linux), а также задачи работающие под управлением микроядра, но использующие функции Linux; эти процессы не удовлетворяют требованиям реального времени, поскольку могут быть блокированы ядром Linux.

   Недостатком такого подхода является необходимость реализации микроядра, обеспечивающего функционирования процессов реального времени. Например, если процесс

реального времени хочет работать с коммуникационным портом (например, COM портом), то драйвер этого порта надо перенести из ядра Linux в микроядро (поскольку использование драйвера Linux сделает этот процесс зависимым от текущей загруженности системы и других непредсказуемых факторов). Наиболее законченной реализацией этого подхода является проект RT-Linux.

8.4.1. RT-Linux

Система RT-Linux является свободно распространяемой, разрабатываемой энтузиастами в ряде университетов мира. Создана в New Mexico Institute of Mining and Technology (USA).

Представляет собой простейшее микроядро, отвечающее за создание и планирование задач, обеспечение их взаимодействия и диспетчеризацию прерываний. Реализован простейший приоритетный механизм планирования и единственный механизм взаимодействия – очередь сообщений FIFO. Ядро Linux работает как самая низкоприоритетная задача. Само ядро подправлено: макроопределения, отвечающие за запрещение/разрешение прерываний заменены на вызов соответствующих функций микроядра. Задачи Linux не могут прервать ядро Linux, задачи же микроядра – могут.

Типичный механизм функционирования системы следующий. Критическая часть приложения (например, получение данных с датчика) реализуется как процесс микроядра. Часть, отвечающая за визуализацию и хранение данных, реализуется как процесс Linux. Эти два процесса взаимодействуют посредством очереди сообщений FIFO.

Простота микроядра позволяет разработчикам легко его модифицировать для удовлетворения своих потребностей.

Минимальный размер системы для записи в ПЗУ (без X-Windows) – 2.7Mb.

8.5. "Системы" на основе Windows NT (Microsoft)

Аргументы Microsoft "за"использование Windows NT в качестве ОСРВ.

- Многопроцессность и многозадачность системы.
- Поддержка многопроцессорности.
- Preemption задач.
- Preemption прерываний и возможность их маскирования. Распределение прерываний между процессорами в многопроцессорной системе. Для уменьшения времени пребывания в процедуре обработки прерывания (Interrupt Service Routine, ISR) обработчик вызывает специальную процедуру (Deffered Procedure Call, DPC) и заканчивает работу. Основная деятельность по обработке прерывания выполняется DPC. Процедуре DPC назначается приоритет и она начинает выполняться, управляемая планировщиком.
- Асинхронный ввод/вывод.
- Прямой доступ к оборудованию посредством интерфейса HAL (Hardware Abstraction Level). HAL обеспечивает изоляцию приложения от деталей реализации оборудования, обеспечивая платформенно-независимый прямой доступ к оборудованию.
- Специальная схема приоритетов. Все приоритеты разделены на две группы:
  1. Класс динамических приоритетов (16): приоритеты от 0 до 15 (0 – самый низкий). Эти приоритеты динамически меняются планировщиком по алгоритму, близкому к принятому в UNIX системах (см. раздел 6.1.2).
  2. Класс приоритетов реального времени (7): приоритеты 16, 22–26, 31. Эти приоритеты фиксированы, задачи из этого класса планируются на основе приоритетной очереди и получают управление раньше задач с динамическими приоритетами.

- Пространство ввода-вывода для задач из класса реального времени не участвует в страничном обмене механизма виртуальной памяти.

- Для закрепления страниц задачи в памяти существует специальный системный вызов (`VirtualLock()`).

- Предоставляются объекты синхронизации: критические секции, таймеры, события, mutex, ...

Аргументы "против"использования Windows NT в качестве ОСРВ.

- Ядро системы не preemptive.

- Недостатки механизма DPC:

  1. Все процедуры DPC, вызываемые обработчиком прерываний, получают один и тот же приоритет (из-за малого количества приоритетов) и обрабатываются планировщиком в порядке поступления (FIFO). Тем самым низкоприоритетные прерывания будут обрабатываться ранее высокоприоритетных, но поступивших позднее.
  2. Система не дает возможности узнать, сколько DPC стоит в очереди, поэтому нельзя узнать, когда начнет обрабатываться прерывание. Таким образом, существует случайная задержка между приходом прерывания и началом его обработки. Отметим, что для таких быстродействующих устройств, как дисковые и сетевые контроллеры, задержка в несколько миллисекунд может оказаться критической.
  3. Для каждого прерывания только один экземпляр DPC может быть в очереди. Следовательно, процедура DPC должна уметь обрабатывать повторяющиеся прерывания, а оборудование должно уметь буферизировать прерывания во избежание потери данных. Это удорожает как драйверы устройств, так и оборудование.

  Отметим, что несмотря на недостатки DPC, Windows NT вынуждена выносить обработку прерываний из ISR в DPC, поскольку во время ISR все прерывания запрещены и их можно потерять.

- Малое количество приоритетов в классе реального времени (7) приводит к тому, что много задач будут иметь одинаковый приоритет и планироваться алгоритмом типа round robin. Следовательно, время до начала исполнения задачи будет случайной величиной (зависит от текущей загруженности системы).

- Не решена проблема инверсии приоритетов (см. раздел 6.1.1). Вместо традиционного для ОСРВ механизма наследования приоритетов (см. раздел 6.1.1) Windows NT назначает задаче, из-за низкоприоритетности простаивающей в течение некоторого периода времени, случайный уровень приоритета, позволяющий ей начать работу. Это непредсказуемо и неприемлемо для ОСРВ. Поскольку приоритет задач класса реального времени не меняется, то этот механизм действует только на задачи из динамического класса. Тем самым ситуация только ухудшается: приоритет низкоприоритетных задач реального времени не меняется, а высокоприоритетные задачи могут быть вытеснены совсем низкоприоритетными задачами из динамического класса.

- Высокоприоритетные задачи могут блокироваться низкоприоритетными. Дело в том, что приоритеты задач из класса реального времени все-таки могут меняться. Некоторые компоненты ядра работают на уровне приоритета динамического класса. Следовательно, некоторые системные вызовы приводят к понижению приоритета и выводу задачи из класса реального времени. При этом она может быть заблокирована другой задачей, имевшей до этого более низкий приоритет.

- Все страницы неактивного процесса (например, ожидающего данных), могут быть перенесены на диск, несмотря на закрепление их вызовом `VirtualLock()`. Это приводит к случайным задержкам при активизации процесса (например, при поступлении ожидавшихся данных).
- Для Windows NT официально не приводятся времена системных вызовов и времена блокирования прерываний.
- Систему невозможно использовать без дисплея и клавиатуры.
- Система предъявляет слишком большие для ОСРВ запросы на память.

Для устранения этих недостатков ряд компаний предлагает программные (аппаратные) средства. Основные идеи их построения те же, что и для Linux. Поскольку исходный код системы недоступен, то, в отличие от Linux, существует только один способ приспособить систему к требованиям реального времени: разработать микроядро, обеспечивающее надлежащее планирование задач и диспетчеризацию прерываний, а Window NT выполнять как процесс.

8.5.1. Hyperkernel

Система Hyperkernel выпускается фирмой Nematron. Представляет собой ядро, обеспечивающее детерминистичное планирование и работающее на уровне привилегий 0 процессора Intel 80x86 вместе с Windows NT. Задачи Hyperkernel не видны Windows NT. Для них определены 8 уровней приоритетов с preemptive планированием. В качестве средства разработки используются стандартные для Windows NT компиляторы Visual C/C++ и специальные библиотеки. Используется API Win32 и стандартный HAL. Разрешение таймера: 1 микросекунда, минимальный квант времени 20 микросекунд. Время задержки реакции на прерывание: 5 микросекунд, переключение контекста – 4 микросекунды на Intel Pentium 133Mhz.

8.5.2. LP RT-Technology

Система LP RT-Technology выпускается фирмой LP Elektronik GmbH и включает три компоненты.

1. Плату для шины ISA, обеспечивающую 7 дополнительных уровней прерываний. Для взаимодействия с остальной системой плата использует NMI – немаскируемое прерывание процессора Intel 80x86.
2. **LP-RTWin Toolkit** – комплект разработчика ISR, используемый совместно с Visual C/C++ и отладчиком SoftICE фирмы NuMega.
3. **LP-VxWin RTAcc** – программный комплекс, обеспечивающий сосуществование Windows NT и VxWorks на одном PC. Две операционные системы взаимодействуют посредством протокола TCP/IP через разделяемую память. В качестве средства разработки используется Tornado – комплект разработчика для VxWorks.

8.5.3. Realtime ETS Kernel

Система Realtime ETS Kernel выпускается фирмой Phar Lap SoftWare в двух вариантах.

1. **TNT Embedded ToolSuite, Realtime Edition**, включающий: Realtime ETS Kernel, ETS TCP/IP, отладчик CodeView с поддержкой Borland Turbo Debugger ассемблер 386ASM, linker, поддержку компиляторов Visual C/C++, Borland C/C++, Watcom C/C++ и API Win32.
2. **Realtime ETS Kernel** – полная замена Windows NT, включает: компактное ядро (28Kb), поддерживающее Win32 API и использующее стандартные библиотеки компиляторов; ядро имеет 32 уровня приоритетов и может быть записано в ПЗУ.

8.5.4. Component Integrator

Система Component Integrator выпускается фирмой VenturCom, Inc. В отличие от предыдущих систем, здесь не вводится новое ядро, а предлагаются пакеты, расширяющие ряд узких мест Windows NT.

1. **Embedded Component Kit (ECK)**, включающий: поддержку работы без дисплея и клавиатуры, уменьшение потребности в памяти, отключение страничного обмена, поддержку флеш-памяти (возможность загрузки с флеш-памяти объемом 10Mb) и шины VME.

2. **Real-time Extension (RTX)**, включающий: поддержку таймеров (разрешение – 1 микросекунда, минимальный квант времени – 100 микросекунд), отключение виртуальной памяти, работа с физической памятью (т.е. тождественное отображение виртуальной памяти на физическую), 128 новых уровней привилегий. Время задержки реакции на прерывание: 5–20 микросекунд,

8.5.5. Willows RT

Система Willows RT выпускается фирмой Willows SoftWare, Inc. В отличие от предыдущих систем, здесь приложение вообще не будет запускаться в Window NT. Windows NT используется только как платформа разработки, приложение же будет работать под управлением настоящей ОСРВ (QNX, VxWorks, ...). Система состоит из трех частей.

1. Библиотеки TWIN32 и TWIN16, заменяющие библиотеки Microsoft и реализующие API.

2. TWIN Platform Abstraction Layer, ядро, обеспечивающее системные вызовы Microsoft.

3. TWIN Binary Interface, обеспечивающий трансляцию вызовов Platform Abstraction Layer в API используемой ОСРВ.

9. Языки разработки для систем реального времени

Основными критериями при выборе языка для разработки приложения реального времени являются:

1. **Получение наивысшей производительности** приложения реального времени. Из этого требования как правило 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. Отметим, что для современных процессоров качество компилятора особенно важно, поскольку для них оптимизация может ускорять работу программы в несколько раз по сравнению с неоптимизированным вариантом, причем часто оптимизирующий компилятор может породить код более быстрый, чем написанный на ассемблере человеком. 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.

Часто помимо перечисленных выше технических требований выбор языка разработки диктуется организационно-финансовыми мотивами:

1. **Выбирать приходиться из списка поддерживаемых** языков, которыми обычно являются C и C++. У поставщика ОСРВ бывает можно заказать поддержку нового языка, но это будет стоить очень дорого.

2. **Выбор языка может диктоваться наличием значительно объема программного обеспечения на этом языке**. Обычно фирма-разработчик ПО (программного обеспечения) уже имеет архив предыдущих разработок, и язык, на котором они написаны, диктует в этом случае язык для новых разработок.

3. **Выбор языка может диктоваться отраслевым стандартом** в той области, для которой пишется приложение. Например, для разработок в военной области в США принят язык Ada, и все ПО для этой сферы пишется на этом языке.

Основные языки разработки для ОСРВ:

- **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.

- **C**. Обеспечивает получение высокой производительности за счет хорошо разработанных оптимизирующих компиляторов, которые для современных процессоров часто дают код более эффективный, чем написанный на ассемблере. Язык C 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.

- **C++**. Включает язык C как подмножество и наследует все его положительные качества. C++ добавляет поддержку объектно-ориентированного подхода на уровне языковых конструкций.

- **Java**. Как язык интерпретируемого типа, имеет очень низкую эффективность получаемого кода. Доступ оборудованию и вызовы процедур на других языках – только посредством библиотечных функций (обычно написанных на C). Java обеспечивает наивысшую переносимость приложения – на уровне двоичного кода, и является объектно-ориентированным языком.

- **Ada**. Язык Ada разрабатывался (в 1975–1979гг.) специально для применения во встраиваемых системах и системах реального времени министерства обороны США. Обеспечивает получение высокой производительности за счет хорошо разработанных оптимизирующих компиляторов. Язык Ada дает прямой доступ к оборудованию и возможность вызова процедур на других языках (что, однако, не приветствуется стандартом). Приложения получаются переносимыми за счет ежегодной аттестации компиляторов на

удовлетворение стандарту. Однако, объектно-ориентированный подход на уровне языковых конструкций отсутствует.

- **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.

10. Среды разработки для систем реального времени

К средам разработки в системах реального времени уделяют значительно больше внимания, чем в "обычных" системах. Это связано как со сложностью и ответственностью разрабатываемых приложений, так и со сложностью модели разработки, когда платформа, где разрабатывается приложение, отличается от платформы, где оно запускается. Основные требования к средам разработки для ОСРВ:

1. **Поддержка выбранного языка программирования**.
2. **Обеспечение совместной работы** коллектива разработчиков над одним проектом.
3. **Обеспечение управления проектом**: добавление/удаление файлов с автоматической генерацией makefile-ов, контроль версий, ведение нескольких конфигураций.
4. **Обеспечение разработки для нескольких платформ**: разделение результатов трансляции одного приложения для разных целевых систем.
5. **Поддержка запуска и отладки приложений** (напомним, часто система, где приложение запускается, отличается от системы, где оно разрабатывается).
6. **Управление документацией**: средства автоматической генерации документации и разнообразных отчетов.
7. **Возможность подключения внешних утилит**: возможность использования альтернативного редактора, компилятора, отладчика, средства контроля версий и т.д. Это позволяет использовать одну и ту же среду разработки на разных платформах.

11. Архитектуры процессоров

Рассмотрим архитектуры основных процессоров, применяемых в ОСРВ.

11.1. Основные черты архитектуры и их влияние на системы реального времени

Рассмотрим основные пути, которыми разработчики процессоров пытаются повысить их производительность и то, как это влияет на системы реального времени.

11.1.1. CISC и RISC процессоры

Основной временно́й характеристикой для процессора является время цикла, равное $1/\mathbf{F}$, где $\mathbf{F}$ – тактовая частота процессора. Время, затрачиваемое процессором на задачу, может быть вычислено по формуле $\mathbf{C}*\mathbf{T}*\mathbf{I}$, где $\mathbf{C}$ – число циклов на одну инструкцию, $\mathbf{T}$ – время на один цикл, $\mathbf{I}$ – число инструкций на задачу.

Разработчики "классических" систем (которые теперь называют CISC (Complete Instruction Set Computer)) стремились уменьшить фактор $\mathbf{I}$. В процессорах реализовывались все более сложные инструкции, для выполнения которых внутри него самого запускались специальные процедуры (так называемый микрокод), загружаемые из ПЗУ внутри процессора. Этому пути способствовало то, что улучшения в технике производства полупроводников делали возможным реализацию все более сложных интегрированных цепей. Однако, на этом пути очень трудно уменьшить два других фактора: $\mathbf{C}$ поскольку инструкции сложные и требуют *программного декодирования* и $\mathbf{T}$ в силу *аппаратной сложности*.

Концепция RISC (Reduced Instruction Set Computer) возникла из статистического анализа того, как программное обеспечение использует ресурсы процессора. Исследования системных ядер и объектных модулей, порожденных оптимизирующими компиляторами, показали подавляющее доминирование простейших инструкций даже в коде для CISC машин. Сложные инструкции используются редко, поскольку микрокод обычно не содержит в точности те процедуры, которые нужны для поддержки различных языков высокого уровня и сред исполнения программ. Поэтому разработчики RISC процессоров убрали реализованные в микрокоде процедуры и передали программному обеспечению низкоуровневое управление машиной. Это позволило заменить процессорный микрокод в ПЗУ на подпрограмму в более быстрой ОЗУ.

Разработчики RISC процессоров улучшили производительность за счет уменьшения двух факторов: $\mathbf{C}$ (за использования только простых инструкций) и $\mathbf{T}$ (за счет упрощения процессора). Однако, изменения, внесенные для уменьшения числа циклов на инструкцию и времени на цикл, имеют тенденцию к увеличению числа инструкций на задачу. Этот момент был в центре внимания критиков RISC архитектуры. Однако, использование *оптимизирующих компиляторов* и других технических приемов, практически ликвидирует эту проблему.

11.1.2. Основные черты RISC архитектуры

У RISC процессора все инструкции имеют одинаковый формат и состоят из битовых полей, определяющих код инструкции и идентифицирующих ее операнды. В силу этого *декодирование инструкций производится аппаратно*, т.е. микрокод не требуется. При этом в силу одинакового строения всех инструкций процессор может декодировать несколько полей *одновременно* для ускорения этого процесса.

Инструкции, производящие операции в памяти, обычно либо увеличивают время цикла, либо число циклов на инструкцию. Такие инструкции требуют дополнительного времени для своего исполнения, так как требуется вычислить адреса операндов, считать их из памяти, вычислить результат операции и записать его обратно в память. Для уменьшения негативного влияния таких инструкций, разработчики RISC процессоров выбрали *архитектуру чтение/запись*, в которой все операции выполняются над операндами в регистрах процессора, а основная память доступна только посредством инструкций чтения/записи. Для эффективности этого подхода RISC процессоры имеют *большое количество регистров*. Архитектура чтение/запись также позволяет *уменьшить количество режимов адресации памяти*, что позволяет упростить декодирование инструкций.

Для CISC архитектур время исполнения инструкции обычно измеряется в числе циклов на инструкцию. Разработчики RISC архитектур, однако, стремились получить скорость выполнения инструкции, равную *одной инструкции за цикл*.

Для RISC процессора во многих случаях только наличие *оптимизирующего компилятора* позволяет реализовать все его возможности. Отметим, что компилятор может наилучшим

образом оптимизировать код именно для RISC архитектур (в силу их простоты). Программирование на языке ассемблера исчезает для RISC приложений, так как компиляторы языков высокого уровня могут производить очень сильную оптимизацию.

11.1.3. Конвейеризация

Конвейеризация является одним из основных способов повышения производительности процессора. Конвейерный процессор принимает новую инструкцию каждый цикл даже если предыдущие инструкции не завершены. В результате выполнение нескольких инструкций перекрывается и в процессоре находятся сразу несколько инструкций в разной степени готовности.

Исполнение инструкций может быть разделено на несколько стадий: **выборка**, **декодирование**, **исполнение**, **запись результатов**. **Конвейер инструкций** может уменьшить число циклов на инструкцию посредством одновременного исполнения нескольких инструкций, находящихся на разных стадиях. При правильной аппаратной реализации конвейер, имеющий n стадий может одновременно исполнять n последовательных инструкций. Новая инструкция может приниматься к исполнению на каждом цикле, и эффективная скорость исполнения, таким образом, есть один цикл на инструкцию. Однако, это предполагает, что конвейер всегда заполнен полезными инструкциями и нет задержек в прохождении инструкций через конвейер.

Управление конвейером инструкций требует надлежащего эффективного управления такими событиями, как **переходы**, **исключения** или **прерывания**, которые могут полностью нарушить поток инструкций. Например, результат условного перехода известен, только когда эта инструкция будет исполнена. Если конвейер был заполнен инструкциями, следующими за инструкцией условного перехода и переход состоялся, то все эти инструкции должны быть выброшены из конвейера.

Более того, внутри конвейера могут оказаться **взаимозависимые** инструкции. Например, если инструкция в стадии декодирования должна читать из ячейки памяти, значение которой является результатом работы инструкции, находящейся в стадии исполнения, то конвейер будет остановлен на один цикл, поскольку этот результат будет доступен только после стадии записи результатов. Поэтому компилятору необходимо **переупорядочить** инструкции в программе так, чтобы по-возможности избежать зависимостей между инструкциями внутри конвейера.

Для уменьшения времени простоя конвейера применяют ряд мер.

- **Таблица регистров (register scoreboarding)** позволяет проследить за использованием регистров. Она имеет бит для каждого регистра процессора. Если этот бит установлен, то регистр находится в состоянии ожидания записи результата. После записи результата этот бит сбрасывается, разрешая использование этого регистра. Если этот бит сброшен для всех регистров, значения которых используются в текущей инструкции, то ее можно выполнять, не дожидаясь завершения исполнения предыдущих инструкций.

- **Переименование регистров (register renaming)** является аппаратной техникой уменьшения конфликтов из-за регистровых ресурсов. Компиляторы преобразуют языки высокого уровня в ассемблерный код, назначая регистрам те или иные значения. В суперскалярном процессоре операция может потребовать регистр до того, как предыдущая инструкция закончила использование этого регистра. Это состояние *не является конфликтом данных*, поскольку этой операции не требуется значение регистра, а только сам регистр. Однако, эта ситуация приводит к остановке конвейера до освобождения регистра. Идея разрешения этой проблемы состоит в следующем: берем свободный регистр, переименовываем его для соответствия параметрам инструкции, и даем инструкции его использовать в качестве требуемого ей регистра.

Задержки внутри конвейера могут быть также вызваны временем доступа к оперативной памяти DRAM, которое намного превышает время цикла. Эта проблема в значительной степени снимается при использовании кэш памяти и буфера предвыборки инструкций (**очереди инструкций**).

Так как поток инструкций в CISC процессоре **нерегулярный** и время исполнения одной инструкции ($\mathbf{C} * \mathbf{T}$) не постоянно, то конвейеризация в этом случае имеет серьезный недостаток, делающий ее малопригодной к использованию в CISC процессорах: именно, она приводит к очень сильному усложнению процессора.

RISC процессоры используют один и тот же формат всех инструкций для того, чтобы ускорить декодирование и упростить управление конвейером, поэтому все инструкции исполняются за **один** цикл.

Одним из способов дальнейшего повышения быстродействия является конвейеризация стадий конвейера. Такие процессоры называются **суперконвейерными**. При таком подходе каждая стадия конвейера, такая как кэш (см. ниже) или АЛУ (арифметическое и логическое устройство), может принимать новую инструкцию каждый цикл, даже если эта стадия не завершила исполнение текущей инструкции. Отметим, что добавление новых уровней конвейеризации имеет смысл только в случае, если разработчик может **значительно увеличить частоту** процессора. Однако, увеличение производительности за счет увеличения внутренней частоты процессора имеет ряд недостатков. Во-первых, это увеличивает потребление энергии процессором, что делает суперконвейерные процессоры малопригодными для встраиваемых систем. Во-вторых, это вводит новые трудности в сопряжении процессора с памятью нижнего уровня, такой как DRAM. Быстродействие этой памяти растет не так быстро, как скорость процессоров, поэтому чем быстрее процессор, тем больше разрыв в производительности между ним и основной памятью.

Другим способом увеличения производительности процессоров является выполнение более чем одной операции одновременно. Такие процессоры называются **суперскалярными**. Они имеют **два или более конвейеров инструкций**, работающих параллельно, что значительно увеличивает скорость обработки потока инструкций. Одним из достоинств суперскалярной архитектуры является возможность увеличения производительности без необходимости увеличения частоты процессора. Суперскалярному процессору требуется более **широкий** доступ к памяти, так, чтобы он мог брать сразу группу из нескольких инструкций для исполнения. **Диспетчер** анализирует эти группы и заполняет каждый из конвейеров так, чтобы снизить взаимозависимость данных и конфликты регистров. Выполнение инструкций может быть не по порядку поступления, так, чтобы команды перехода были проанализированы раньше, убирая задержки в случае осуществления перехода. Компилятор должен оптимизировать код для обеспечения заполнения всех конвейеров.

Требование одного и того же ресурса несколькими инструкциями блокирует их продвижение по конвейеру и приводит к вставке циклов ожидания требуемого ресурса. Суперскалярная архитектура с тремя исполняющими устройствами будет полностью эффективной, только если поток инструкций обеспечивает одновременное использование этих трех устройств. Для обеспечения этого условия с минимальными расходами в процессоре выделяют исполняющие устройства, работающие независимо:

- **Целочисленное устройство** (IU, Integer Unit) – выполняет целочисленные операции (арифметические, логические и операции сравнения) в своем АЛУ.

- **Устройство для работы с плавающей точкой** (FPU, Floating Point Unit) – обычно отделено от целочисленного устройства, которое работает только с целыми числами и числами с фиксированной точкой. Большинство FPU совместимы со стандартом ANSI/IEEE для двоичной арифметики с плавающей точкой.

- **Устройство управления памятью** (MMU, Memory Management Unit) – вычисляет реальный физический адрес по виртуальному адресу.

- **Устройство предсказания переходов** (BU, Branch Unit) – занимается предсказанием условных переходов, для того, чтобы избежать простоя конвейера в ожидании результата вычисления условия перехода.

Поскольку условные переходы могут свести на нет все преимущества конвейерной организации процессора, остановимся более подробно на приемах, используемых BU для уменьшения их негативного влияния.

- **"Отложенные слоты"** (delay slots). Инструкцию, передающую управление от одной части программы другой, трудно исполнить за один цикл. Обычно загрузка процессорного указателя на следующую инструкцию требует один цикл, предвыборка новой инструкции требует еще один. Для избежания простоя, некоторые RISC процессоры (например, SPARC) позволяют вставить дополнительную инструкцию в так называемый "отложенный слот". Эта инструкция, которая расположена непосредственно после команды перехода, но будет выполнена до того, как будет совершен переход.

  Однако, для суперскалярных RISC процессоров отложенные слоты работают не очень хорошо. Задержка при переходе может быть два цикла, а суперскалярный процессор, который выполняет за цикл n инструкций, должен найти n инструкций для помещения в конвейеры.

- **"Спекулятивное" исполнение инструкций** (speculative execution). Некоторые RISC процессоры (например, старшие модели семейств PowerPC и SPARC) используют так называемое 'спекулятивное' исполнение инструкций: процессор загружает в конвейер и начинает исполнять инструкции, находящиеся за точкой ветвления, еще не зная, произойдет переход или нет. При этом часто выбирается наиболее вероятная ветвь программы (на основе того или иного подхода, см. ниже). Если после исполнения команды перехода оказалось, что процессор начал исполнять не ту ветвь, то все загруженные в конвейер инструкции из этой ветви и результаты их обработки сбрасываются, и загружается правильная ветвь.

- **Биты предсказания перехода в инструкции**. Некоторые RISC процессоры (например, PowerPC) используют биты предсказания перехода, которые устанавливает компилятор в инструкции перехода, и предсказывающие, будет или нет совершен переход.

- **Эвристическое предсказание переходов**. Некоторые RISC процессоры уменьшают задержки, вносимые переходами, за счет использования **встроенного предсказателя переходов**. Он предсказывает, что переходы вперед (проверки) произведены не будут, а переходы назад (циклы) - будут.

Для эффективной работы устройства предсказания перехода важно, чтобы код условия для условного перехода был вычислен как можно раньше (за несколько инструкций до самой команды перехода). Этого добиваются несколькими способами.

- **Независимость арифметических операций и кода условия**. В CISC архитектурах все арифметические операции выставляют код условия по своему результату. Это сделано для уменьшения фактора **I** – числа инструкций на задачу, поскольку есть вероятность того, что следующая инструкция будет вычислять код условия по результату предыдущей инструкции и, следовательно, может быть удалена. Однако это приводит к тому, что между командой вычисления кода условия и командой перехода очень трудно вставить полезные инструкции, так как они изменят код условия. В RISC архитектурах арифметические операции *не изменяют код условия* (если противное явно не указано в инструкции, см. ниже). Поэтому возможно между инструкцией, вычисляющей код условия, и командой перехода вставить другие инструкции (переупорядочив их). Это позволит заранее узнать, произойдет или нет переход и загрузить конвейер инструкциями.

Поскольку возможна ситуация, когда следующая инструкция будет вычислять код условия по результату предыдущей инструкции, то в RISC архитектурах часть (SPARC) или все (PowerPC) арифметические операции также имеют вторую форму, в которой будет выставляться код условия по их результату. Таким образом, часть или все арифметические операции присутствуют в двух вариантах: один не изменяет код условия (подавляющее большинство случаев использования), а другой вычисляет код условия по результату операции.

- **Использование нескольких равноправных регистров с кодом условия**. Некоторые RISC процессоры (например, PowerPC) используют несколько равноправных регистров, в которых образуется результат вычисления условия. Над этими регистрами определены логические операции, что иногда позволяет оптимизирующему компилятору заменить команды перехода при вычислении сложных логических выражений на команды логических операций с этими регистрами.

- **Использование кода условия в каждой инструкции**. Некоторые RISC процессоры (например, ARM) используют код условия в каждой инструкции. В формате каждой инструкции предусмотрено поле, где компилятором записывается код условия, при котором она будет выполнена. Если в момент исполнения инструкции код условия не такой, как в инструкции, то она игнорируется. Это позволяет вообще обойтись без команд перехода при вычислении результатов условных операций.

11.1.4. Кэш память

Время, необходимое для выборки инструкций, в основном зависит от подсистемы памяти и часто является ограничивающим фактором для RISC процессоров в силу высокой скорости исполнения инструкций. Например, если процессор может брать инструкции только из DRAM с временем доступа 60 ns, то скорость их обработки (при расчете одна инструкция за цикл) будет соответствовать тактовой частоте 16.7 MHz. Эта проблема в значительной степени снимается за счет использования **кэш памяти**.

Кэш память (cache) – это быстрое статическое ОЗУ (SRAM), вставленная между исполнительными устройствами и системным ОЗУ (RAM). Она сохраняет последние использованные инструкции и данные, так, что циклы и операции с массивами будут выполняться быстрее. Когда исполняющему устройству нужны данные и они не находятся в кэш памяти, то это **кэш-промах**: процессор должен обратиться к внешней памяти для выборки данных. Если требуемые данные находятся в кэше, то это **кэш-попадание**: доступ к внешней памяти не требуется.

Таким образом, кэши разгружают внешние шины, уменьшая потребность в них процессора. Это позволяет нескольким процессорам разделять внешние шины без уменьшения производительности каждого из них.

Кэш содержит строки из нескольких последовательных байтов (обычно 32 байта), которые загружаются процессором, используя так называемый *импульсный* (или блочный) *доступ* (burst access). Даже если CPU нужен один байт, все равно будет загружена целая строка, так как вероятно, что тем самым будут загружены следующие выполняемые инструкции или используемые данные. Блочные передачи обеспечивает высокие скорости передачи для инструкций или данных в последовательных адресах памяти. При таких передачах только адрес первой инструкции или данного будет послан в подсистему внешней памяти. Все последующие запросы инструкций или данных в последовательных адресах памяти не требуют дополнительной передачи адреса. Например, загрузка 16 байтов требует 5 циклов, если MC68040 делает блочную передачу для загрузки строки кэша, и 8 циклов, если память не поддерживает блочный режим передачи.

Кэш, в котором вместе хранятся данные и инструкции, называется **единым кэшем**. Одним из способов повышения производительности является введение в процессоре *трех* шин: *адреса*, *инструкций* и *данных*. В **Гарвардской архитектуре кэша** разделяют кэши для инструкций и данных для удвоения эффективности кэш памяти. В типичной Гарвардской

архитектуре присутствуют три вида кэш памяти: специальные кэши (например, **TLB**), внутренние кэши инструкций и данных (**первого уровня** или L1 кэш) и внешний единый кэш (**второго уровня** или L2 кэш). В процессорах, имеющих интегрированные кеши первого и второго уровней (т.е. внутри корпуса процессора), часто дополнительно устанавливают единый кэш **третьего уровня** (L3) вне процессора. Обычно L1 кэш работает на частоте процессора и имеет размер 8...32Кб, L2 кэш работает на частоте процессора или ее половине и имеет размер 128Кб... 4Мб, L3 кэш работает на частоте внешней шины и имеет размер 512Кб... 8Мб.

Кэши данных в зависимости от их поведения при записи данных в кэш разделяют на два вида.

1. Кэш с прямой записью (write-through cache). Этот вид кэш памяти при записи в нее сразу инициирует цикл записи во внешнюю память. Основным достоинством такого кэша является простота и то, что данные в кэше и в памяти всегда идентичны, что упрощает построение многопроцессорных систем.

2. Кэш с обратной записью (write-back cache) Этот вид кэш памяти при записи в нее не записывает данные сразу во внешнюю память. Запись в память осуществляется при выходе строки из кэша или по запросу системы синхронизации в многопроцессорных системах. Такая организация кэш памяти может значительно ускорить выполнение циклов, в которых обновляется одна и та же ячейка памяти (будет записано только последнее, а не все промежуточные значения как в кэше с прямой записью). Другим достоинством является уменьшение потребности процессора во внешней шине, что позволяет разделять ее нескольким процессорам. Недостатком такой организации является усложнение схемы синхронизации кэшей в многопроцессорных системах.

В силу его значительно большей эффективности, большинство современных процессоров используют кэш с обратной записью.

Организация кэш-памяти. Кэш основан на **сравнении адреса**. Для каждой строки кэша хранится адрес ее первого элемента, называемый адресом строки. Для уменьшения объема дополнительно хранимой информации (адресов строк) и ускорения поиска адреса используют несколько технических приемов.

- Пусть длина строки есть 2^b байт. Адреса строк выровнены на границу своего размера, т.е. последние b бит адреса – нулевые и потому не хранятся (т.е. размер адреса уменьшен до $32 - b$ бит).

- Фиксируется некоторое i. Строки хранятся как один или несколько (N) массивов, отсортированными по порядку i младших битов адреса (т.е. младших среди оставшихся $32 - b$). Таким образом, k-й элемент массива имеет адрес, биты которого в позициях от $32 - i - b + 1$ до $32 - b$ образуют число, равное k. Это позволяет не хранить эти биты (т.е. размер адреса уменьшен до $32 - b - i$ бит).

- Комбинация чисел b и i подбирается так, чтобы младших $b + i$ бит логического адреса совпадали бы с соответствующими битами физического адреса при страничном преобразовании (т.е. были бы смещением в странице). Это позволяет параллельно производить трансляцию адреса и поиск в кэше (т.е. параллельно работать MMU и кэшу). При типичном размере страницы 4Kb это означает $b + i = 12$.

- Определение того, содержится ли данный адрес в кэше, производится следующим образом. Берутся биты в позициях от $32 - i - b + 1$ до $32 - b$, образующие число k. Затем берутся элементы с номером k в каждом из N массивов и у полученных N строк сравниваются адреса с $32 - i - b$ битами адреса (которые уже транслированы MMU в физический адрес). Если обнаружено совпадение (т.е. имеет место кэш-попадание), то берется байт с номером b в строке.

Если рассматривается внешний кэш, то согласовывать его работу с MMU не требуется, поскольку внешний кэш работает уже с физическим адресом.

Пример: для PowerPC 603 выбрано $b = 5$ (т.е. длина строки 32 байта), $i = 7$ (т.е. длина массива 127), $N = 2$ (т.е. используются два массива).

Для каждой строки кэша с обратной записью помимо адреса хранится также признак того, что эта строка содержит корректные данные, т.е. данные в кэш памяти и в основной памяти совпадают. Этот признак используется для записи строки в память при ее выходе из кэш памяти, а также в многопроцессорных системах.

Алгоритмы замены данных в кэш памяти. Если все строки кэш памяти содержат корректные данные, то для обеспечения кэширования новых областей памяти необходимо выбрать строку, которая будет перезаписана. Эта строка выходит из кэша и, если требуется, ее содержимое будет записано обратно в память. Существуют три алгоритма замены данных в кэше:

- **вероятностный** алгоритм: в качестве номера перезаписываемой строки используется случайное число;
- **FIFO** алгоритм: первая записанная строка будет первой перезаписана;
- **LRU** (Last Recently Used) алгоритм: наименее используемая строка будет заменена новой.

Для повышения производительности процессора вводятся ряд **специальных кэшей**.

- **TLB** (Translation Look-aside Buffers) - это кэш память, используемая MMU для хранения результатов последних трансляций логического адреса в физический. Содержит пары: логический адрес и соответствующий физический адрес.
- **BTC** (Branch Target Cache) – это кэш память, используемая BU для хранения адреса предыдущего перехода и первой инструкции, выполненной после перехода. Имеет целью без задержки заполнить конвейер инструкцией, если переход уже ранее состоялся. BTC может значительно повысить производительность процессора, учитывая время, которое он бы простаивал в ожидании заполнения конвейера после перехода.

Согласование кэшей в мультипроцессорных системах. Если несколько процессоров подсоединены к одной и той же шине адреса и данных и разделяют одну и ту же внешнюю память, то должен быть реализован определенный следящий механизм (snooping) для того, чтобы все внутрипроцессорные кэши всегда содержали **одни и те же** данные.

Рассмотрим, например, систему, содержащую два процессора, каждый из которых может брать управление общей шиной. Если процессор 1, управляющий в данный момент шиной, записывает в ячейку памяти, которая кэширована процессором 2, то данные в кэше последнего становятся устаревшими. Следящий механизм позволяет второму процессору отслеживать состояние шины адреса, даже если он не является в данный момент главным (т.е. управляющим внешней шиной). Если на шине появился адрес кэшированных данных, то эти данные помечаются в кэше как некорректные. Когда второй процессор станет главным, он должен будет выбрать в случае необходимости обновленные данные из разделяемой памяти.

Если процессор 1, управляющий в данный момент шиной, читает из ячейки памяти, которая кэширована процессором 2, то возможно, что реальные данные находятся в кэше процессора 2 (еще не записаны в память, т.е. реализован кэш с обратной записью). Если это так, то следящий механизм инициирует цикл записи строки кэша процессора 2, содержащей затребованные процессором 1 данные, в разделяемую память. После этого эти данные становятся доступными процессору 1 и цикл чтения процессора 1 продолжается.

11.1.5. Многопроцессорные архитектуры

Одним из самых радикальных способов повышения производительности вычислительной системы является установка нескольких процессоров. Выделяют несколько типов построения

многопроцессорных систем в зависимости от степени связи между отдельными процессорами в системе.

1. **Сильно связанные процессоры** (или симметричные мультипроцессорные системы, symmetrical multiprocessor system, SMP). Все процессоры разделяют общую шину и общую память, могут выполнять одну и ту же задачу, причем задача может переходить от одного процессора другому. Если один процессор отказывает, он может быть заменен другим. SMP подразумевает наличие аппаратного протокола синхронизации кэшей всех процессоров (см. выше). Типичный пример: плата с двумя процессорами Pentium.

2. **Слабо связанные процессоры**. Часть системной памяти может быть разделяема, но переход задачи от одного процессора к другому невозможен. Механизмы синхронизации специфичны для каждой системы (почтовые ящики, DPRAM, прерывания). Типичный пример: стойка VME с несколькими процессорными платами и разделяемой памятью на одной из плат.

3. **Распределенные процессоры**. Несколько процессоров не разделяют ни одного общего ресурса, за исключением линии связи. Типичный пример: соединенные посредством Ethernet рабочие станции.

Архитектура SMP является самой дорогой с точки зрения аппаратной реализации и самой дешевой с точки зрения разработки программного обеспечения. И наоборот, распределенные процессоры почти не требуют аппаратных затрат, но являются самым дорогим решением с точки зрения разработки ПО. Для достижения оптимального компромисса для круга решаемых задач используют различные комбинации описанных выше технологий.

11.1.6. Поддержка многозадачности и многопроцессорности

В современных процессорах поддержка многозадачности и многопроцессорности не ограничивается аппаратной частью (вроде рассмотренного выше механизма синхронизации кэшей). Для организации доступа к критическим разделяемым ресурсам необходимо в наборе инструкций процессора предусмотреть специальные инструкции, обеспечивающие доступ к объектам синхронизации. Действительно, сами объекты синхронизации являются *разделяемыми*, а для обеспечения правильного доступа к разделяемым объектам необходимо вводить объекты синхронизации, которые в свою очередь тоже являются разделяемыми и т.д. Для выхода из этого замкнутого круга определяют некоторые "примитивные" объекты синхронизации, к которым возможен одновременный доступ нескольких задач или процессоров за счет использования *специальных инструкций доступа*.

Рассмотрим более подробно случай булевского семафора. Задача или процессор, собирающийся взять управление разделяемым ресурсом, начинают с чтения значения семафора. Если он обнулен, то задача или процессор должны ждать, пока ресурс станет доступным. Если семафор установлен в 1, то задача или процессор немедленно его обнуляют, чтобы показать, что контролируют ресурс. В процессе изменения семафора можно выделить три фазы: **чтение**, **изменение**, **запись**. Если на стадии чтения возникнет переключение задач или другой процессор станет главным на шине, то может возникнуть ошибка, так как две задачи или два процессора контролируют один и тот же ресурс. Аналогично, если переключение контекста произойдет между циклом чтения и записи, то два процесса могут взять семафор, что тоже приведет к системной ошибке. Для решения этой проблемы большинство процессоров имеют инструкцию, выполняющую неделимый цикл чтение – изменение – запись. Поскольку это одна инструкция, то переключение задач во время операции с семафором невозможно. Так как она производит неделимый цикл, то процессор, ее выполняющий, остается владельцем шины до окончания операции с семафором.

11.1.7. Влияние требований реального времени на выбор архитектуры процессора

Не все описанные выше приемы повышения производительности процессора ускоряют работу типичной системы реального времени, а иногда и замедляют ее. Этим объясняется то, что на рынке систем для ОСРВ значительную долю занимают процессоры с устаревшей CISC архитектурой, имеющие примитивный конвейер и кэш. Рассмотрим основные причины этого положения.

- Доводы (статистический анализ), положенные в основу обоснования RISC архитектуры, оказались не совсем верными для ОСРВ. Часто приложение ОСРВ проводит значительную часть времени в обработке прерываний от внешних устройств. Обработчики прерываний часто пишутся на ассемблере и наличие сложных инструкций в CISC процессоре позволяет уменьшить длину обработчика. Наличие сложных режимов адресации памяти позволяет обойтись меньшим количеством регистров и, соответственно, еще уменьшить обработчик (за счет уменьшения количества сохраняемых/восстанавливаемых регистров).

- Увеличение числа регистров в RISC процессорах приводит к увеличению времени на переключение задач и, в частности, на обработку прерываний, поскольку необходимо сохранять/восстанавливать значительное количество регистров.

- Увеличение глубины конвейера приводит к увеличению времени на переключение задач и, в частности, на обработку прерываний, поскольку необходимо дождаться выполнения всех инструкций в конвейере.

- Увеличение глубины конвейера и размеров кэша не всегда приводит к ускорению работы ОСРВ из-за большого количества прерываний (т.е. переключений задач), поскольку конвейеры и кэш не успевают заполниться необходимыми инструкциями и данными.

- Увеличение размеров кэша приводит к увеличению накладных расходов на их синхронизацию в многопроцессорных системах.

- В ряде ситуаций ОСРВ приходится блокировать все конвейеры суперскалярного процессора, кроме одного. Дело в том, что инструкции в таком процессоре могут завершаться не в том порядке, в котором они подавались в процессор. Например, поток инструкций $I_4 \to I_3 \to I_2 \to I_1 \to$ процессор может быть распределен между двумя конвейерами как $I_3 \to I_1 \to$ конвейер$_1$, $I_4 \to I_2 \to$ конвейер$_2$. Если инструкция I_1 выполняется очень долго (например, в ней делается обращение к памяти), то инструкции I_2, I_4 будут исполнены раньше, чем I_1, I_3. Эта ситуация недопустима при работе с вводом/выводом. Поэтому в суперскалярных процессорах существуют инструкции, переводящие его в *последовательный* режим исполнения инструкций для обеспечения их правильного порядка. Часто эти инструкции фактически блокируют все конвейеры, кроме одного.

Для ускорения обработки прерываний, в некоторых процессорах применяются специальные меры.

- Таблица прерываний может храниться во внутренней памяти процессора, что делает ненужной выборку из внешней памяти (Intel 80960).

- Процессор может включать теневые регистры, что делает ненужным сохранение контекста текущей задачи в простых процедурах обработки прерываний (HP-PA).

- Критические процедуры обработки прерываний могут быть заблокированы в кэше инструкций (Motorola 68060).

- Таблица прерываний может хранить первые инструкции обработчика прерываний, что уменьшает простой конвейера (SPARC).

11.2. Организация данных во внешней памяти

В системах реального времени достаточно частой является ситуация, когда в одном VME крейте находятся несколько плат, построенных на процессорах разных производителей. Эти платы обычно решают общую задачу по управлению промышленным оборудованием и обмениваются между собой данными через разделяемую память (собственно, это единственный способ обмена, предоставляемый шиной VME, см. раздел 12.1). В этой ситуации необходимо согласование представления данных в памяти каждым из участвующих в обмене процессоров.

11.2.1. Организация целочисленных данных

При доступе к целочисленным данным основным вопросом является способ нумерации байтов в слове.

Если бы в 32-битной архитектуре минимальным адресуемым элементом оперативной памяти являлось бы 32-битное слово, то вопрос о нумерации байтов в слове не вставал бы. В реальной ситуации, когда минимальным адресуемым элементом оперативной памяти является байт (8-ми битное слово), данные большего размера образуются как объединение подряд идущих байт. Выбор нумерации байт в 32 битном (4 байта) слове может быть произвольным, что дает $24 = 4!$ способа. На практике используются только два: ABCD (называемый big-endian) и DCBA (называемый little-endian).

В **big-endian** модели байты в слове нумеруются от наиболее значимого к наименее значимому. В **little-endian** модели байты в слове нумеруются от наименее значимого к наиболее значимому. Примеры big-endian процессоров: Motorola 68xxx, PowerPC (по умолчанию), SPARC, пример little-endian процессора: Intel 80x86. Процессоры PowerPC, Intel 80960x, ARM, SPARC (64-битные модели) могут работать как в big-endian режиме, так и в little-endian.

Если процессоры, осуществляющие обмен через разделяемую память, используют разный режим нумерации байтов, то потребуется преобразовывать все полученные или переданные данные. Практически все современные процессоры имеют для этой цели специальную инструкцию.

11.2.2. Организация данных с плавающей точкой

Все современные процессоры поддерживают стандарт ANSI/IEEE 754-1985 в организации данных с плавающей точкой. Необходимо только учитывать, что данные с плавающей точкой содержат несколько байт, поэтому на них также оказывает влияние способ нумерации байтов в слове.

floating-point single:

- размер – 4 байта,
- знак s – бит 31 (1 бит)
- показатель p – биты $23 \ldots 30$ (8 бит)
- мантисса x – биты $0 \ldots 22$ (23 бит)

Нормализованное значение (т.е. при $0 < p < 255$)

$$(-1)^s \cdot 2^{p-127} \cdot 1.x$$

floating-point double:

- размер – 8 байт,
- знак s – бит 63 (1 бит)
- показатель p – биты $52 \ldots 62$ (11 бит)
- мантисса x – биты $0 \ldots 51$ (52 бит)

Нормализованное значение (т.е. при $0 < p < 2047$)

$$(-1)^s \cdot 2^{p-1023} \cdot 1.x$$

floating-point quad:
(поддерживается не всеми процессорами)

- размер – 16 байт,
- знак s – бит 127 (1 бит)
- показатель p – биты 112...126 (15 бит)
- мантисса x – биты 0...111 (112 бит)

Нормализованное значение (т.е. при $0 < p < 32767$)

$$(-1)^s \cdot 2^{p-16382} \cdot 1.x$$

11.2.3. Пути повышения производительности оперативной памяти

Для повышения производительности оперативной памяти памяти применяют несколько приемов.

- **Увеличение ширины шины данных**: переход от SIMM (32 бит шина) к DIMM (64 или 128 бит шина) модулям при построении подсистемы памяти.
- **Введение небольшой статической памяти SRAM для буферизации DRAM модулей**: буферизованные DIMM модули.
- **Введение конвейера в модули DRAM**. Используется та же идея, что и при введении конвейера в процессоры. Внутри модуля памяти находится несколько обрабатываемых обращений к памяти в разной степени готовности (формирование адреса, выбор банка, выборка данных, запись их на внешнюю шину и т.д.). Это позволяет модулю памяти принимать/выдавать данные каждый цикл шины (при условии оптимального функционирования конвейера). В силу этого такая память получила название synchronous DRAM (SDRAM). Для такой памяти в качестве времени доступа производители в рекламных целях пишут минимальный цикл шины, что дает фантастические времена доступа менее 10ns (т.е. частота шины более 100MHz). На самом же деле в модулях SDRAM использованы обычные микросхемы памяти, время доступа к которым – около 60ns.
- **"Расслоение" оперативной памяти**. Поскольку процессор обменивается с памятью только блоками размером со строку кэша, то можно разделить этот блок на N частей (N обычно 2, 4, 8) и передать каждую из частей своей подсистеме памяти. В результате получаются N подсистем памяти, работающих параллельно. Если программа требует последовательные адреса памяти (т.е. стратегия предвыборки строки кэша себя оправдывает), то этот подход может в N раз увеличить производительность подсистемы памяти.

11.3. Процессоры Motorola 68xxx

Процессоры Motorola 68xxx длительное время доминировали на рынке промышленных компьютерных систем. Сейчас они оттеснены на второе место процессорами PowerPC.

11.3.1. Общий обзор

Семейство процессоров Motorola 68xxx (сокращенно M68k) характеризуется простотой реализации как аппаратных, так и программных решений на его базе. Первым процессором семейства являлся хорошо известный M68000. Хотя он появился в 1979 году, он является процессором с полной *32-битной* внутренней архитектурой. Подобно всем последующим процессорам, его линейная организация памяти и единое с памятью пространство ввода/вывода облегчает аппаратные и программные разработки. Наконец, специальные выводы процессора

устанавливаются на каждом цикле шины для того, чтобы различить супервизорское и пользовательское адресные пространства. Если эти выводы декодировать при вычислении адреса, то системные ресурсы будут изолированы от несанкционированного доступа пользовательских программ.

Следующие поколения процессоров M68k, включая M68060, с программной точки зрения вносили только новые режимы адресации памяти и некоторые дополнительные инструкции, и поэтому программируются так же легко, как M68000. Эволюция затронула аппаратную архитектуру процессоров: в их составе появились и развивались конвейеры, кэши, MMU, FPU и т.д.

Основываясь на архитектуре M68k Motorola разработала семейство контроллеров MC683xx, которые разделяют на 3 группы: группу 68000, CPU32 и CPU32+. В этих контроллерах вычислительная мощность M68k сочетается с интегрированными периферийными процессорами, образуя высокопроизводительные контроллеры. Наиболее специализированные, ориентированные на ввод/вывод контроллеры (68302 и 68360) включают непрограммируемый RISC процессор, управляющий последовательным коммуникационным каналом. Это дает возможность одному устройству управлять таким сложным последовательным каналом, как Ethernet или ISDN.

Все члены семейства MC683xx имеют межмодульную шину (intermodule bus, IMB). IMB обеспечивает общий интерфейс для всех модулей семейства MC683xx, что позволяет фирме Motorola быстро разрабатывать новые устройства, используя библиотеки существующих модулей.

11.3.2. Основные члены семейства

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

M68000 (1979, 68000 транзисторов, макс. частота: 16.67MHz, 1 MIPS):

- асинхронные передачи данных (кроме области интерфейса 6800)
- *16-битная* шина данных, отсутствует динамическое изменение ширины шины
- *24-битная* шина адреса
- 14 режимов адресации памяти (включая косвенные регистровые)
- 16 32-битных регистров общего назначения
- два уровня привилегий: пользовательский и супервизорский
- два указателя стека для разделения пользовательского и супервизорского стеков
- фиксированная в памяти таблица прерываний (начиная с адреса 0)
- одна неделимая инструкция TAS (Test And Set) для установки семафоров

M68010 (1983, 68000 транзисторов, макс. частота: 16.67MHz, 1 MIPS):

- перемещаемая таблица прерываний
- поддержка механизма виртуальной памяти

M68020 (1984, 195000 транзисторов, макс. частота: 25MHz, 5 MIPS):

- асинхронные передачи данных
- *32-битная* шина данных, динамическое изменение ширины шины (адаптируется к 8-ми, 16-ти и 32-х битным обменам с внешними устройствами)
- *32-битная* шина адреса
- 18 режимов адресации памяти (включая косвенные через память)
- *256-байт* кэш инструкций

- *3-х стадийный* конвейер, что позволяет одновременно обрабатывать до трех слов одной операции или три последовательные инструкции
- интерфейс сопроцессора, что позволяет подключить внешнее FPU (MC68881 или MC68882) и MMU (MC68851)
- дополнительный указатель стека для разделения аппаратных и программных прерываний
- две дополнительных неделимых инструкции: CAS и CAS2 (Compare And Swap 32 или 64 бита) для установки семафоров
- новые инструкции для работы с битовыми полями

M68030 (1986, 300000 транзисторов, макс. частота: 50MHz, 8 MIPS):

- Гарвардская архитектура
- два различных кэша: *256-байт* кэш инструкций + *256-байт* кэш данных
- асинхронные передачи данных
- синхронный интерфейс, что позволяет осуществлять блочные (burst) передачи в/из кэша
- MMU, позволяющее работать со страницами размером от 256 байт до 256 килобайт

M68040 (1989, 1200000 транзисторов, макс. частота шины: 40MHz, 8 MIPS, 3.5Mflops):

- синхронные передачи данных
- отсутствует динамическое изменение ширины шины
- частота процессора равна удвоенной частоте шины (максимально 80MHz)
- *6-ти стадийный* конвейер
- FPU
- *4-килобайт* кэш инструкций + *4-килобайт* кэш данных
- механизм синхронизации шины (bus snooping) (арбитраж шины должен быть внешним, процессор имеет вывод запроса шины), это обеспечивает согласование кэшей в многопроцессорных системах

M68060 (1994, 2500000 транзисторов, макс. частота шины: 66MHz, 100 MIPS):

- суперконвейерный, суперскалярный 32 битный гибридный CISC-RISC процессор
- набор инструкций M68040 реализован аппаратной логикой, а не микрокодом
- основные устройства: буфер инструкций, 4-х стадийное конвейерное устройство предвыборки, два 4-х стадийных конвейерных целочисленных устройства, устройство переходов, FPU повышенной точности
- кэш переходов (BTC)
- поток инструкций разделяется на два конвейера на FIFO стадии
- устройство предвыборки преобразует входной поток M680x0 инструкций, имеющих переменную длину, в поток RISC инструкций фиксированной длины
- M68060 может может исполнять за один цикл 4 инструкции M680x0: две целочисленных инструкции, одну инструкцию перехода и одну инструкцию с плавающей точкой; этот параллелизм обеспечивает высокую скорость исполнения даже для кода, не перекомпилированного специально для M68060
- *4-килобайт* кэш инструкций + *4-килобайт* кэш данных
- автоматическое уменьшение потребляемой мощности: внутренние функциональные блоки автоматически выключаются, если они не используются в течении ряда циклов

M68302 (микроконтроллер группы 68000):

- IMP (Integrated Multiprotocol Processor)
- M68302 состоит из процессора M68000, System Integration Block (SIB) и Communication Processor (CP)
- SIB содержит контроллер DMA, два 16-битных таймера общего назначения, контроллер памяти и контроллер прерываний
- CP является выделенным RISC процессором, обслуживающим 6 последовательных портов, и отвечает за работу с последовательными каналами по выбранному пользователем протоколу (ISDN, UART, HDLC, BSC и другие)

M68360 (или QUICC) (микроконтроллер группы CPU32+):

- M68360 состоит из процессора CPU32+, SIM60 (System Integration Module) и CPM (Communication Processor Module)
- процессор CPU32+ является процессором M68020 без кэша и сопроцессорного интерфейса; система команд процессора M68020 расширена CPU32+ специфическими инструкциями табличной интерполяции
- SIM60 интегрирует основные устройства общего назначения, которые могут быть полезны в любых 32-битных процессорных системах: синтезатор частоты, таймеры-будильники, таймер периодического прерывания, контроллер памяти, способный без дополнительной логики управлять 32-битными DRAM
- CPM содержит 2 DMA контроллера, 4 таймера общего назначения, контроллер прерываний, 7 коммуникационных контроллеров, управляющих 7-ю последовательными физическими каналами
- поддерживаются протоколы UART, ISDN, HDLC, BSC, AppleTalk
- протокол Ethernet поддерживается в версии MC68EN360

11.3.3. Программная модель

Прикладной программе доступны 16 регистров общего назначения и несколько служебных регистров.

Регистры d0 – d7 (регистры данных):

Могут содержать целочисленные данные следующих типов

1. Бит (M68020 и выше, только инструкции, работающие с битовыми полями)
2. Двоично-закодированные десятичные числа (BCD); байт содержит одну цифру, существуют инструкции, работающие с двумя цифрами в одном байте
3. Байт (8 бит); при записи в регистр старшая часть не используется и не изменяется
4. Слово (16 бит); при записи в регистр старшая часть не используется и не изменяется
5. Длинное слово (32 бит)
6. Счетверенное слово (64 бит); используются любые два регистра, над такими операндами определена только инструкция пересылки (MOVEM).

Регистры a0 – a7 (регистры данных):

Могут содержать целочисленные данные следующих типов

1. Слово (16 бит); при записи в регистр старшая часть заполняется знаковым битом источника
2. Длинное слово (32 бит)

Регистр pc (program counter):
: содержит адрес следующей инструкции. При записи в этот регистр происходит переход по адресу, который был записан.

Регистр кода условия ccr (condition code register):
: является частью регистра статуса (status register, SR), который не доступен пользовательской программе как регистр. Коды условия устанавливаются по результату арифметических операций или специальными инструкциями, и используются в командах условного перехода.

Поддерживаются следующие режимы адресации памяти (в терминах языка C).

- $*(\mathsf{A}_n + +)$
- $*(- - \mathsf{A}_n)$
- $*(\mathsf{B}_n + \mathsf{d} + \mathsf{X}_n)$
- $*(*(\mathsf{B}_n + \mathsf{d}) + \mathsf{X}_n + \mathsf{o})$
- $*(*(\mathsf{B}_n + \mathsf{d} + \mathsf{X}_n) + \mathsf{o})$

где

- A_n – адресный регистр a0 – a7
- B_n – адресный регистр a0 – a7 или pc
- X_n – отсутствует или регистр a0 – a7 или d0 – d7; у M68020 и выше может быть $\mathsf{X}_n * \mathsf{s}$, $\mathsf{s} = 1, 2, 4, 8$
- d – константа со знаком (8, 16, 32 бит), включая 0
- o – константа со знаком (16, 32 бит), включая 0

Инструкции процессоров Motorola 68xxx имеют от нуля до трех операндов. Большинство инструкций (пересылки, арифметические, логические) имеют два операнда, один из которых не изменяется в операции (источник), а другой является результатом операции (приемник). Обычно используется синтаксис, в котором источник является левым операндом, а приемник - правым.

Существует несколько инструкций, неявно использующих указатель стека a7. Это стековые операции и вызовы функций.

11.4. Процессоры Intel 80x86

Процессоры Intel 80x86 доминируют на рынке персональных компьютеров. На рынке промышленных систем их роль незначительна.

11.4.1. Общий обзор

Первый представитель семейства - i8086, появился в 1979г. Это был 16-ти битный процессор, обеспечивающий совместимость с предыдущими 8-ми битными процессорами i8080 и i8085. Это наложило определенный отпечаток на программную архитектуру i8086:

- устаревший набор инструкций,
- множество нелогичных ограничений на операнды.

Основные общие черты семейства i80x86

- пространство ввода/вывода отделено от пространства памяти,

- сегментная организация памяти,
- малое число регистров,
- невзаимозаменяемость регистров (много инструкций, неявно использующих жестко закрепленные регистры).

Процессор i80386 был первым 32-х битным процессором в семействе и уже мог работать с плоской моделью памяти. Однако, этот процессор сохранил программную совместимость с предыдущими членами семейства. Это еще больше увеличило его сложность и законсервировало устаревшую программную архитектуру i8086.

11.4.2. Основные члены семейства

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

i80386 (1987, 275000 транзисторов, 33MHz, 8 MIPS):

- *32-бит* шина данных и *32-бит* шина адреса
- 8 32-битных регистров общего назначения
- динамическое изменение ширины шины (16 или 32 бит)
- пространство памяти отделено от 64Кб пространства ввода/вывода
- буфер предвыборки 16 байт
- MMU (страничное или(и) сегментное) с кэшем результатов трансляции; может работать в 3-х режимах:
  - *real mode* : 24 бит сегментный адрес, нет трансляции
  - *protected mode* : 32 бит адрес, страничное или(и) сегментное преобразование
  - *virtual 8086 mode* : 24 бит сегментный адрес, страничная трансляция
- внешнее FPU i80387
- внешний контроллер (внешней) кэш памяти i82385
- поддержка многозадачности посредством регистра задачи и дескрипторов задач

i80486 (1989, 1200000 транзисторов, частота шины 33MHz, 20 MIPS):

- внутренняя частота процессора равна либо частоте шины (DX 33MHz), либо удвоенной частоте шины (DX2 66MHz), либо утроенной частоте шины (DX4 100MHz)
- единый 8Кб (16Кб) кэш инструкций и данных (с поддержкой блоковых передач (burst access))
- кэш сквозной записи (write through)/ обратной записи (write back)
- механизм синхронизации кэшей в многопроцессорных системах (bus snooping)
- 32 байт буфер предвыборки
- встроенное FPU (без конвейера)
- 5-ти стадийный конвейер, использующий таблицу регистров (register scoreboard)

Pentium (1993, 3100000 транзисторов, частота шины 50/60/66MHz, 100 specint92, 80 specfp92):

- внутренняя частота процессора равна либо частоте шины (60/66MHz), либо 1.5 частоты шины (75/90/100MHz), удвоенной частоте шины (120/133MHz), 2.5 частоты шины (150/166MHz), либо утроенной частоте шины (180/200MHz)
- 64-бит внешняя шина данных, но 32-битная внутренняя архитектура
- суперскалярная архитектура: 2 IU

- аппаратное декодирование
- устройство предсказания переходов (BU)
- 5-ти стадийный конвейер в IU, 8-ми стадийный конвейер в FPU
- Гарвардская архитектура: 8Кб кэш инструкций и 8Кб кэш данных
- механизм синхронизации кэшей MESI (Modified, Exclusive, Shared, Invalid), используемый в PowerPC
- гибридная CISC/RISC архитектура

Pentium Pro (1996, 5500000 транзисторов, 150+MHz, 366 specint92, 283 specfp92):

- суперскалярный процессор: до 5 инструкций за цикл
- 14-ти стадийные конвейеры (2 для IU и 1 для FPU)
- Гарвардская архитектура: 16Кб кэш инструкций и 8Кб кэш данных
- архитектура DIB (Dual Independent Bus): одна шина связывает процессор с кэш памятью второго уровня, а вторая - с ОЗУ
- 256(512)Кб встроенный кэш второго уровня; расположен на отдельном кристалле, но в том же корпусе, связан с процессором 64-битной шиной, работающей на частоте процессора
- исполнение инструкций не по порядку
- спекулятивное исполнение: выполнение инструкций за точкой ветвления
- переименование регистров (register renaming)

Pentium-II (1997, 7500000 транзисторов, 233+MHz):

- Pentium Pro ядро с архитектурой DIB
- суперскалярный процессор: 2IU и 2FPU
- Гарвардская архитектура: 16Кб кэш инструкций и 16Кб кэш данных
- 512Кб встроенный кэш второго уровня; расположен на отдельном кристалле, связан с процессором 64-битной шиной, работающей на половине частоты процессора (для модификации Pentium-II Xeon – на частоте процессора)

Pentium-III (1999, 450+MHz):

- Pentium II с дополнительными мультимедийными инструкциями
- 512Кб встроенный кэш второго уровня; расположен на отдельном кристалле, связан с процессором 64-битной шиной, работающей на половине частоты процессора (для модификации Pentium-III Xeon – на частоте процессора)

P7 (Intel и HP, 2000):

- 64-бит архитектура
- RISC процессор с набором команд PA-RISC, способен выполнять инструкции i8086

Pentium-4 (2000, 1400+MHz):

- Pentium III с увеличенной частотой шины между L2 кэшем и внешней памятью.

11.4.3. Программная модель

Прикладной программе доступны 8 регистров общего назначения и несколько служебных регистров.

Регистры eax, ebx, ecx, edx:

Могут содержать целочисленные данные следующих типов

1. Бит (только инструкции, работающие с битовыми полями)

2. Двоично-закодированные десятичные числа (BCD); байт содержит одну цифру, существуют инструкции, работающие с двумя цифрами в одном байте
3. Байт (8 бит); при записи в регистр старшая часть не используется и не изменяется; возможен доступ к битам 0...7 и битам 8...15 соответственно по именам al, bl, cl, dl и ah, bh, ch, dh
4. Слово (16 бит); при записи в регистр старшая часть не используется и не изменяется; обращение осуществляется по имени ax, bx, cx, dx соответственно
5. Длинное слово (32 бит)

Регистры esi, edi, ebp, esp:
: Могут содержать целочисленные данные следующих типов

1. Бит (только инструкции, работающие с битовыми полями)
2. Двоично-закодированные десятичные числа (BCD); байт содержит одну цифру, существуют инструкции, работающие с двумя цифрами в одном байте
3. Слово (16 бит); при записи в регистр старшая часть не используется и не изменяется; обращение осуществляется по имени si, di, bp, sp соответственно
4. Длинное слово (32 бит)

Регистр кода условия ccr (condition code register):
: является частью регистра флагов (EFLAGS), который не доступен пользовательской программе как регистр. Коды условия устанавливаются по результату арифметических операций или специальными инструкциями и используются в командах условного перехода.

Следующие регистры доступны пользовательской программе, но не используются в прикладных программах для UNIX систем (поскольку они работают в плоской модели памяти):

Сегментные регистры gs, fs, es, ds, cs, ss:
: содержат 16-бит селекторы сегментов, неявно участвуют при формировании адреса

Все режимы адресации памяти процессоров Intel 80x86 можно записать одной формулой: адрес ячейки памяти есть

$$\mathsf{base} + \mathsf{index} * \mathsf{scale} + \mathsf{displacement}$$

где

- base – базовый регистр: eax, ebx, ecx, edx, esi, edi, ebp, esp
- index – индексный регистр: eax, ebx, ecx, edx, esi, edi, ebp
- scale – целая константа 1, 2, 4, 8
- displacement – смещение 8 или 32 бит

Любой из элементов адреса может отсутствовать (с одним исключением: если отсутствует index, то должен отсутствовать и scale).

Инструкции процессоров Intel 80x86 имеют от нуля до трех операндов (явных или неявных). Большинство инструкций (пересылки, арифметические, логические) имеют два операнда, один из которых не изменяется в операции (источник), а другой является результатом операции (приемник). Фирма Intel использует синтаксис, в котором приемник является левым операндом, а источник - правым. Но большинство ассемблеров в UNIX системах использует синтаксис a la Motorola 68xxx: левый операнд - источник, правый - приемник.

У большинства двухоперандных инструкций один из операндов (источник или приемник) может быть регистром или ячейкой памяти, другой тогда может быть регистром или непосредственным значением. Это позволяет разделить двухоперандные инструкции на следующие группы:

- регистр - регистр
- регистр - память
- память - регистр
- непосредственное значение - регистр
- непосредственное значение - память

Существуют инструкции, (неявно) осуществляющие операции типа "память - память". Это строковые инструкции и операции со стеком.

Инструкции, неявно использующие жестко закрепленные регистры:

- умножение и деление с двойной точностью
- ввод/вывод
- работа со строками
- циклы
- сдвиги
- операции со стеком (включая вызовы функций)
- инструкция трансляции

В 32-битном режиме работы процессора большинство инструкций имеют операнды размером 8 или 32 бит; использовать операнды размером 16 бит можно при использовании специального префикса инструкции. Однако, изменить размер смещения в адресе (8 или 32 бит) таким способом невозможно.

11.5. Процессоры PowerPC

Архитектура PowerPC (Performance Optimized With Enhanced Risc Personal Computer) была разработана совместно IBM, Motorola и Apple. В настоящее время она доминирует на рынке промышленных систем.

11.5.1. Общий обзор

Архитектура PowerPC определяет три архитектурных уровня:

- *UISA* (User Instruction Set Architecture) определяет уровень архитектуры, которому должно удовлетворять пользовательское программное обеспечение. UISA задает программную модель и модель памяти для пользовательских программ. Именно, UISA определяет доступный прикладной программе набор инструкций, набор регистров, типы данных, соглашения о хранении чисел с плавающей точкой в памяти, модель исключений, видимую прикладной программой.
- *VEA* (Virtual Environment Architecture) определяет дополнительный уровень архитектуры, которому должно удовлетворять пользовательское программное обеспечение, выходящее за рамки обычных требований прикладных программ. VEA задает модель памяти для окружений, в которых множество устройств могут получать доступ к памяти, определяет модель кэша и инструкции управления кэшем.
- *OEA* (Operating Environment Architecture) определяет уровень архитектуры, которому должно удовлетворять супервизорское программное обеспечение (операционные системы). OEA определяет модель управления памятью, регистры уровня суперпользователя, требования к синхронизации процессов, модель исключений.

Эти спецификации позволяют разрабатывать новые процессоры семейства, сохраняя программную совместимость с существующими и будущими процессорами PowerPC. Впервые в истории разработки процессоров важнейшие усилия по спецификации были предприняты до появления первого процессора.

В дополнение к этим архитектурным определениям три производителя (IBM, Motorola, Apple) разработали *эталонную платформу* для разработки плат на основе PowerPC. *CHRP* (Common Hardware Reference Platform) является открытой спецификацией для разработки компьютерных систем на основе PowerPC. Отметим два важнейших аспекта этой спецификации. Во-первых, спецификация описывает устройства, интерфейсы и форматы данных, требуемые для разработки и построения законченной компьютерной системы. Она описывает методы абстрагирования аппаратных деталей от операционной системы. Более того, CHRP предписывает использовать вторую PCI шину, так, что шина PowerPC разделяется только L2 кэшем, контроллером памяти и мостом PCI. Это позволяет разработчикам использовать новые процессоры PowerPC без изменений в остальной части платы. Во-вторых, спецификация описывает эталонную реализацию операционной системы, согласованную с известными операционными системами для PowerPC: AIX (IBM), Solaris (Sun), Windows NT (Microsoft).

11.5.2. Основные члены семейства

Все перечисленные ниже процессоры удовлетворяют стандарту на PowerPC и отличаются набором аппаратных устройств.

PowerPC 603 (1993, 1600000 транзисторов):

- 8Кб кэш инструкций + 8Кб кэш данных (32-бит Гарвардская архитектура)
- 5 независимых исполняющих устройств: 1 IU + 1 FPU + 1 BU + 1 load/store unit + 1 system register unit
- FPU конвейеризировано, так, что инструкции сложения и умножения одинарной точности могут обрабатываться каждый цикл
- 4-х стадийный конвейер
- MMU, выполняющее сегментную и страничную трансляцию адреса из 52-битного логического адреса в 32-битный физический адрес
- два 64-входовых TLB
- потребление 3Вт на частоте 80MHz, четыре программно контролируемых режима экономии энергии

PowerPC 603e (1993, 1600000 транзисторов):
Вариант PowerPC 603 для настольных систем, частота до 240MHz

PowerPC 604 (1994, 3600000 транзисторов):

- суперскалярный процессор (4 инструкции за цикл)
- 16Кб кэш инструкций + 16Кб кэш данных (32-бит Гарвардская архитектура)
- 6 независимых исполняющих устройств: 3 IU + 1 FPU + 1 BU + 1 load/store unit
- 6-ти стадийный конвейер
- два 128-входовых TLB
- возможность последовательного выполнения инструкций
- *переименование регистров* (register renaming)
- *динамическое* предсказание переходов
- *поддержка многопроцессорности* посредством специального протокола синхронизации кэшей

- потребление 10Вт на частоте 100MHz, один программно контролируемый режим экономии энергии

PowerPC 604e (1994, 3600000 транзисторов):
Вариант PowerPC 604 для настольных систем, частота до 300MHz

PowerPC 620 (1995, 7000000 транзисторов, на частоте 133MHz 225 specint 92, 300 specfp 92):
- *128*-бит шина данных, *40*-бит шина адреса, *64*-бит регистры
- суперскалярный процессор (*6* инструкций за цикл)
- *32Кб* кэш инструкций + *32Кб* кэш данных (*64*-бит Гарвардская архитектура)
- *6* независимых исполняющих устройств: 2 IU + 1 FPU + 1 BU + 1 load/store unit + 1 complex unit
- 5-ти стадийный конвейер
- 2-х уровневое MMU, первичное MMU имеет 64-входовый TLB, вторичное MMU имеет 128-входовый TLB
- для ускорения переходов count register (ctr) имеет теневой регистр, в который он переименовывается во время перехода
- интегрированный контроллер L2 кэша
- потребление 30Вт на частоте 100MHz, один программно контролируемый режим экономии энергии

PowerPC G3 (1997, не более 30000000 транзисторов, частота 300MHz):
развитие PowerPC 620, интегрированный L2 кэш

PowerPC G4 (1998, не более 50000000 транзисторов, частота 400MHz):
развитие PowerPC G3

IBM PowerPC 403 (1994):
- PowerPC микроконтроллер
- 2Кб кэш инструкций + 1Кб кэш данных (32-бит Гарвардская архитектура)
- периферийные интерфейсные устройства: интерфейс шины, DMA контроллер, контроллер прерываний (на 6 запросов), последовательный порт
- до восьми интерфейсов банков памяти и устройств ввода/вывода
- 4 таймера
- низкое потребление энергии

Motorola MPC500 (1994, 40 MIPS, 25MHz):
- PowerPC микроконтроллер
- 4Кб кэш инструкций
- 4Кб SRAM
- 4 независимых исполняющих устройств: 1 IU + 1 FPU + 1 BU + 1 load/store unit
- контроллер прерываний на 32 запроса (время задержки 1mks на частоте 25MHz)
- контроллер памяти, способный управлять 12 микросхемами памяти
- встроенный автотест
- очень низкое потребление энергии (530mW)

Motorola PowerQUICC (1996, 52 MIPS, 40MHz):
- улучшенная версия QUICC (M68360)

- ядро CPU32+ заменено на ядро MPC500
- 4 высокоскоростных последовательных коммуникационных канала контролируются выделенным RISC коммуникационным процессором, работающим независимо от основного процессора
- PowerQUICC по сравнению с QUICC имеет контроллер PCMCIA 2.01 и аналог процессора DSP

11.5.3. Программная модель

Прикладной программе доступны следующие регистры.

32 целочисленных регистра общего назначения (РОН) r0 – r31:
: содержат слово (32/64 бит)

32 регистра с плавающей точкой f0 – f31:
: содержат значение с плавающей точкой (64 бит)

cr (condition register):
: это 32-битный регистр, разделенный на восемь 4-х битных полей cr0–cr7. Поля регистра cr могут быть установлены одним из следующих способов.

  - Указанное поле регистра cr может быть установлено с помощью инструкции пересылки mtcrf в cr из РОН.
  - Указанное поле регистра cr может быть установлено с помощью инструкции пересылки mcrf в cr из другого поля cr.
  - Указанное поле регистра xer может быть скопировано в регистр cr с помощью инструкции mcrxr.
  - Указанное поле регистра fpscr может быть скопировано в регистр cr с помощью инструкции mcrfs.
  - Поля регистра cr могут быть изменены с помощью логических операций, определенных над полями cr.
  - cr0 может быть неявным результатом целочисленной инструкции. Все целочисленные инструкции имеют бит Rc; если его установить, то биты в cr0 будут установлены сравнением результата инструкции с нулем.
  - cr1 может быть неявным результатом инструкции с плавающей точкой и указывать на статус исключения с плавающей точкой. Все инструкции с плавающей точкой имеют бит Rc; если его установить, то биты в cr1 будут установлены копированием соответствующих битов из fpscr.
  - Указанное поле регистра cr может быть результатом целочисленной или вещественной инструкции сравнения.

fpscr (Floating Point Status and Control Register):
: это 32-битный регистр, содержащий все биты сигналов исключений для операций с плавающей точкой, биты суммарных исключений, биты разрешения исключений, биты управления округлением, необходимые для удовлетворения стандарту IEEE 754.

xer register:
: это 32-битный регистр, содержащий флаги переполнения и переносов для целочисленных операций. Также содержит число байтов, которые нужно передать в инструкциях Load String Word Indexed (lswx) или Store String Word Indexed (stswx)

lr (link register):
: это 32/64-битный регистр, содержащий адрес перехода для инструкций Branch Conditional to Link Register (bclr) и Branch and Link (bl). Содержит адрес вызвавшей функции сразу после вызова.

ctr (count register):

это 32/64-битный регистр, содержащий счетчик цикла, который может быть декрементирован в течение выполнения надлежащим образом закодированных инструкций перехода. Регистр ctr может также содержать адрес перехода для инструкции Branch Conditional to Count Register (bcctr).

Поддерживаются следующие два режима адресации памяти.

- адрес = (rA|0)+offset (включая offset=0)
- адрес = (rA|0)+rB

где обозначено

- (rA|0) - РОН r1...r31, если rA не равно r0, иначе 0
- offset - 16-бит смещение (знаково расширяемое)
- rB - РОН r0...r31

Все инструкции (за исключением load/store) имеют операнды в регистрах и потому размер всех операндов равен размеру слова (32/64 бит). Подавляющее большинство инструкций – трехоперандные, хотя существует инструкция с 6-ю операндами.

В большинстве арифметических инструкций можно установить код условия cr0 по результату.

Отметим, что несмотря на принадлежность к классу RISC процессоров, PowerPC имеет даже больше инструкций, чем классический CISC процессор Motorola 68xxx.

11.6. Процессоры SPARC

В 1987 году Sun Microsystems аннонсировала SUN-4 - первую компьютерную систему, основанную на новой RISC CPU архитектуре *SPARC* (Scalable Processor ARChitecture). В отличие от многих других процессоров, SPARC позиционировался как *открытая архитектура*, которую могут использовать все.

11.6.1. Общий обзор

SPARC процессоры используют Берклевскую архитектуру, основанную на регистровых окнах. Внутренние регистры образуют блоки с частичным перекрытием. Исследователи Беркли предложили использовать внутренний стек для того, чтобы избежать сохранения и восстановления регистров во внешней памяти. Основной целью было ускорение вызовов процедур за счет минимизации числа обращений к памяти, требуемых для передачи параметров и получения результата.

В каждый момент времени функция может иметь доступ к 32 регистрам: 8 *global registers* (r0 - r7, общие для всех функций) и окну из 24 регистров (r8 - r31). Регистровые окна перекрываются по 8-ми регистрам. В каждой функции выделяют 8 *in registers* (r24 - r31 или i0 - i7) (совпадают с *out registers* в процедуре, вызвавшей данную), 8 *local registers* (r16 - r23 или l0 - l7) (доступны только данной процедуре) и 8 *out registers* (r8 - r15 или o0 - o7) (совпадают с *in registers* в любой процедуре, вызванной данной). При вызове функции происходит переключение окон, так, что вызванная функция получает новый набор регистров l0 - l7, o0 - o7 и разделяет регистры i0 - i7 с вызвавшей функцией (где они адресовались как o0 - o7). Поэтому в регистрах o0 - o7 удобно размещать параметры для вызванной процедуры, в регистрах l0 - l7 - локальные переменные.

Размер каждого окна - 16 регистров (8 *out* + 8 *local* регистров). Фирма SUN специфицировала, что число окон может быть от 2 до 32 (в зависимости от реализации). Тем самым общее число регистров в окнах - от 40 (2x16+8) до 520 (32x16+8).

Если глубина вложенности функций превышает число окон, то процессор генерирует прерывание и операционная система должна сохранить часть окон в памяти.

Существует возможность вызвать функцию без переключения окон.
Другой важной особенностью SPARC архитектуры являются delay slots (см. с. 64).

11.6.2. Основные члены семейства

Реализация всех SPARC процессоров удовлетворяет версии архитектуры SPARC с тем или иным номером. Важнейшей функцией SPARC International Compatibility and Compliance Committee является выработка и публикация SPARC Compliance Definitions, а также инструкций по переходу от одного определения к другому.

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

SPARC (1987, 55000 транзисторов, 33 MHz, 20 MIPS):

- *136* 32-бит регистров
- 32-бит шина адреса и данных
- 4-х стадийный конвейер
- 2 исполняющих устройства: 1 IU (data + address) + 1 shift unit
- внешний CMU (Cache controller and MMU) и FPC (Floating Point Controller)
- всего 50 инструкций, каждая исполняется за 1 цикл

MICROSPARC-II (1994, 85 MHz, 85 MIPS, 64 Specint 92, 55 Specfp 92):

- 32-бит SPARC V8 архитектура
- 16Кб кэш инструкций + 8Кб кэш данных (32-бит Гарвардская архитектура)
- 3 исполняющих устройства: 1 IU + 1 FPU + 1 MMU
- интерфейс сопроцессора
- встроенная логика для поддержки DRAM и ввода/вывода

SuperSPARC-II (1995, 3100000 транзисторов, 90 MHz, 148 Specint 92, 143 Specfp 92):

- 32-бит SPARC V8 архитектура
- 3 инструкции за цикл
- 8-ми портовый 32x32 файл регистров
- 20Кб кэш инструкций + 16Кб кэш данных (32-бит Гарвардская архитектура)
- 64-входовый TLB (для обоих кэшей)
- 64-бит шина данных
- 4-х стадийный конвейер
- *7* исполняющих устройств: 3 IU + 1 FPU multiply + 1 FPU divide + 1 BU + 1 load/store unit
- биты предсказания переходов

UltraSPARC-II (1996, 250 MHz, 400 Specint 92, 450 Specfp 92):

- *64-бит* SPARC V9 архитектура
- *4* реально исполняемых инструкции за цикл
- суперскалярный процессор: *6* исполняющих устройств: 2 IU + 2 FPU + 1 BU + 1 load/store unit
- *9*-ти стадийный конвейер
- предсказание переходов
- register scoreboarding
- bi-endian (big- и little-endian порядок байтов)

- 64-бит виртуальный адрес и целочисленные данные
- спекулятивное исполнение
- многоуровневая обработка прерываний, организованных в стек
- встроенная мультимедийная поддержка для 2-х и 3-х мерной графики и новый оптимизированный набор мультимедиа инструкций
- поддержка сильно связанных процессоров (до 4-х процессоров могут разделять одну шину адреса)

11.6.3. Программная модель

Прикладной программе доступны следующие регистры.

32 целочисленных регистра общего назначения (POH) r0 – r31:
: содержат слово (32 бит). Любая пара rN, r(N+1) при четном N может содержать данные длиной 64 бит. Регистры организованы в виде окон по 24 регистра с перекрытием по 8-ми регистрам. Выделяют

  g0 - g7: это регистры r0 - r7, являются общими для всех функций (т.е. не участвуют в переключении регистровых окон). Регистр g0 - специальный: при чтении из него всегда читается 0, при записи в него ничего не происходит (запись не производится, из регистра всегда читается 0)

  o0 - o7: это регистры r8 - r15 (*out registers*); являются *in registers* для любой функции, вызванной данной (т.е. доступны вызванной функции); регистр o7 - специальный, инструкция CALL записывает в него свой собственный адрес (т.е. адрес возврата - 8)

  l0 - l7: это регистры r16 - r23 (*local registers*); не доступны ни функции, вызвавшей данную, ни функциям, вызванным данной

  i0 - i7: это регистры r24 - r31 (*in registers*); являются *out registers* для функции, вызвавшей данную (т.е. доступны вызвавшей функции и не доступны любой функции, вызванной данной)

32 регистра с плавающей точкой f0 - f31:
: содержат значение с плавающей точкой (32 бит). Любая пара fN, f(N+1) при четном N может содержать данные длиной 64 бит (floating point double), любая четверка fN, f(N+1), f(N+2), f(N+3) при кратном 4-м N может содержать данные длиной 128 бит (floating point quadre).

y (multiply/divide register):
: содержит старшую часть произведения (в инструкциях умножения) или старшую часть делимого (в инструкциях деления); читается и записывается инструкциями RDY и WRY

Целочисленные коды условия icc (integer condition code):
: являются частью PSR (Processor State Register), который не доступен пользовательской программе как регистр. Коды условия устанавливаются по результату арифметических операций (если это указано в инструкции) или специальными инструкциями и используются в командах условного перехода.

Поддерживаются следующие два режима адресации памяти.

- адрес = rA+offset (включая offset=0)
- адрес = rA+rB

где обозначено

- rA, rA - РОН r0...r31
- offset - 13-бит смещение (знаково расширяемое)

Все инструкции (за исключением load/store) имеют операнды в регистрах и потому размер всех операндов равен размеру слова (32 бит). Подавляющее большинство инструкций – трехоперандные.

В большинстве арифметических инструкций можно установить коды условия icc по результату.

11.7. Процессоры Intel 80960x

Процессоры Intel 80960x занимают значительное место на рынке встраиваемых компьютеров. На их базе построено значительное количество управляющих систем для принтеров, SCSI контроллеров, сетевых коммутаторов.

11.7.1. Общий обзор

Процессоры Intel 80960x были специально разработаны для встраиваемых систем. Являются 32-битными RISC процессорами с чертами, присущими CISC процессорам. Как RISC процессоры имеют:

- значительное число регистров (32)
- архитектуру load/store
- фиксированный формат инструкций (все инструкции, кроме load/store, имеют размер 4 байта)

Как CISC процессоры имеют:

- переменную длину инструкций (инструкции load/store могут иметь длину 8 байт, все остальные имеют длину 4 байта)
- значительное число режимов адресации памяти (11)

Специально для систем реального времени имеют:

- встроенный программируемый контроллер прерываний, подключенный к 8-ми внешним линиям прерывания
- встроенные подпрограммы (микрокод) для автоматического переключения контекста при прерывании
- встроенное ОЗУ (обычно 1Кб) для хранения таблицы прерываний
- встроенный кэш регистров, ускоряющий переключение контекста
- встроенные программируемые таймеры (ряд моделей)

Процессоры Intel 80960x особенно эффективны в приложениях, требующих обслуживания большого количества прерываний и пересылки больших объемов данных, таких, как обработка графической информации и коммуникационные задачи.

Основной особенностью процессоров Intel 80960x является использование автоматически сохраняемых при вызове процедур и переключениях контекста наборов регистров, что приводит к минимизации числа обращений к памяти. В каждый момент времени функция может иметь доступ к 32 регистрам: 16-ти *global registers* (g0 - g15), общих для всех функций, и набору из 16-ти *local registers* (l0 - l15), доступных только этой функции. При вызове процедуры происходит переключение наборов локальных регистров, так, что вызванная функция

получает новый набор регистров l0 - l15 и разделяет регистры g0 - g15 с вызвавшей функцией. Поэтому в регистрах g0 - g15 удобно размещать параметры для вызванной процедуры, в регистрах l0 - l15 - локальные переменные. Внутри процессора есть кэш для наборов локальных регистров (обычно на 4 набора). Если глубина вложенности функций превышает размер кэша, то процессор генерирует прерывание и операционная система должна сохранить часть наборов регистров в памяти. Отметим, что существует возможность вызвать функцию без переключения наборов локальных регистров.

11.7.2. Основные члены семейства

Каждый процессор в приведенных ниже технических данных обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

i80960Kx (1988, 20 MHz, 7.5 MIPS):

- 32-бит архитектура с 4Gb адресным пространством
- 32-бит *мультиплексированная* шина адреса и данных, динамическое изменение ширины шины (адаптируется к 8-ми, 16-ти и 32-х битным обменам с внешними устройствами)
- 512 байт кэш инструкций, загружаемый блочными пересылками (burst access)
- кэш контроллер (у i80960KB)
- 16 глобальных и 16 локальных регистров, кэш на 4 набора локальных регистров
- register scoreboarding (см. с. 62)

i80960Sx (1988, 16 MIPS):

- 16-бит внешняя шина данных
- Гарвардская архитектура: *512-байт* кэш инструкций + *256-байт* кэш данных
- встроенное FPU производительностью 0.5 MFlops (у i80960SB)

i80960Cx (600000 транзисторов, 33 MHz, 66 MIPS):

- 32-бит шины адреса и данных
- 128 битные внутренние шины данных
- *2* инструкции за цикл
- 1Кб кэш инструкций
- встроенное 1Кб ОЗУ
- устройство предсказания переходов
- встроенная поддержка 4-х каналов DMA (Direct Memory Access)

i80960Jx (1994, 50 MHz, 45 MIPS):

- *1* инструкция за цикл
- *8* наборов локальных регистров
- встроенный контроллер прерываний, возможность управлять кэшированием для размещения таблицы прерываний, обработчиков прерываний и их стеков в кэшируемой области
- *4Кб* кэш инструкций + *2Кб* кэш данных (у i80960JF)
- встроенное 1Кб ОЗУ
- *2 встроенных таймера*

i80960Hx (1995, 75 MHz, 150 MIPS):

- электрически и логически (т.е. по ножкам микросхемы и по исполняемому коду) совместим с i80960Cx

- *16Кб* кэш инструкций + *8Кб* кэш данных
- 32-бит шины адреса и данных с поддержкой конвейерных и блочных пересылок обеспечивают пропускную способность 160Мб/с
- встроенное *2Кб* ОЗУ
- встроенные 32-бит таймеры

i80960RP (1996, 33 MHz):

- 32-бит процессор, содержащий ядро i80960JF
- поддержка интерфейсов с двумя шинами PCI
- прямой доступ между шинами PCI и локальной шиной процессора
- встроенный контроллер памяти
- поддержка интерфейса шины $\mathbf{I_2C}$
- энергопотребление меньше 3 Вт

11.7.3. Программная модель

Прикладной программе доступны следующие регистры.

32 целочисленных регистра общего назначения (РОН) r0 - r31:
: содержат слово (32 бит). При этом

  - пара rN, r(N+1) при четном N может содержать данные длиной 64 бит
  - тройка rN, r(N+1), r(N+2) при N, кратном 4, может содержать данные длиной 96 бит
  - четверка rN, r(N+1), r(N+2), r(N+3) при N, кратном 4, может содержать данные длиной 128 бит

  Над такими операндами (длиной более 32 бит) определены только инструкции пересылки (load/store). Выделяют

  g0 - g15:
  : это регистры r0 – r15 (*global registers*); являются общими для всех функций (т.е. не изменяются при вызове функций). Регистр g15 зарезервирован в качестве указателя текущего стекового кадра (он используется процессором, когда надо сохранить или восстановить набор локальных регистров).

  l0 - l15: это регистры r16 – r31 (*local registers*); не доступны ни функции, вызвавшей данную, ни функциям, вызванным данной. При вызове функции старый набор локальных регистров сохраняется во внутреннем кэше, и вызванная функция получает новый набор, в котором:

    l0: содержит предыдущее значение указателя текущего стекового кадра

    l1: содержит указатель стека

    l2: содержит адрес возврата

    l3 - l15: значения не определены

ip (instruction pointer register):
: содержит адрес текущей инструкции. Доступен только по чтению и может быть использован при формировании адреса.

Целочисленные коды условия icc (integer condition code):
: являются частью AC (Arithmetic Control Register), который не доступен пользовательской программе как регистр. Коды условия устанавливаются специальными инструкциями и используются в командах условного перехода.

Все режимы адресации памяти процессоров Intel 80960x можно записать одной формулой: адрес ячейки памяти есть

$$\mathsf{base} + \mathsf{index} * \mathsf{scale} + \mathsf{displacement}$$

где

- base – базовый регистр: r0 – r31, ip
- index – индексный регистр: r0 – r31 (отсутствует, если *base* есть ip)
- scale – целая константа 1, 2, 4, 8, 16
- displacement – неотрицательное смещение 12 или 32 бит (12 бит смещение можно использовать только в режиме base + displacement).

Любой из элементов адреса может отсутствовать (с одним исключением: если отсутствует index, то должен отсутствовать и scale).

Все инструкции (за исключением load/store) имеют операнды в регистрах и потому размер всех операндов равен размеру слова (32 бит). Подавляющее большинство инструкций – трехоперандные.

Отметим, что это единственный из рассматриваемых нами процессоров, в котором жестко (аппаратно) закреплено, что **стек растет вверх** (от старших адресов к младшим).

11.8. Процессоры ARM

Процессоры ARM, разрабатываемые фирмой ARM (Advansed RISC Machines), играют значительную роль на рынке встраиваемых систем, особенно на рынке миниатюрных систем, сочетающих высокие пиковые нагрузки с длительными периодами ожидания (например, мобильные телефоны).

11.8.1. Общий обзор

Компания ARM (Advansed RISC Machines) была основана в ноябре 1990 года фирмами

- Acorn Computers (информационные технологии для образования, Великобритания)
- Apple Computers
- VLSI Technology

Основной целью компании является разработка микропроцессорных ядер и их лицензирование широкому кругу производителей. В силу малости процессорного ядра ARM (всего 35000 транзисторов в базовом ядре ARM7) оно идеально подходит для интеграции в специализированные микросхемы потребителей. ARM Design Service Group постоянно работает с партнерами, обеспечивая ARM экспертизу OEM потребителям, желающим иметь встроенные в микросхемы решения на основе ядер ARM.

В настоящее время следующие компании лицензировали ARM и производят микросхемы на его основе:

1. VLSI Technology
2. Texas Instruments (TI)
3. Samsung Corporation
4. NEC Corporation
5. GEC Plessey Semiconductors (GPS)

6. Cirrus Logic
7. Digital Equipment Corporation
8. Symbios Logic
9. Sharp Corporation
10. Asahi Kasai Microsystems (AKM)
11. European Silicon Structures (ES2)
12. Lucky Goldstar Corporation
13. Intel Corporation
14. IBM Corporation

На основе ARM ядра разработано более 30 микропроцессоров и специализированных микросхем. Они находят применение в сотовых телефонах, органайзерах, модемах, графических ускорителях, видеофонах, камерах, телефонных коммутаторах, игровых приставках, дисковых накопителях, высокопроизводительных рабочих станциях, автомобильных навигационных системах, цифровых декодерах, smart картах, лазерных принтерах.

Основные отличительные черты архитектуры ARM:

1. Все инструкции являются условными (т.е. выполняются, только если код условия совпадает с кодом, указанным в инструкции). Это позволяет увеличить плотность кода и уменьшить потребность в инструкциях близкого перехода. Как следствие, нет отдельных команд условного перехода.
2. Все целочисленные арифметические инструкции могут выполнять операцию сдвига над операндами за тот же цикл, что выполняется и сама инструкция. Как следствие, нет отдельных команд сдвига.
3. Нет целочисленной инструкции деления.
4. Возможность выполнять DSP-подобные функции:
   а) присутствуют инструкции умножения и умножения со сложением (multiply-accumulate (MLA))
   б) присутствуют инструкции блочного чтения из памяти и блочной записи в память, позволяющие переслать любое подмножество из 16-ти регистров общего назначения.
5. Некоторые модели могут работать в так называемом **THUMB** режиме: инструкции кодируются 16-ю битами вместо 32-х. Это значительно увеличивает плотность кода, но накладывает ряд ограничений на систему команд:
   а) полноценно доступны только 8 регистров из 16-ти, остальные могут ограниченно использоваться только в некоторых инструкциях (например, MOV, ADD и CMP);
   б) не поддерживается условное исполнение инструкций, как следствие, появилась новая инструкция условного перехода;
   в) не поддерживается операция сдвига над операндами в целочисленных арифметических инструкциях, как следствие, появились новые инструкции сдвига;
   г) все инструкции двухоперандные (а не трехоперандные как в обычном режиме).

11.8.2. Основные члены семейства

Каждый процессор в приведенных ниже технических данных в-основном обладает всеми возможностями предыдущих процессоров для обеспечения совместимости.

ARM1: Прототип, использовался только в тестовых системах

ARM2 (8 MHz, 4.7 MIPS):
64Кб адресное пространство

ARM3 (33 MHz, 18 MIPS):
- ARM2 ядро
- 4Кб единый кэш
- интерфейс сопроцессора
- добавлена новая инструкция SWP для работы с семафорами

ARM6 (36000 транзисторов, 33 MHz, 28 MIPS):
- 4Гб адресное пространство
- bi-endian (big- и little-endian порядок байтов)
- интерфейс сопроцессора

ARM600:
ARM6 со встроенным MMU

ARM7 (35000 транзисторов):
- ARM6 ядро, способное работать на повышенной частоте
- 3-х стадийный конвейер
- улучшенная инструкция аппаратного умножения (нужна для работы DSP)

ARM7D:
ARM7 с поддержкой отладки

ARM7DM:
ARM7D с улучшенным умножением

ARM7DMI (40 MIPS):
ARM7DM с ICEbreaker (встроенная поддержка In Circuit Emulation)

ARM70DM:
ARM7DMI (как отдельная микросхема)

ARM700 (40 MHz, 36 MIPS):
- ARM7 ядро
- 4Кб единый кэш
- writeback buffer
- встроенное MMU

ARM7500:
- ARM7 ядро
- 8Кб единый кэш
- writeback buffer
- встроенное MMU
- встроенный IOMD
- встроенный видеопроцессор

ARM7Txx:
: ARM7xx (xx - одно из приведенных выше сочетаний) с поддержкой THUMB режима

ARM8 (80 MHz, 80 MIPS):
- совместим с ARM6 и ARM7
- 5-ти стадийный конвейер
- спекулятивное исполнение

StrongARM (SA110: 100 MHz, 115 MIPS; 200 MHz, 230 MIPS):
- высокоскоростной вариант ARM ядра, разработан совместно ARM ltd и Digital
- 16Кб кэш инструкций + 16Кб кэш данных (Гарвардская архитектура)
- глубокий конвейер
- полная совместимость кода не гарантируется в силу появления глубокого конвейера и раздельного кэша

AMULET2e (40 MIPS):
- это асинхронная версия ARM6, более быстрая, чем ARM7, но более медленная, чем ARM8
- 150 mW в активном состоянии, 0.1 mW в состоянии ожидания
- малое потребление мощности и механизм использования энергии делают AMULET2e идеальным процессором для приложений, где периоды высокой вычислительной нагрузки сочетаются с длительными периодами ожидания ввода

11.8.3. Программная модель

Прикладной программе доступны

16 целочисленных регистров общего назначения (РОН) r0 - r15:
: содержат слово (32 бит). Некоторые из этих регистров имеют специальное назначение:

  r15 - program counter (pc):
  : содержит адрес инструкции, находящейся через две инструкции от исполняемой в данный момент (т.е. адрес текущей инструкции + 8; при записи в этот регистр происходит переход по записанному адресу

  r14 - link register (lr):
  : после инструкции Branch and Link (BL) (вызов функции) содержит адрес следующей инструкции (адрес возврата); во всех остальных инструкциях это обычный РОН

Коды условия cc (condition code):
: являются частью CPSR (Current Program Status Register), который не доступен пользовательской программе как регистр. Коды условия устанавливаются по результату арифметических операций (если это указано в инструкции) или специальными инструкциями и используются для определения того, нужно ли исполнять текущую инструкцию (напомним, все инструкции являются условными).

Поддерживаются следующие режимы адресации памяти.

Для чтения/записи слова (32 бит) или беззнакового байта:
: имеется один режим адресации

$$\text{базовый регистр} \pm \text{смещение}$$

где базовый регистр – один из РОН r0 – r15, а смещение может иметь три вида:

- константа – 12-битная константа

- регистр – один из РОН r0 – r15
- (регистр <операция> константа) где
  - регистр – один из РОН r0 – r15,
  - <операция> – одна из следующих операций сдвига

    LSL: Logical Shift Left, сдвиг влево, вдвигаются нули

    LSR: Logical Shift Right, сдвиг вправо, вдвигаются нули

    ASR: Arithmetic Shift Right, сдвиг вправо, вдвигается знаковый разряд операнда регистр

    ROR: ROtate Right, циклический сдвиг вправо

    RRX: ROtate Right with eXtend циклический сдвиг вправо 33-х битной величины (<бит переноса>, регистр) на 1, правый операнд (т.е. константа) должен отсутствовать
  - константа – 5-бит константа, задающая число сдвигов

Каждый из трех видов режима адресации имеет три варианта (что дает девять режимов адресации):

- **обычный**, адрес есть сумма базового регистра и смещения
- **с преиндекированием**, адрес есть сумма базового регистра и смещения, если инструкция чтения/записи выполнена (удовлетворен ее код условия), то адрес записывается в базовый регистр
- **с постиндексированием**, адрес есть базовый регистр, если инструкция чтения/записи выполнена, то сумма базового регистра и смещения записывается в базовый регистр

Для чтения/записи полуслова (16 бит) или чтения знакового байта:

(есть только в архитектуре ARM4 и выше) имеется один режим адресации

базовый регистр ± константа

где базовый регистр – один из РОН r0 – r15, а константа – 8-битная константа. Этот режима адресации имеет три варианта: **обычный**, **с преиндекированием** и **с постиндексированием**.

Все инструкции имеют поле кода условия (4 бита). Если текущее состояние флагов в регистре CPSR совпадает с указанным в поле кода текущей инструкции, то она будет выполнена, иначе - пропущена.

Все инструкции (за исключением load/store) имеют операнды в регистрах и потому размер всех операндов равен размеру слова (32 бит).

Все арифметические инструкции (включая логические) имеют бит, при установке которого по результату операции будут выставлены коды условия.

Отличительной особенностью процессоров ARM является то, что все арифметические инструкции (включая логические) могут использовать в качестве одного из операндов так называемый <shifter_operand>, который может иметь одну из следующих форм.

<immediate>:

<shifter_operand> = <immediate>, где <immediate> – 32 бит константа, в которой только в каких-то 8-ми подряд идущих позициях могут быть не нули, номер первой позиции должен быть четным; кодируется в инструкции как 8-ми битная константа <8_bit_immediate> и 4-х битный сдвиг <rotate_imm>, при этом <shifter_operand> = <8_bit_immediate> Rotate_Right (<rotate_imm> * 2)

rA: <shifter_operand> = rA, где rA – один из РОН r0 – r15

rA <shift> <shift_imm>:

<shifter_operand> = rA <shift> <shift_imm>, где <shift> есть одна из операций

LSL: Logical Shift Left, сдвиг влево, вдвигаются нули

LSR: Logical Shift Right, сдвиг вправо, вдвигаются нули

ASR: Arithmetic Shift Right, сдвиг вправо, вдвигается знаковый разряд rA

ROR: ROtate Right, циклический сдвиг вправо

RRX: ROtate Right with eXtend циклический сдвиг вправо 33-х битной величины (<бит переноса>, rA) на 1, правый операнд (т.е. <shift_imm>) должен отсутствовать

<shift_imm> - 5-ти битная константа

rA <shift> rB:

<shifter_operand> = rA <shift> rB, где <shift> есть одна из описанных выше операций, <shift_imm> - 5-ти битная константа

Отметим, что при использовании последней формы <shifter_operand> трехоперандная арифметическая инструкция реально использует 4 регистра.

12. Архитектура системной шины

Системная шина обеспечивает взаимодействие процессора и периферийных устройств. Поскольку часто основной задачей системы реального времени является управление тем или иным оборудованием, то качеству системной шины для промышленных компьютеров уделяется повышенное внимание.

В настольных компьютерах и промышленных системах распространены следующие системные шины.

Сравнение системных шин	
Название шины	**Производительность Мб/с**
PC/XT (8 бит)	4.7
PC/AT (16 бит)	16.66
MULTIBUS 1	24
EISA	33
VME32	40
MCA32	33
MULTIBUS 2	70
VME64	80
NUBUS	80
PCI32	132
VLB32	135
MCA64	160
AUTOBAHN 1	200
PCI64	264
AUTOBAHN 2	400
PCI64-66	528
FUTUREBUS+	1000

Отметим, что при выборе шины часто приходится руководствоваться не только ее производительностью, а и такими факторами, как наличие периферийного оборудования для нее, возможность “горячей” (т.е. без выключения компьютера) замены оборудования на шине и т.д. Мы рассмотрим господствующую в настоящее время среди промышленных систем шину VME, а также приобретающую все большую популярность шину PCI.

12.1. Архитектура шины VME

Стандарт на шину VME появился в 1981г., когда фирмы Motorola, Mostek и Signetics договорились об едином стандарте на шину для промышленных систем. В основу электрической спецификации шины был положен стандарт на шину VERSAbus фирмы Motorola, а в основу механической спецификации – стандарт Eurocard консорциума европейских компаний. Новый стандарт был назван VERSAmodule Eurocard, сокращенно VME. Однако, все компании, кроме Motorola, отказались расшифровывать VME как "VERSAmodule Eurocard", поскольку "VERSAmodule" является зарегистрированной торговой маркой Motorola. Поэтому в настоящее время считается, что термин "VME" является единым обозначением, а не аббревиатурой. На шину VME существует стандарт IEEE 1014-1987.

Плата шины VME может быть одинарной или двойной высоты. Плата одинарной высоты (или 3U) имеет размеры 100мм × 160мм и один 96 контактный разъем DIN 41612, называемый P1. Плата двойной высоты (или 6U) имеет размеры 233мм × 160мм и второй 96 контактный разъем DIN 41612, называемый P2. Лицевая сторона платы имеет ширину 20мм. Платы разного размера могут работать совместно в одной системе.

На шине VME может быть до 21 разъема. Питание на платы подается через разъемы шины.

Основные характеристики шины.

- VME является асинхронной шиной.
- VME использует идею полного отображения на память. Каждое устройство представляется как адрес памяти или группа адресов.
- VME имеет раздельные шины адреса и данных. Плата одинарной высоты имеет 24-битную шину адреса и 16-ти битную шину данных, плата двойной высоты – 32-битные шины адреса и данных.
- Возможность блочных пересылок, когда адрес задается только один раз.
- Поддержка мультипроцессорности:
  - приоритетный доступ к разделяемым ресурсам
  - возможность параллельного или исключительного доступа к разделяемым ресурсам
  - 4 уровня доступа
  - структура доступа master/slave
  - защита разделяемых ресурсов посредством неделимых циклов шины
- 7 уровней запроса прерываний

Типичный обмен по шине состоит из цикла арбитража (для получения доступа к шине), адресного цикла (для выборки устройства), цикла реального обмена данными.

Шина VME как бы состоит из 4-х подшин.

1. **Шина арбитража** отвечает за определение приоритета запроса и разделение шины. В качестве арбитра выступает плата, находящаяся в разъеме 1 шины VME.
2. **Шина обмена данными** включает
   - шину данных, по которой возможны обмены размером 8/16/32 бит
   - шину адреса, по которой возможны обмены размером 16/24/32 бит
   - управляющую шину, по которой передаются размеры адреса и данных, тип обмена (чтение или запись), признак окончания обмена, признак ошибки и т.д.

3. **Шина управления прерываниями** отвечает за приоритетное управление прерываниями

4. **Служебная шина** отвечает за подачу напряжений питания +5В, -12В, +12В на платы, информирование последних о проблемах с питанием, подачу периодического сигнала, инициализацию и сброс шины.

Блочный режим позволяет передать до 256 байтов по шине данных, однократно задав адрес на шине адреса. Следовательно, шина адреса неиспользуется большую часть времени обмена. Стандарт VME64 определяет 64-битные передачи за счет не используемой шины адреса: младшая часть 64-битного блока передается по шине данных, а старшая часть – по простаивающей шине адреса.

12.2. Архитектура шины PCI

Шина PCI (Peripheral Components Interconnect) появилась как стандарт фирмы Intel на соединение локальной шины процессора (соединяющей его с памятью) и периферийных устройств. В 1992г. появилась версия 1 стандарта, а в 1993г. – версия 2. В 1995г. появился стандарт на расширение PCI-66MHz.

Основные характеристики шины.

- PCI является синхронной шиной, частота синхронизации 33MHz или 66 MHz.
- PCI использует три адресных пространства:
  1. **пространство памяти**
  2. **пространство ввода/вывода**
  3. **пространство регистров конфигурации**
- PCI имеет мультиплексированную шину адреса и данных размером 32 бит или 64 бит.
- Возможность подключения до 256 шин PCI
- Параллельная работа различных шин
- Буферизация обменов
- Протокол конфигурации устройств

Предметный указатель (русский)

Предметный указатель (английский)