Сергей Вартанов

Основы построения операционных систем.

Учебно-методическое пособие.

Ереван – 2026

Текст книжки на русском языке.

ՍԵՐԳԵՅ ՎԱՐՏԱՆՈՎ

ՕՊԵՐԱՑԻՈՆ ՀԱՄԱԿԱՐԳԵՐԻ ԿԱՌՈՒՑՄԱՆ ՀԻՄՈՒՆՔՆԵՐԸ

Ուսումնամեթոդական ձեռնարկ

ԵՐԵՎԱՆ - 2026

Оглавление.

Введение.

Глава 1. Архитектура универсального вычислителя.
1.1. Вычислители и их устройство.
1.2. Адресация.
1.3. Прерывания.
1.4. Кэш-память процессоров.

Глава 2. Операционные системы, причины их создания и разновидности.
2.1. Логика многозадачности.
2.2. Службы операционных систем.
2.3. Требования к операционным системам.
2.4. Загрузка операционных систем.
2.5. Задачи и подзадачи, процессы и потоки.
2.6. Фоновые и резидентные программы.

Глава 3. Задачи, задания и управление ими.
3.1. Запуск задач и заданий.
3.2. Загрузчик и Стартер задач.
3.3. Блоки управления задачами (приложения и потоки).
3.4. Переключение задач и потоков.
3.5. Прикладные задачи и запросы ресурсов операционных систем.

Глава 4. Память и управление ею.
4.1. Формирование памяти вычислителей и приложений.
4.2. Оверлеи, свопинг.
4.3. Организация виртуальной памяти.

Глава 5. Особенности разработки программ ОС.
5.1. Реентерабельные программы.
5.2. Копии программных модулей.
5.3. Мьютексы.

Глава 6. Ввод-вывод. Файловые системы. Методы доступа.
6.1. Устройства ввода-вывода и методы доступа к ним.
6.2. Файловые структуры.
6.3. Системы ввода-вывода с буферизацией.

Глава 7. Многомашинные и многопроцессорные комплексы.
7.1. Формирование многомашинных и многопроцессорных систем.
7.2. Структуры управления в многомашинных комплексах и задачи ОС.
7.3. Организация общей памяти в многомашинных комплексах.

Введение.

Данное руководство посвящено описанию курса по операционным системам (ОС), в котором изложены задачи ОС, принцы построения ОС, методы решений задач управления ресурсами вычислительных комплексов.
Начинается курс с предыстории возникновения и эволюции в понимании вычислений, формировании теоретических и практических основ при создании различных моделей вычислений и вычислителей.
Вычисления и вычислительные науки имеют вековую историю, если исходить от начала целенаправленных и полноценных исследований. В первую очередь, с коллективных обсуждений многими замечательными математиками содержательного понятия "алгоритм вычислений" и присущих ему качествах.
Как итог, был сформулирован т.н. Тезис Чёрча (иногда – Тезис Черча-Тьюринга), который был расширен С. Клини до утверждения, что все частичные функции, вычислимые посредством алгоритмов, совпадают с классом частично-рекурсивных функций, эквивалентным классу всех Машин Тьюринга.
Тезис Чёрча до сих пор является основой для понимания и исследований в области вычислений и вычислительной техники.
В 1941-ом году немецким инженером Конрадом Цузе (Zuse) был создан первый в мире реально работающий компьютер Z3 (Z наверняка от первой буквы фамилии, хотя это не афишировалось), а несколько позже - и первый в истории язык высокого уровня Plankalkül (для вычислителя Z4), название которого можно трактовать как "запланированное вычисление".
В 1940-е началось бурное развитие как теоретических моделей организации, алгоритмизации и программирования вычислений, так и создание, в основном, лабораторных вычислителей. Хотя, именно Z4 стал первым в мире коммерческим компьютером, проданным в 1950-ом году. Так или иначе, вычислители оставались достаточно уникальными устройствами. Исследования и разработки велись, в основном, в направлении создания новой элементной базы, а также реализации теоретических моделей вычислений, таких, как
- машины последовательных вычислений (т.н. "модель фон Неймана");
- векторные вычисления;
- векторно-конвейерные вычисления;
- ассоциативные вычисления;
- матричные вычисления;
- потоковые вычисления;

которые получили ещё одно разделение по подходам и архитектуре: последовательные вычисления и вычислители, и параллельные вычисления и вычислители.
Практически, во всех перечисленных моделях стали присутствовать элементы многопроцессорности, по крайней мере, были имплантированы специализированные субпроцессоры, используемые для отдельных функций (как правило – для реализации операций ввода-вывода).
Одновременно вычислительные установки стали разделять по принципам организации: однопроцессорные, многопроцессорные и многомашинные вычислительные комплексы.
Во второй половине 1960-х сложилась базовая теоретическая основа для описания параллельных вычислений, начиная с работы Карпа и Миллера [2], послужившей стимулом для разработки различных эволюций параллельных моделей вычислений. В то же время Флинном (Flynn) [3] была предложена, ставшая классической, классификация вычислителей и вычислений, из четырех классов которой только один относился к традиционным тогда последовательным вычислениям.
Знаковым событием 60-х стало появление архитектуры IBM-360, ставшей, по сути, эталонной для однопроцессорных вычислителей. В последующие годы системы 360/370 стали обрастать обширными линейками периферийных устройств, обеспечивающих ввод, вывод и ввод-вывод информации. К таким можно отнести дисплейные станции (сначала алфавитно-цифровые, позже добавились графические), всевозможные принтеры, плоттеры, сканнеры и пр. Росли возможности дисковых устройств, постепенно оттесняя магнитные ленты и другие на бумажных и пр. носителях.

Добавились и средства телекоммуникаций, позволяющих взаимодействовать с вычислителями на удаленных расстояниях. Одновременно стали появляться многопроцессорные модели вычислителей.
По существу, появление именно IBM-360/370 в плане архитектуры, различных вариантов межмашинного и межпроцессорного взаимодействия, организации многозадачных решений, стандартизации решений и пр., в том числе, в создании культуры и принципов разработок как в части электроники, так и в части программного обеспечения, стали эталонными. Здесь же были сформулированы и основные требования к операционным системам (ОС), а также принципы их построения.
В данном учебном курсе мы не останавливаемся на конкретных особенностях конкретных ОС, а описываются наиболее распространенные решения тех или иных компонентов.
В курсе будут представлены
- принципы построения вычислителей;
- принципы работы процессоров, обработки машинных кодов на базе двухадресных, трехадресных и многоадресных математических машин;
- значимость и принципы организации систем прерываний, а также возможности работы с синхронными и асинхронными событиями в целом;
- обоснования к созданию операционных систем, их общая структура, функциональные задачи;
- задачи ядра ОС и их служб;
- схемы загрузки ОС;
- варианты формирования очередей заданий для ОС;
- схемы загрузки задач и их старта.

Кроме того в курсе рассматриваются основания к организации многозадачности, различные варианты ее организации (такие, как реально-параллельные системы на основе многопроцессорности, системы с квантованием времени, возможности распараллеливания на основе операций и событий ввода/вывода, и др.).
В курсе представлены способы разделения памяти на память ОС и память приложений, формы регистрации запрошенной памяти операционной системой и приложениями.
Рассматриваются возможности формирования пулов переменных (стеки вызовов, динамическая память приложений и отдельных подзадач), динамически запрашиваемых элементов оперативной памяти в моменты загрузки, старта и выполнения задач и подзадач.
В контексте существования программных модулей операционных систем, обслуживающих множество вычислительных процессов, рассмотрены проблемы реентерабельности при разработке программ, способы динамической загрузки копий программных модулей, в том числе, динамически загружаемых программных библиотек.
В курсе представлены различные способы организации взаимодействий с устройствами ввода/вывода при архитектурах, позволяющих
- прямую миграцию данных в оперативную память и архитектурах с вводом/выводом, осуществляемым центральным процессором;
- ввод/вывод с буферизацией и без таковой;
- разнообразие системы прерываний по вводу/выводу;
- принципы взаимодействия с вычислительными сетями.

Завершающим этапом курса является представление многомашинных систем, в частности, кластеров. Рассмотрены способы межмашинных соединений, а также задачи управления в многомашинных системах.
Для проведения учебного курса была создана демонстрационная программа с возможностями отображения всех перечисленных выше тем и имитацией отдельных схем работы процессоров, обработки прерываний, вариантов многозадачности и др.

*) Все изображения, представленные рисунками, являются скриншотами с экрана демонстрационной программы с эмуляцией поведения вычислителей и ОС. Программа написана автором на языке императивного параллельно программирования Caper.

Глава 1. Архитектура универсального вычислителя.

В качестве преамбулы к главе сформулируем несколько основополагающих тезисов.
Архитектура вычислителя предопределяет архитектуру операционной системы для него.
Архитектура вычислителя предопределяет концепцию программирования для него.
Архитектуры современных универсальных вычислителей в разной степени наследуют т.н. фон Неймановскую структуру, которая предполагает наличие трех компонент в устройстве: процессор, оперативная память и средство ввода-вывода информации. При этом, наиболее часто упоминается принцип размещения программных кодов на машинном языке именно в оперативной памяти, как и данных. В этой связи отметим, что программный код может быть изменен в ходе вычислений как извне, так и самой программой (см. об изменении и самоизменении программ ниже). Однако, в определенных архитектурах присутствует защищенное разделение на память программы и память данных.
Современный компьютер, любая вычислительная установка является совокупностью разнообразных ресурсов, которыми следует управлять.
Для исполняемых на вычислителях программ возникают задачи доступа к ресурсам и их использования, что требует рутинного программирования. В то же время, в среде вычислителей одновременно может присутствовать и исполняться множество вычислений. Соответственно, возникают проблемы конкуренции за ресурсы при их захвате, использовании и освобождении.
Операционные системы – программно-аппаратные комплексы, предназначенные для загрузки и выгрузки пользовательских (прикладных) программ, инициирования их выполнения, контроля всего ряда событий, происходящих в вычислительной установке, для контроля и управления ресурсами вычислителей, в том числе, процессоров.
Операционная система и ее основные компоненты работаю с ресурсами по принципу Монитора, т.е. доступ и манипулирование всяким ресурсом осуществляется посредством соответствующих компонентов ОС, без самостоятельного и непосредственного доступа к ресурсу.

[1]) Принятое понятийное разделение методов в управлении совместно используемыми ресурсами: Семафоры, Мониторы, Сентинелы (по [1]).

1.1. Вычислители и их устройство.

Всякий вычислительный комплекс, начиная с отдельного компьютера и заканчивая сложными многопроцессорными и многомашинными вычислителями, является
- системой, построенной на основе теоретического подхода в понимании вычислений и их реализации;
- системой, содержащей в себе электронные и программные компоненты, образующие совокупность ресурсов;
- системой, в структуре которой присутствуют основные элементы, осуществляющие процесс вычислений - процессоры, носители оперативной памяти и базовые средства коммуникаций (системные шины и специальные шины подключения и взаимодействия процессоров и оперативной памяти), и вторичные средства, часто называемые периферийными: ввода и вывода информации, ее хранения и транспортирования.

Теоретические принципы построения систем и их отдельных элементов называют архитектурами (процессоров, оперативной памяти, периферийных устройств и пр.).

В некоторых архитектурах процессоры, оперативная память, элементы управления вводом/выводом могут быть разнесены по типу, предназначению, задачам. К примеру, DSP-процессоры (процессоры обработки сигналов) обладают статической и динамической памятью, а собственно сам процессор является интегратором набора специализированных субпроцессоров.
Универсальные процессоры, которые являются основными устройствами исполнения программ, называются центральными процессорами (в англоязычной терминологии – CPU - Central Processing Unit).
На Рис.1 представлены варианты машин процессоров: двухадресная, наиболее распространенная, трехадресная и многоадресная.
Двухадресные процессоры работают по следующей схеме:

1. Во внутренней памяти процессора формируются значения и/или указания на первый и второй операнд.
2. КОП (код операции) – как правило, указание соответствующей коду секции в памяти с микропрограммами на реализующую операцию микропрограмму.
3. Результат работы микропрограммы размещается, как правило, в первом операнде.

Трехадресные процессоры работают примерно так же, как и двухадресные, однако результат операции размещается по месту, указанному третьим операндом.
Многоадресные процессоры могут работать по разным схемам, в зависимости от общего подхода к архитектуре вычислителя. В частности, в случае архитектуры с возможностями множественного присвоения, в третьем и далее операндах может быть размещен результат текущей операции. *)
Еще одним вариантом может быть исполнение в рамках схемы SIMD (Single Instruction Multiple Data) по классификации Флинна [3], когда одна операция применяется ко всему множеству операндов.
Существуют архитектуры с возможностью записи нескольких команд, которые могут быть выполнены за один такт процессора. Наиболее известны суперскалярные и VLIW архитектуры, которые относятся к курсу по параллельным вычислителям и вычислениям.
Здесь же, в целях демонстрации структур операционных систем и их функций, мы ограничимся традиционными однопроцессорными и многопроцессорными архитектурами фон Неймана.
Ранние (до появления т.н. кэшей) процессоры на каждом элементарном шаге (может состоять из нескольких тактов) выполняют следующие действия:

1. Осуществляется одна или более выборка данных фиксированной длины из оперативной памяти (позже - из внутренних кэшей процессора) в регистр процессора. Именно фиксированная длина выбора данных (прямо зависит от количества передаваемых бит по шине, соединяющей процессор и память) за один шаг стала предтечей типа “слово” (word) в языках программирования (напомню, что слово со знаком - int). При этом в ранних версиях вычислителей IBM при длине слова в 4 байта были команды и в 6, и в 8 байтов, что требовало провести, по крайней мере, две выборки.
2. В большинстве архитектур процессоров присутствует регистр (Instruction Pointer – IP, в IBM – это часть управляющего регистра PSW), в котором хранится адрес следующей команды, который формируется на основе кода операции и длин соответствующих операндов. Т.е. увеличением адреса текущей команды на ее длину формируется адрес следующей команды.
3. Далее, процессором осуществляется подготовка операндов. При этом данные уже могут располагаться в регистрах процессора, и тогда все упрощается, или располагаться в оперативной памяти, что потребует от процессора их перекачки во внутренние регистры. И опять же длиной в размер слова выборки.
4. Только после подготовки и необходимых размещений данных в регистрах осуществляется инициирование работы микропрограммы, соответствующей коду операции. По ее завершению результат размещается, как правило, по адресу первого операнда для двухадресных машин.
5. Осуществляется переход к выполнению следующей операции по содержимому IP.

Отметим, что присущие всем универсальным процессорам средства организации итераций (проще говоря, команды условного или безусловного перехода по адресу команды) интерпретируются как размещение в IP или PSW нового адреса следующей команды.
По существу, процессор является интерпретатором выбираемых команд (см. на Рис. 1 стрелки, указывающие на соответствующие кодам операций микропрограммы).

*) PSW - Program Status Word, слово состояния программы, принятое в IBM-360 [4] и присутствующее в современных ESA/390 (IBM-390) [5]. Что самое замечательное, IBM сохранило архитектурные особенности своих машин, фактически, только увеличив вдвое размеры регистров общего пользования и управляющих регистров (что говорит о глубокой продуманности архитектуры еще в 1960-е годы, изменяя в развитие только элементную и количественную основы). Так, если у IBM-360 использовались три последних байта из восьми под адрес следующей команды, то у современных моделей общие регистры имеют 8 (64 бита), а PSW для режима z/Architecture – 16 байтов (128 бит). Адрес следующей инструкции занимает последние 8 байт. В любом случае, будь то отдельный регистр или часть другого архитектурного элемента процессора, указатель следующей инструкции мы будем обозначать как IP.

*) В классической схематологии программ множественное присвоение представляется выражением $y_1, y_2, \ldots, y_n := f(x_1, x_2, \ldots, x_m)$, т.е. результат f над **m** параметрами размещается в **n** переменных (местах) за один шаг вычисления.

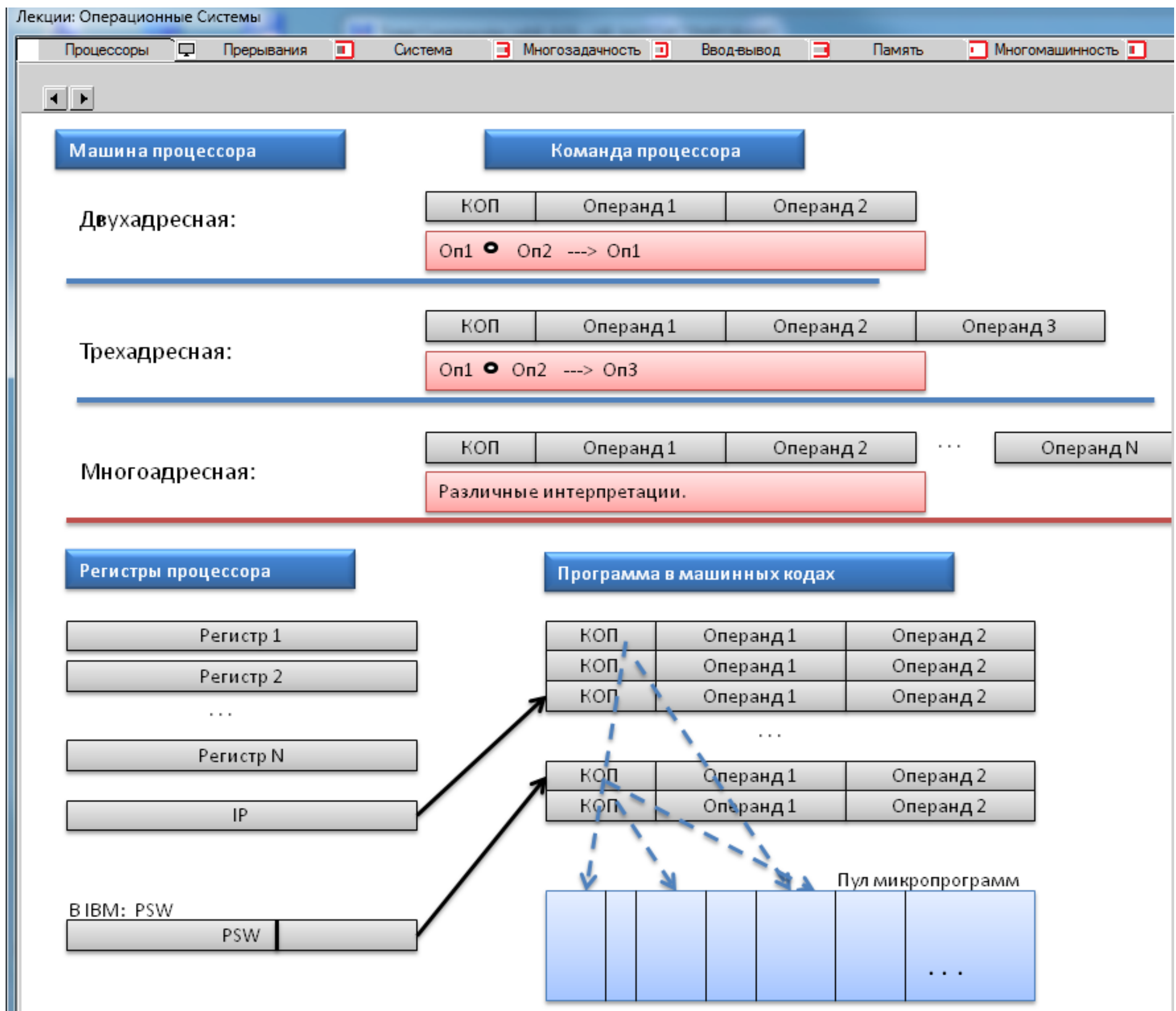


Рис.1. Базовые элементы процессоров.

Подчеркнем, что основой подходов к конструированию вычислителей является булева алгебра, а микропрограммы формируются по уже сложившейся традиции на операциях OR, AND, NOT и добавленной для удобства XOR.

1.2. Адресация.

Необходимо понимание таких понятий как абсолютная и относительная адресация, в первую очередь, программ (см. Рис.2). Чаще всего эти понятия используются при размещении программ и данных в оперативной памяти, хотя по логике адресации такой подход применим и в других случаях.
Абсолютный адрес – номер байта в последовательности байтов от 0 до M-1, где M – объем памяти.
Относительный адрес формируется из содержимого двух регистров и константы: (B)+(I)+D, Где B – т.н. базовый регистр, содержащий абсолютный адрес, I – индекс-регистр, D – положительное число, называемое смещением.
Как правило, понятия базового регистра и смещения присутствует во всех компьютерах при формировании адреса, находящегося в памяти, выделенной под приложение (прикладную программу). При этом базовый регистр, как правило, содержит абсолютный адрес начала программы, индекс-регистры используются в тех же вычислителях IBM для гибкости при динамических изменениях адресов внутри приложений, т.е. задает смещение относительно базового в динамической форме, в то время, как абсолютное смещение – число постоянного

отступа от какого-либо адреса. Отдельные процессоры могут использовать сокращенную, без индекс-регистра, форму адресации: A = (B) + D.
Особое внимание следует обратить внимание на понятие выровненности адреса на длину слова выборки, т.к. процессор может выбирать данные из оперативной памяти только по адресам, кратным длине слова.
В параметрах компиляции исходных кодов программ есть и указание на выравнивание (в частности, в Visual Studio), что учитывается при создании структур данных и объектов в ООП при размещении и адресовании их членов.

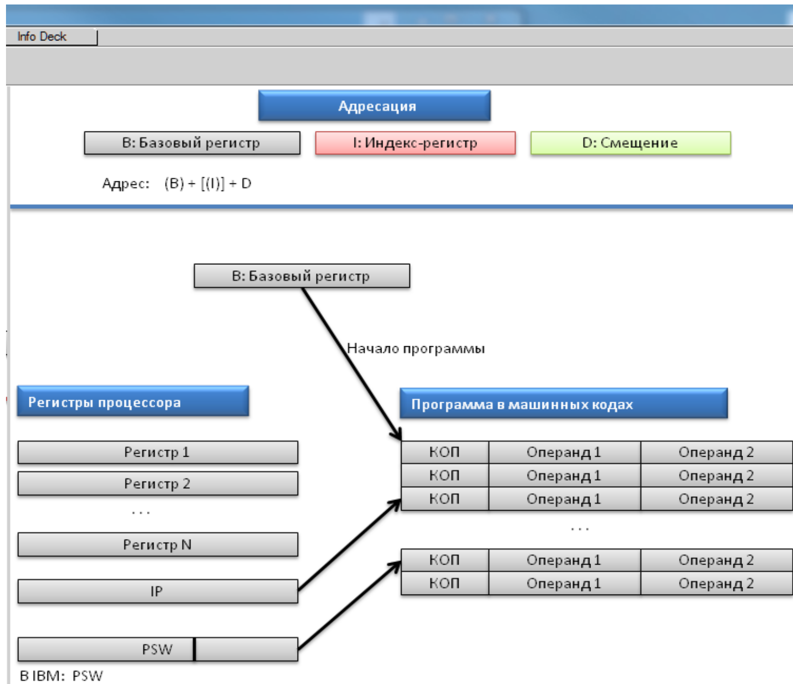


Рис. 2. Адресование инструкций в памяти с помощью IP и PSW.

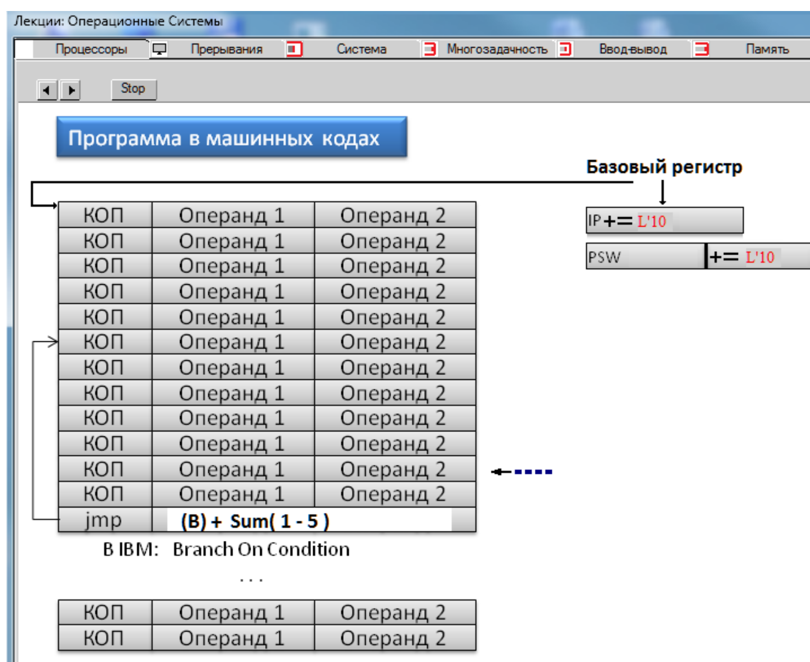


Рис. 3. Формирование адреса перехода: к содержимому базового регистра прибавляется сумма длин команд, расположенных до точки перехода по команде “jmp”. Данная

сумма вставляется в IP (PSW).

На Рис.3 показано формирование адресов в IP. Базовый регистр указывает на начало программы (в целом – на любую выбранную последовательность команд). В IP и PSW – адрес, сформированный содержимым базового регистра, просуммированного с длиной 10-ти команд. Стрелка справа эмулятора работы процессора указывает на текущую исполняемую команду. Команда "jmp" в операндах имеет относительный адрес, сформированный суммой содержимого базового регистра и суммы длин первых пяти команд, и являющийся указанием на точку перехода. Команды перехода осуществляют размещения адресов в параметрах в регистр IP. Соответственно, для процессора следующей исполняемой командой станет указанное в IP.
По существу, процессор – потенциально бесконечно работающее (по крайней мере, пока есть электропитание и он работоспособен) устройство по пошаговой интерпретации команд из оперативной памяти (ничто, кроме эффективности, не мешает сделать процессор для считывания команд и их исполнения в какие-нибудь регистры с магнитного диска, ленты и пр., т.е. с места, где команды будут записаны и доступны для считывания; хоть с узелковой "записи" (кипу) инков, было бы устройство). Даже остановка процессора по команде, присущая процессорам IBM, это всего лишь переход в состояние, но никак не прекращение работы процессора. Большинство современных процессоров запускают т.н. Idle-процесс – компонента ядра ОС, которая выполняется на всех незанятых подпроцессорах (ядрах); используется с целью сбережения энергии.

1.3. Прерывания.

Принципы построения и функционирования процессоров – одна, но не единственная, из краеугольных основ построения систем управления вычислителями.
Безусловное и определяющее значение имеют такие компоненты, как устройство оперативной памяти, устройство взаимодействия с внешними устройствами ввода/вывода информации, наличие дополнительных устройств вроде аппаратных таймеров и некоторых других.
К таким компонентам относится и система аппаратных прерываний, которая обеспечивает немедленное реагирование процессора на события внутри и вовне него. По сути, это система реагирования процессора на сигналы от нескольких совокупностей триггеров, выдающих сигналы о событиях - прерываниях.
И опять "законодателем мод" здесь явились вычислители IBM, разработчики которых выделили следующие группы в качестве источников прерываний:
- аппаратные ошибки самого процессора;
- аппаратные ошибки оперативной памяти (обычно, на основе средств контроля четности в байтах, или ошибок управления памятью);
- создание внутреннего события процессора с помощью т.н. команд супервизора (Supervisor Call, SVC *));
- программные прерывания из-за ошибок программ (к примеру, деление на ноль, переполнения, обращение к "чужой памяти", и пр.);
- события ввода/вывода (пожалуй, наиболее разветвленная система событий в силу большого количества различий в устройствах, а также методов ввода/вывода);
- события от аппаратных таймеров, всевозможных "кнопок от пульта" и др., названных внешними прерываниями.

Возможны, конечно, и другие варианты разделения на группы прерываний, но, так или иначе, они будут отражать общее понимание прерываний.
Механизмы прерываний крайне важны для быстрого и эффективного реагирования на события. И вот почему.
Разнообразный набор аппаратных и программных средств требует управления ими. Всякое управление начинается с определения состояния (работоспособности, в первую очередь) того или иного устройства, возможностей проведения тех или иных операций, и т.д.

В этой связи существуют два принципиальных подхода к операциям взаимодействия: синхронный (т.е. со стороны “интересанта” в устройстве или программе), или асинхронный, который исходит, как правило, со стороны исполнителя функции.
Так, всякое устройство может быть опрошено программой, исполняемой на процессоре, или микропрограммой. В таком случае говорим о синхронном способе. При этом процессор (микропрограмма) или программа ждет отклика с возможными данными о состоянии.
А вот в случае события на устройстве (к примеру, начало операции, событие в процессе работы, завершение работы, пр.) сигнал о событии может поступать через прерывание к процессору асинхронно, т.е. без намеренных действий по ожиданию события.
На Рис. 4 представлена общая схема реагирования процессора на прерывания. Так, в оперативной памяти по заранее оговоренным адресам размещаются адреса – точки входа в программы, которые будут обрабатывать прерывания (отдельной категорией являются прерывания, которые обрабатываются с помощью запуска микропрограмм, но в данном случае они не принципиальны). Обычно, эти позиции заполняются программами первоначальной загрузки операционных систем, а после могут быть изменены загруженным ядром ОС. Так или иначе, эти адреса указывают на секции ядра ОС, предназначенные, как минимум, для первоначальной обработки прерываний.
В случае возникновения сигнала от соответствующего триггера прерывания процессором, согласно типу прерывания, текущее значение IP сохраняется в оперативной памяти (становится значением “старого” IP), а из соответствующего прерыванию адреса в перечне “новых” в IP загружается адрес следующей выполнимой команды программной секции ядра, например, согласно схеме на Рис.4. Процессор на следующем такте, как уже говорилось, “слепо” продолжает свою работу с указанной команды. Обработчик прерывания, стартовав, имеет доступ к старому адресу, значениям регистров, которые могут послужить для корректного возврата к ранее работающей программе после обработки прерывания. Это может произойти спустя время так, как установлено в подсистеме управления задачами (после обработки прерывания менеджер задач может передать управление другому приложению, а не тому, в процессе работы которого было вызвано прерывание), через восстановление регистров, в том числе, IP на сохраненный в момент прерывания “старый”. До момента восстановления прерванной программы может произойти множество событий.
Прерывания всегда обрабатываются управляющей системой (это не всегда ОС), однако определенные типы прерываний из обработчиков могут прямо передаваться прикладным программам, которые ожидают эти прерывания. Причем, без каких-либо особых промежуточных действий.

*) **SVC- прерывания** – SuperVisor Call - команды процессора с операндом-числом, указывающим на одну из программ супервизора (ядра). Вызвав прерывание, соответствующая программная секция супервизора начинает свое выполнение.
Внешние прерывания – например, на PC – кнопки Reset и Shut down (перезагрузка, включение/выключение); на панелях компьютеров IBM – кнопки Stop/Start процессора, кнопки записи в оперативную память и чтения из нее, и подобное.
Особое место в ряду внешних прерываний занимают аппаратные таймеры, которые представляют собой регистры с возможностями записи в них числового значения. Инициировав начало работы, из двоичного числа в регистре будет вычитаться по единице через определенный квант времени до возникновения нуля в качестве значения. Обнуление таймера вызовет аппаратное прерывание, после обработки которого секцией ядра в ОС информация о событии передается ожидающей это событие программе: либо одной из служб операционной системы, либо пользовательской программе. По сути, таймеры – средства организации событий, зависящих от времени. В частности, таймаутов.
В данном случает речь шла о декрементных таймерах, основанных на убывании значений. Существуют инкрементные таймеры, например, в PC, и которые с определенной частотой приплюсовывают к текущему значению единичку.
Наличие таймеров (по крайней мере, одного), вызывающих аппаратные прерывания, в архитектуре процессоров играет значительную роль как для операционных систем (многозадачные системы с квантованием времени, управление вводом/выводом, пр.), так и для

приложений в частях, обеспечивающих собственное управление, обычно, вводом/выводом информации, при ожидании событий.
Помимо IBM аппаратные таймеры присутствуют в архитектуре процессоров SPARK, в классе DSP-процессоров (по крайней мере, четыре), др. В то же время, пойдя по пути упрощения и удешевления компания Intel в архитектуру своих процессоров x86 таймеры, кроме единственного, называемого системным, и содержащим астрономическое время, начиная с определенной даты, не включила, что значительно усложняет создание и отслеживание событий, опирающихся на время.
С целью как-то компенсировать их отсутствие компания Microsoft в своих операционных системах создала механизм виртуальных таймеров, которые крайне неточны (обычно, это считывание системного таймера на начальный момент, и считывание таймера в цикле, пока не наступит нужное количество миллисекунд в разнице между текущим значением и начальным; очень плохой способ ожидания, занимающий процессор).
По той же причине (не единственной) для этих процессоров крайне ограничены возможности организации мультизадачности. Но об этом позже в соответствующем разделе о первоосновах разработки многозадачных систем.

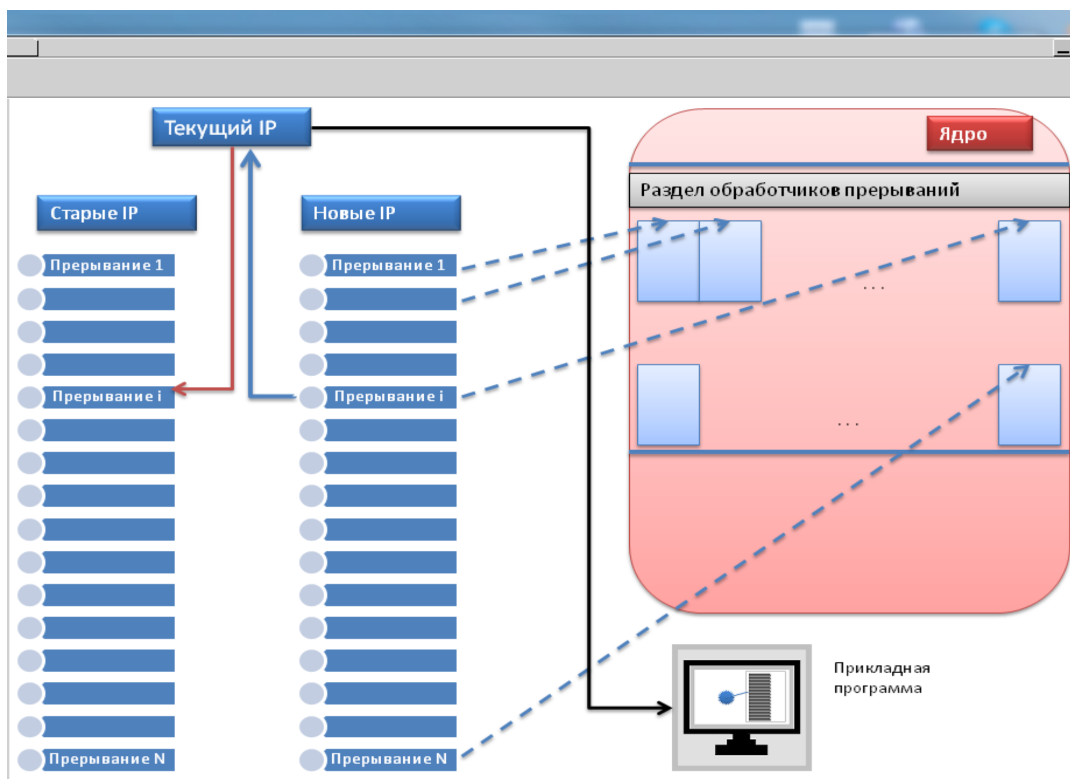


Рис. 4. Схема срабатывания прерываний.

Естественно, возможно возникновение прерываний того же типа или других. При этом, по прерываниям, как правило, не создается очередей, хотя это не исключается. В связи с этим программы обработки прерываний могут блокировать и разблокировать как все прерывания, так и отдельные их группы.
Часто “накат” прерываний просто сбрасывается уже аппаратной частью.
Как правило, с целями низкоуровневого регулирования возникновений прерываний и последовательности их обработки, разные типы прерываний могут обладать различными приоритетами по степени важности. Так, в IBM наиболее приоритетными (в целом, это остается актуальным и для других архитектур и систем) были прерывания от схем контроля процессора, возникновение которых блокировало, практически, все прочие. Сам процессор обладал средствами тестирования и выявления причин аппаратных сбоев.

Обычно, более приоритетными блокировались менее приоритетные типы прерываний, в том числе, и свой класс. При этом, здесь есть варианты решений. В частности, использованием механизма многоядерной/многопроцессорной многозадачности при обязательной реентерабельности программ обработки (эти понятия будут раскрыты позже).
На Рис. 5 отображено блокирование программных прерываний на момент обработки (колонка 3 в секциях обработчиков) на имитаторе прерываний.
ОС Windows, UNIX и пр. UNIX-подобные обладают средствами формирования событий (не только прерываний, но и их, в том числе) в очередях. На Рис. 5 в Windows очереди событий (как правило, не меньше 64К памяти) создаются подсистемой управления задачами. Для UNIX-подобных очереди создаются и управляются с помощью специальных функций.

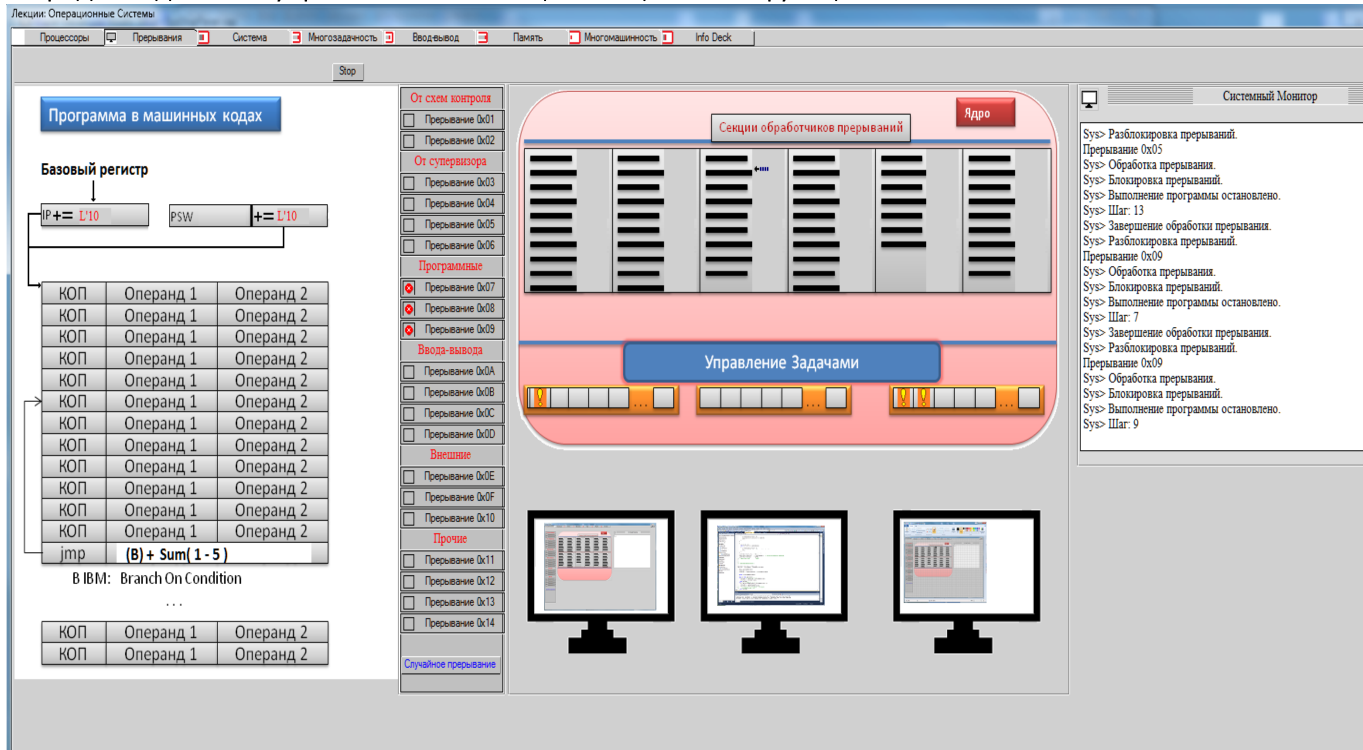


Рис. 5. Прерывания и секции их обработки. Блокирование прерываний (см. фрагмент “Программные”) и обработка в секции обработчиков (см. стрелку в “Секции обработчиков” – исполнение секции на CPU). Очереди событий приложений в разделе “Управления Задачами”.

Во всех современных операционных системах программы работы с аппаратной частью вычислителя (обработчики прерываний, в первую очередь), программы, реализующие основные схемы управления ресурсами вычислителей и программами, сгруппированы в т.н. ядре системы. Однако в определенных случаях отклик на некоторые прерывания может происходить и микропрограммами BIOS, т.е. прерывание могут вызывать функции (секции) BIOS.
Ядро обладает собственными данными, в частности, блоками памяти, описывающими и представляющими, в первую очередь, аппаратные ресурсы (часто называются UCB – Unit Control Block; в отдельных системах - Device Control Block). В UCB указываются типы устройств, конкретные модели, рабочие параметры, производители и даты выпуска, и пр.
При этом, точного определения ядер систем нет, т.к. они по своим функциям могут варьироваться от системы к системе, придерживаясь общей концепции.
Практически все модули ядра и многие из ОС в целом работают с привилегиями: имеют доступ и делают то, что запрещено прикладным программам. А именно: использовать машинные команды, которые непосредственно затрагивают ресурсы процессора и вычислителя в целом. Как правило, это регистры, кэш-память, области определения обработчиков прерываний, физическая память, взаимодействие с устройствами ввода-вывода (подпроцессоры, контроллеры, подобное). Машинные команды, которые обеспечивают перечисленное, называются и являются привилегированными. Возможность их выполнения поддерживается аппаратно. Попытка же

выполнить такую команду из прикладной программы, созданной на языке ассемблера или на языке высокого уровня, но имеющей т.н. inline-вставки на Ассемблере, приводит к аварийному прерыванию.

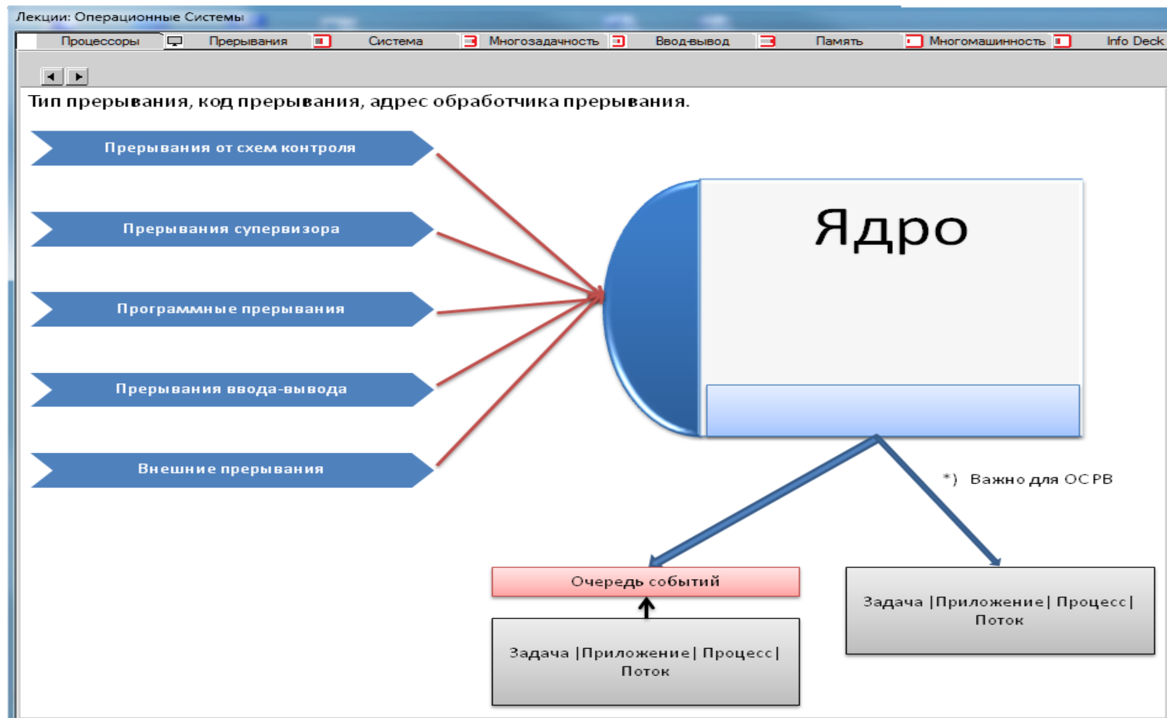


Рис. 6. Ядро и прерывания с прямой передачей ожидающему приложению или в очередь событий.

На Рис.6 показано как соответствующая секция ядра принимает прерывание и после незначительной обработки (фиксация факта прерывания, анализ и назначение, если прерывание относится к конкретному приложению) размещает информацию о событии либо в очередь событий приложения, либо прямо вызывает функцию приложения, назначенного для обработки данного прерывания. Последнее используется, в основном, в системах реального времени (ОС РВ) и что связано с максимально быстрой реакцией на связанных с вычислителем устройствах (используется в технологических цепочках в промышленности, в БИУС-ах систем вооружений и др.).

1.4. Кэш-память процессоров.

К современным новациям можно отнести появление в процессорах кэшей памяти разных уровней (разработчики сами определяют функциональную нагрузку каждого уровня). И связано это с двумя основными причинами: скоростью взаимодействия с оперативной памятью, и удобством и скоростью анализа исполняемого кода программ в реальном времени.
Кэш–память – быстрая внутренняя память процессора, предназначенная, в первую очередь, для минимизации обращений к оперативной памяти. Загрузка в регистры процессора и выгрузки из них осуществляется в размерах слова выборки, следовательно, количество обращений к ОП, к примеру, для чтения равно объему перекачиваемого, поделенному на длину слова, в то время, как одномоментное перекачивание в кэш (конечно, начиная с определенного объема) много быстрее, чем многократно в размере слова (возможно, с избыточной информацией, в размере кэша, измеряемого мегабайтами). Тем самым кэши способствуют быстрой выборке и разбору каждой команды из кэша, быстрому доступу к данным.
Помимо быстродействия выборок из кэш-памяти возникла возможность анализировать фрагмент текущего кода прикладной программы, расположенного в кэше, в процессе вычислений. В частности, анализ может осуществляться на возможность распараллеливания фрагмента

программы в контексте различных подходов в концепции динамического распараллеливания в теории параллельных вычислений (в частности, в суперскалярных процессорах).
В DSP-процессорах через кэш осуществляется распределение команд согласно их типу *) по соответствующим подпроцессорам.
Анализ исполняемого кода программ осуществляется микропрограммами из BIOS.
Если в программном участке в кэш-памяти осуществлен выход за размеры фрагмента (например, командой перехода или обращением к данным, расположенным вовне), то происходит изменение содержимого кэша подкачкой из ОП так, чтобы указываемые адреса были в кэше.

*) В DSP-процессорах команды разделяются на аддитивные, мультипликативные, команды сравнения и пр. Им соответствуют специализированные подпроцессоры.

Глава 2. Операционные системы, причины их создания и разновидности.

Познакомившись с общими принципами и базовой комплектацией вычислителей мы можем сказать и о причинах появления операционных систем.
Значительная часть организации вычислений приходится на выполнение реакции на события, на операции по вводу-выводу, по запросам на выбор элементов динамической памяти, на загрузку и выполнение программ, на другое (к примеру, на организацию межпроцессорного и межмашинного взаимодействия).
В этой связи понуждать разработчиков и программистов каждый раз придумывать и разрабатывать способы решений всего перечисленного, или “тащить” вслед за программой большое количество служебных и, в целом, рутинных программных модулей – довольно дорогое, и, главное, малополезное удовольствие. Особенно, если учесть, что программисты ошибаются в силу разных причин. Именно поэтому все в той же компании IBM с разработкой архитектуры продумывалась и концепция операционной системы для моделей линейки IBM-360. Надо подчеркнуть эту важную особенность для разработок тех времен: архитектура разрабатывалась с учетом принципов управления вычислителями, схемам решения задач на них. Причиной, по которой мы начинаем столь издалека, является тот факт, что до сих пор такой подход остается единственно верным. Не считая того факта, что решения, заложенные в архитектуру вычислителей IBM так же остаются, практически, образцовыми.
Важнейшим признаком поддержки многозадачности со стороны архитектуры вычислителя являются возможности не давать приложениям захватывать центральный процессор на неопределенно длительное время. Простейшим примером может послужить зацикливание участка программы без использования функций обращения к службам ОС, в которых могли бы присутствовать средства контроля и прерывания приложения. Что, по существу, блокирует операционную систему от осуществления задач перераспределения ресурсов между приложениями.
В этой связи, помимо концепции прерываний, разрабатывались подпроцессоры ввода/вывода с возможностями взаимодействия по записи/чтению непосредственно с оперативной памятью, не задействуя для этих целей центральный процессор. Что подразумевало еще одну основу для создания многозадачности. Здесь можно провести сравнение с процессорами Intel, в которых чтение/запись, по существу, осуществляет CPU.
Субпроцессоры IBM называются каналами ввода/вывода, обладают собственным машинным языком, состоящим из небольшого, но достаточного набора команд с фиксированной длиной, и управлением, представителем которого является регистр CSW (аналог PSW, Channel Status Word), в котором отражалось текущее состояние подсистемы В/В, регулировался доступ к оперативной памяти, др.
Наиболее важное – во время работы каналов процессор остается свободным и переводится командой “Стоп” в режим ожидания. Именно эта возможность стала первопричиной для создания управления многими программами, одновременно загруженными в ОП, переключая процессор с

программы на программу на время ожидания завершения В/В (при синхронном режиме, об асинхронном режиме см. соответствующий раздел в данном тексте).
В этой же связи, из-за разработки заведомо многозадачного режима было создано т.н. “ключевание” памяти, т.е. вся оперативная память состояла из одинаковых блоков с фиксированной длиной, имеющих электронные биты хранения ключа памяти.
В начальный момент сразу после загрузки операционной системы вся память имеет ключ 0.
По мере загрузки приложений каждое из них получало фрагменты ОП, состоящие из указанных блоков с единым ненулевым ключом, приписанным приложению. После завершения приложения память, выделенная для него, освобождается с установкой нулевого ключа.
Ключи памяти стали первоосновой для регулирования разграничения и доступа к памяти: попытки обращения из программы к памяти с чужим ключом вызывает аппаратное прерывание (на аппаратном уровне сопоставлялись ключ приложения, находящийся в PSW, и ключ адресуемой памяти), т.е. не требовало нетривиальной виртуализации адресного пространства и доступа к нему.
Таким образом, идеологи IBM изначально нацеливались на создание многозадачного вычислителя и закладывали в него необходимое аппаратное устройство. Как изначально ими заявлялось, они создали “идеальную” архитектуру для вычислителей того поколения и класса, и позволяющих одновременно работать с сотнями разноудаленных пользователей (тогда: 7 каналов по 256 устройств на каждом). Известны эти вычислители как “мэинфреймы” (mainframe).
Многие компоненты и решения были позже заимствованы другими разработчиками и компаниями.
Отметим, что данная архитектура сохраняется до сих пор, практически, без изменений.
Современные компьютеры IBM на базе IBM-390 обладают только увеличенными количественными характеристиками (в части, размеры регистров увеличены вдвое, сохраняя при этом сущностную составляющую).
Правда, все перечисленное не гарантирует от проблемы захвата процессора одним приложением.
Здесь важна роль администратора системы, предполагающего такую возможность. Решение, как правило, одно: установка аппаратного таймера временем, отводящимся работе приложения, что вызовет прерывание и отскок к ядру ОС с последующей возможностью переключения на другие задачи.
Подчеркнем одно из правил: чем меньше архитектура вычислителя и его ОС согласуются с задачами, которые придется на нем решать, тем больше ресурсов (время, память, объемы применяемых функций и пр.) мы будем затрачивать на их реализацию.
Координация выполнения программ, в том числе, в вопросах их загрузки и размещения в ОП, в совместном использовании внутренних и внешних ресурсов вычислителей, обеспечение единообразных способов их использования, контроль за выполнением программ, обеспечение работоспособности, устойчивости вычислителей и периферии – основные задачи операционных систем, эффективность решение которых подчеркивает достоинства и недостатки ОС.
Операционные системы, по своей теоретической предрасположенности, являются т.н. мониторами в ряду трех разновидностей в управлении ресурсами: семафоры, мониторы, сентинелы в классификации, данной в [1] *). Здесь вполне уместна аналогия с прорабом и заведующим инструментальным складом в одном лице: первый распоряжается проведением работ, второй – выдачей и контролем за различными выданными инструментами - ресурсами.
При этом, отдельные компоненты операционных систем могут использовать все три способа управления.

*) Сентинел – вычислитель, обслуживающий пользователей и программы выполнением по запросам от них функций и процедур (известны, как серверы разной направленности). Подобный подход не требует от программ дополнительной алгоритмизации и предоставления ресурсов для реализации требуемых алгоритмов. Так, к сентинелам можно отнести серверы баз данных, работающих по принципу “запрос-ответ (результат по запросу в виде данных)” и не требующих от запрашивающего программирования поиска, ресурсов для его реализации и компоновки результатов.

2.1. Логика многозадачности.

Как архитектура, так и созданные для нее операционные системы мэинфреймов IBM изначально были ориентированы на многозадачную работу вычислительной установки (в отличие от предыдущих архитектур). Это выражалось во всех составляющих компонентах как вычислителя, так и ОС: в машинных командах процессора, в аппаратной поддержке оперативной памяти с разделением ее ключами памяти (и приложений), во всеобъемлющем подходе к периферийным устройствам и вводу-выводу в целом.
Мы разделим подходы к созданию операционных систем по архитектурным особенностям:

1. Архитектуры с независимым от центрального процессора (CPU) вводом-выводом посредством комплекса субпроцессоров непосредственно в оперативную память или из нее (IBM).
2. Архитектуры с аппаратными таймерами с возможностями организации квантования времени (процессоры IBM, SPARC, DSP-процессоры, обладающие по 4 и более таймерами, некоторые другие).
3. Архитектуры с независимым от центрального процессора (посредством комплекса субпроцессоров) вводом-выводом и с разделенными динамической и статической памятью (типа DSP-процессоров). В общем случае: с раздельной памятью для приложений и ввода-вывода.
4. Многопроцессорные системы (множество коммутируемых CPU).
5. Многомашинные системы (комплекс вычислительных машин, объединенных средствами связи, в том числе, вычислительными сетями).

Особое место в данном перечне занимает т.н. “ корпоративная многозадачность”, которая основана на довольно “бедных” архитектурных ресурсах, не имеющих ни средств отделения ввода/вывода от CPU, ни аппаратных таймеров с прерываниями. Потому в перечне мы обозначим как “-1” пункт с архитектурой Intel x86.
Существенную роль при создании многозадачной операционной системы для IBM играли процессы с вводом-выводом посредством
- выделения специальных подпроцессоров со своим командным рядом и возможностью прямых чтения-записи в ОП без участия CPU;
- разделение памяти электронными ключами, что позволяло загружать множество задач, т.к. ключи позволяли контролировать доступ к оперативной памяти (адресацию) на аппаратном уровне;
- адекватной системой прерываний, в том числе, от таймеров;
- возможностью останавливать центральный процессор, если нет на данный момент программ к исполнению, или переключать его на другую задачу;
- многопроцессорность и более поздняя многоядерность процессоров.

Перечисленное стало предпосылками к “правильной” многозадачности в отличие от более поздних систем, созданных на более простых архитектурах (для первых версий ОС Windows, OS-2 для архитектуры Intel – это, скорее, псевдомногозадачность; ситуация несколько улучшилась с появлением многоядерности процессоров, когда выполнение задач ввода/вывода можно возложить на выделенное ядро(а)).
Важнейшей компонентой, которую изначально закладывали в архитектуру IBM-360, была возможность устанавливать электронные ключи (4 бита) в четырехкилобайтные блоки памяти. Т.е. заведомо предусматривалась возможность разделения памяти по 16-ти группам блоков, что, в свою очередь, позволяло размещать самостоятельные и изолированные друг от друга ключами памяти приложения, их коды программ и наборы динамически запрошенных полей памяти для данных. Допускалось до 16 включительно различных приложений: с 0-ыми ключами для операционной системы, с ненулевыми – для приложений. Данный вариант назывался системой с фиксированным числом задач. Несколько позже появился вариант с переменным числом

прикладных задач, еще позже – системы виртуальных страниц и системы виртуальных машин. При этом все они опирались на замечательное свойство электронной защиты памяти, препятствующей доступу одного приложения к памяти другого. Это качество IBM существенно отличается от систем с виртуальной адресацией (последняя относительно медленнее, чем система с аппаратной поддержкой).
Одной из самых значимых подсистем в реализации многозадачности является организация ввода-вывода информации и соответствующая служба ОС.
Проблемы заключаются в том, что скорость работы процессора и скорости взаимодействий с устройствами периферии вычислителей существенно отличались и отличаются до сих пор. Иногда на несколько порядков, от микросекунд и наносекунд для команд процессоров и миллисекунд, секунд и минут для внешних устройств. Выполняемая на CPU программа, запросив чтение с устройства или запись на устройство, будет

- либо приостановлена до завершения операции ввода-вывода;
- либо, обозначив потребность в вводе-выводе, продолжит работу до возникновения события (в том числе, прерывания), согласуясь с логикой программы, и связанного с ожиданием завершения вызванной программой операции ввода-вывода; ожидание может и не наступить, если к данному моменту операция В/В была завершена.

В первом случае действие называется синхронным, во втором – асинхронным. Проблема синхронизации с результатом В/В может наступить, если операция В/В была не завершена к нужному моменту (как правило, переводом программы в режим ожидания: циклом опроса (плохой способ, но, часто, вынужденный), например, установки соответствующего флага в переменной, либо передачей управления ОС функции ожидания до момента возвращения ею управления приложению, либо, при наличии соответствующих средств, перевод приложения в специальный режим ожидания события по вводу-выводу, и пр.).

**) В целом, синхронными называются вызовы процедур из программ-инициаторов (в частности, опросы устройств или других программ), в то время как асинхронные вызовы предполагают создание запросов без остановки последующих операций в программах или устройствах.
Если синхронные вызовы довольно просты и связаны с остановкой - ожиданием результата вызова, то асинхронные вызовы потребуют дополнительных действий – программирования реакции на событие, информирующем о завершении некоего вызова или перехода в режим ожидания события завершения, если событие не наступило (т.е. запрос не был выполнен) к моменту актуальности получения результатов. Заметим, что программа может получить информацию (возможно, через прерывание от ОС, или выставлением информационного флага) об асинхронном событии.

Рис. 7. Синхронные и асинхронные вызовы.

Определенное решение проблемы независимого от CPU ввода/вывода было предложено в архитектуре процессоров цифровой обработки сигналов DSP, в которых присутствует два вида памяти: статическая и динамическая. Процессорами управления В/В потоки информации считываются в или выводятся из статической независимо от CPU, в то время, как сам CPU может выбирать или заносить информацию из статической в динамическую для последующей обработки, или выгружать в статическую для вывода.
Таймеры. Наличие аппаратных таймеров позволило создавать класс систем с квантованием времени, т.е., в зависимости от очередности и приоритетности, загруженным приложениям назначаются кванты времени, в течение которых процессор может выполнять фрагмент той или иной задачи. По обнулению таймера происходит аппаратное прерывание, которое перехватывает ядро, а следом супервизором задач происходит переключение на другую задачу, если таковая имеется. Т.к. кванты времени, как правило, оставались небольшими, то возникала иллюзия одновременной работы программ.
Наконец, многопроцессорность, которая появилась уже в первых моделях машин, обеспечивалась шиной “процессор-процессор”, что позволяло соединять напрямую, по крайней мере, два CPU. Такие соединения обеспечивались командами ассемблера для взаимодействия процессоров. Многопроцессорность, в целом, к тому моменту была не уникальна, т.к. уже существовали вычислители со многими процессорами и с различными архитектурными решениями. Но это – тема отдельного курса. Сейчас же отметим, что многопроцессорность с разделенной памятью (каждый процессор обладает собственной физической памятью; к таким архитектурам можно отнести процессоры Motorola 68000 и выше, имеющих локальную шину в блоке из процессора, оперативной памяти и средств ввода/вывода; блоки могут быть размещены на внешней шине для межблочного взаимодействия; такая архитектура была поддержана OS-9) снимает много проблем по разделению и выполнению приложений.
Многопроцессорность с общей памятью (такая, которая реализована в современных многоядерных системах) решает только проблему одновременного выполнения множества задач, в то время как задачи разделения и защиты памяти приложений сохраняются. В том числе, и на аппаратном уровне, т.к. должна быть решена (и решается) задача синхронизации одновременных обращений к памяти различными ядрами.
Так или иначе, многопроцессорность/многоядерность обеспечивают одновременное (реальнопараллельное) исполнение многих задач (см. Рис.).
То же самое относится и к многомашинным вычислительным установкам. Значимым отличием является тот факт, что межмашинное взаимодействие значительно медленнее, чем межпроцессорное, и не только из-за средств связи между машинами, но и потому, что посредниками взаимодействий являются операционные системы.
Однако, с точки зрения логики организации одновременно выполняемых процессов особых различий между многопроцессорностью и многомашинностью нет.
Таким образом, мы перечислили основные архитектурные особенности вычислителей, которые стали основой для полноценной многозадачности операционных систем.
К сожалению, в силу больших упрощений процессоров x86 разработчикам ОС для них пришлось приложить немало усилий, чтобы сформулировать принципы решений по многозадачности. В частности, возникла идея “кооперативной” многозадачности, свойства которой оставались и остаются достаточно сомнительными. Но об этом в отдельном разделе.

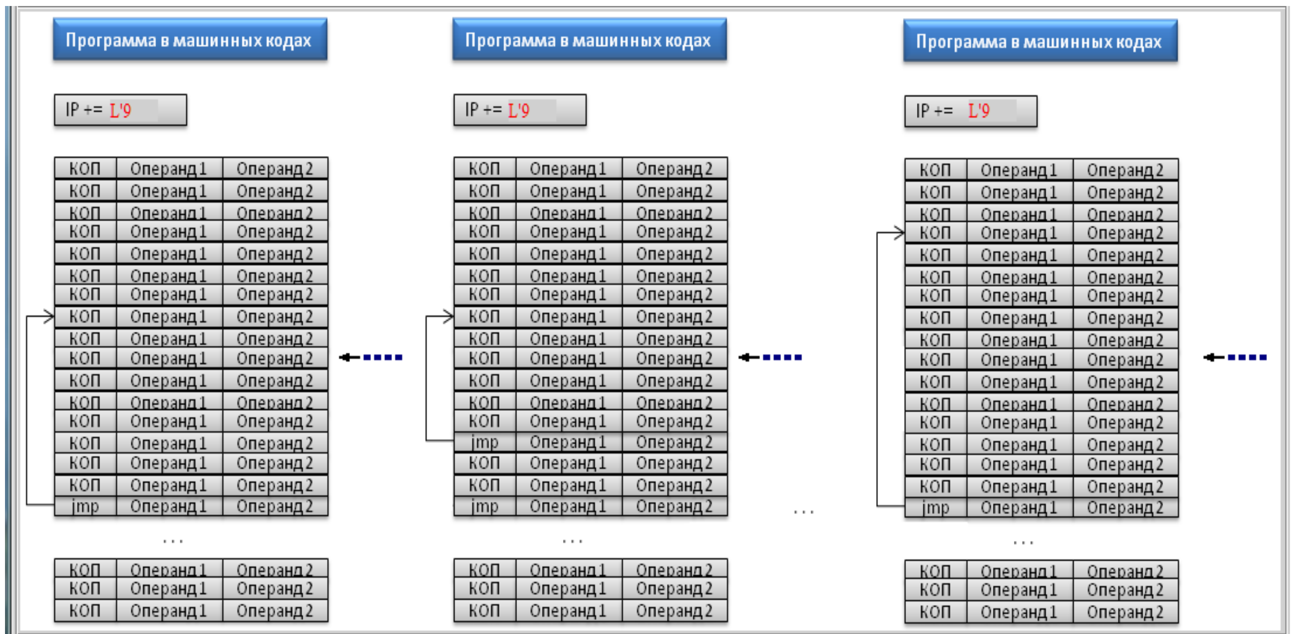


Рис. 8. Реально-параллельное исполнение программ в многоядерной/многопроцессорной архитектуре (IP и стрелки указывают на очередную машинную команду для каждого ядра/процессора).

Нужно отметить, что процессоры и ядра могут выполнять одинаковые потоки задач не с одинаковой скоростью, т.к. каждый из ядер/процессоров может быть загружен в разной степени и другими задачами.
С целью беспрепятственного, без задержек, исполнения собственных задач ОС (в особенности – обработки прерываний) в момент загрузки ОС могут резервировать под последующие собственные нужды, по крайней мере, одно ядро или процессор. Так же, как и память, разделяя память ОС и незанятую память, которая может быть отдана под приложения.

***) Многие СУБД при установке на серверах под операционными системами требуют указания количество ядер/процессоров, которые могут использоваться процедурами баз данных. Так же фиксируется и объем оперативной памяти под нужды СУБД, т.к. подобные системы обладают собственными средствами управления ресурсами. По сути, это система в системе.

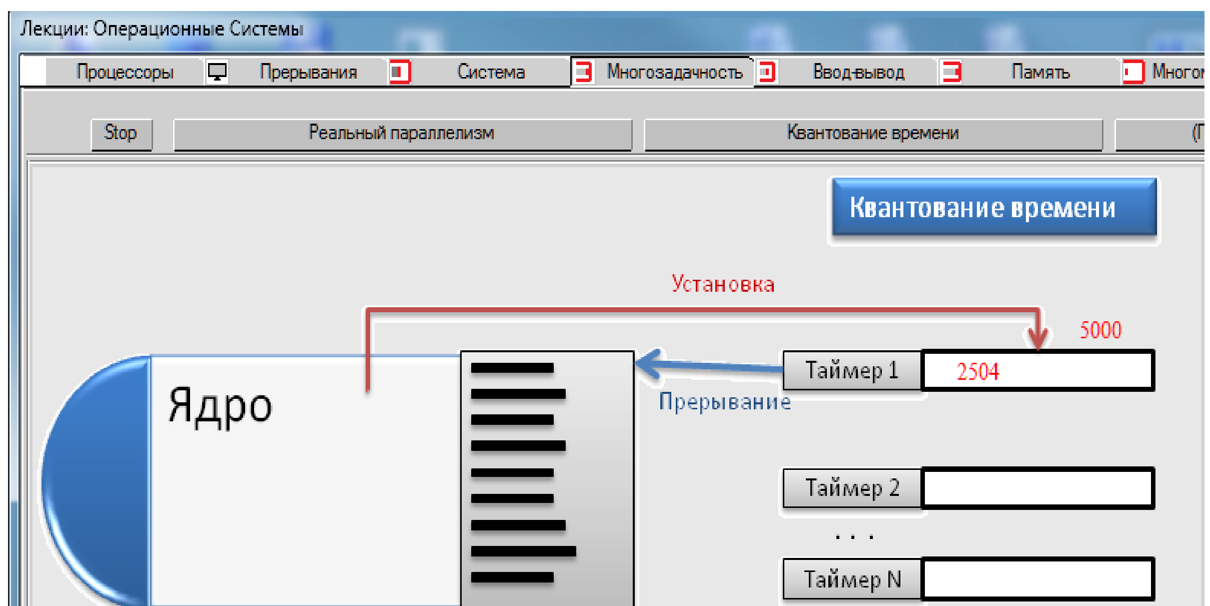


Рис. 9. Исполнение программ с квантованием времени (имитатор). В данном случае установлен квант в 5000 миллисекунд, текущее значение таймера 2504 мс.

2.2. Службы операционных систем.

Обычно весь функционал операционных систем содержательно разделяется на статичный и потенциально динамический. Ключевую роль здесь играют возможные изменения, вариативность тех или иных компонент ОС. Так, обработчики прерываний, базовая схема управления памятью, управления задачами вряд ли будут изменяться в рамках архитектуры конкретной операционной системы. А вот в надстроечной части тех же компонент (допустим, реализации “сборки мусора” в ОП после завершения приложений, организации оверлеев и swap-функций), некоторых тестирований ресурсов, и пр. можно выносить вовне ядра. Так сделано и для решения задач управления вводом/выводом для различных устройств, особенно учитывая их многообразие, что требует различных компоновок и перекомпоновок программных средств. На Рис. 7 представлены основные службы ОС.
Как правило, возникновение событий (запросы и освобождения ресурсов) из ядра ОС транслируется к соответствующей службе. В частности, к службе управления памятью могут возникнуть обращения, когда, как указывалось, возникнет необходимость “сборки мусора”, когда отдельные фрагменты памяти, конечно же, по ошибке, остаются неудаленными (к примеру, по забывчивости программиста). Это явление известно как “утечка памяти” (memory leak). Иная форма контроля памяти – это сбор и эвристические оценки по ее использованию теми или иными приложениями. К примеру, некоторые версии ОС Windows могут не сразу физически освобождать оперативную память, а неявно ее резервировать для повторного старта того или иного приложения. На основе эвристик, конечно. Этими вторичными действиями занимается соответствующая служба ОС. При этом зарезервированная память включается в размеры свободной физической памяти.
Куда сложнее и разнообразнее действия по управлению вводом/выводом. Как указывалось, в силу разнообразия устройств, адресации в них, способов чтения/записи и пр. Подробнее они будут продемонстрированы в соответствующем разделе данного курса. Пока же остановимся на факте существования в ОС соответствующей службы.
Еще одной службой ОС является поддержка системных команд ОС. Как и всякая установка, механизм, решающий задачи, требуется и система, хотя бы элементарная, управления данной установкой.
Это решается службой поддержки взаимодействия с пользователями (не только с людьми, но и с программами). В основном, система команд ОС ориентирована на реконфигурирование периферии, на создание и удаление наборов данных (управление файловыми системами), отображение информации о них и пр.
К таким операциям с соответствующими командами можно отнести (основное)

- монтирование и демонтирование устройств, т.е. подключение и регистрация в ОС новых устройств и их отключение; в качестве примера можно привести подключение внешних устройств флеш-памяти через порт USB, когда через прерывание ОС инициирует (с возможной генерацией командной строки или прямым вызовом программы) службу В/В произвести монтирование нового устройства; некоторые системы могут блокировать автоматическое подключение устройства, и тогда это придется делать вручную с помощью набора соответствующей команды; отключение происходит аналогичным способов, прямо вытащив устройство из разъема (опасный способ, т.к. не гарантирует сохранение последней записываемой на него информации; подробнее в разделе, посвященному вводу/выводу и вопросам буферизации записи), что, конечно, тоже может вызвать прерывание и инициировать процесс демонтирования, а может и не вызвать, и тогда при попытке работы с изъятым устройством оно будет установлено в режим присутствующего в системе, но неработающего;
- запуск и терминирование необходимых программ;
- регистрация и анализ присутствия пользователей в системе;
- создание и удаление наборов данных (директории/папки, файлы).

Широко представлены средства просмотра информации:

- о директориях и отдельных файлах (время/дата создания, время/дата последнего обновления, о пользователях, использующих данные, их привилегиях, и пр.);

- о работающих на данный момент приложениях;
- об объемах занятого и свободного пространства ОП;
- о загрузке процессора;
- другое, зависящее от реализации ОС и ее служб.

Для многопроцессорных и многомашинных устройств объемы функций управления и информации, безусловно, значительно расширяется: требуется знание о каждой отдельной машине или процессоре комплекса, о группах машин, их общих и раздельных ресурсах, средствах взаимодействия, и пр.
Доступ к той или иной информации, как и к ресурсам вычислительной установки, должен регулироваться и регистрироваться.
Так или иначе, любая вычислительная установка должна иметь режим администратора для человека, обладающего доступом ко всем ресурсам системы и реализующего запросы прочих пользователей.
Для целей формирования необходимой вычислительной среды операционные системы имплантируют командные интерпретаторы (обычно все команды и их параметры представлены в виде одной текстовой строки) и службу поддержки системного монитора (системная консоль, см. Рис. 6) для ввода команд и вывода запрошенной информации или автоматическом выводе информации о текущих событиях (см. Рис. 7).

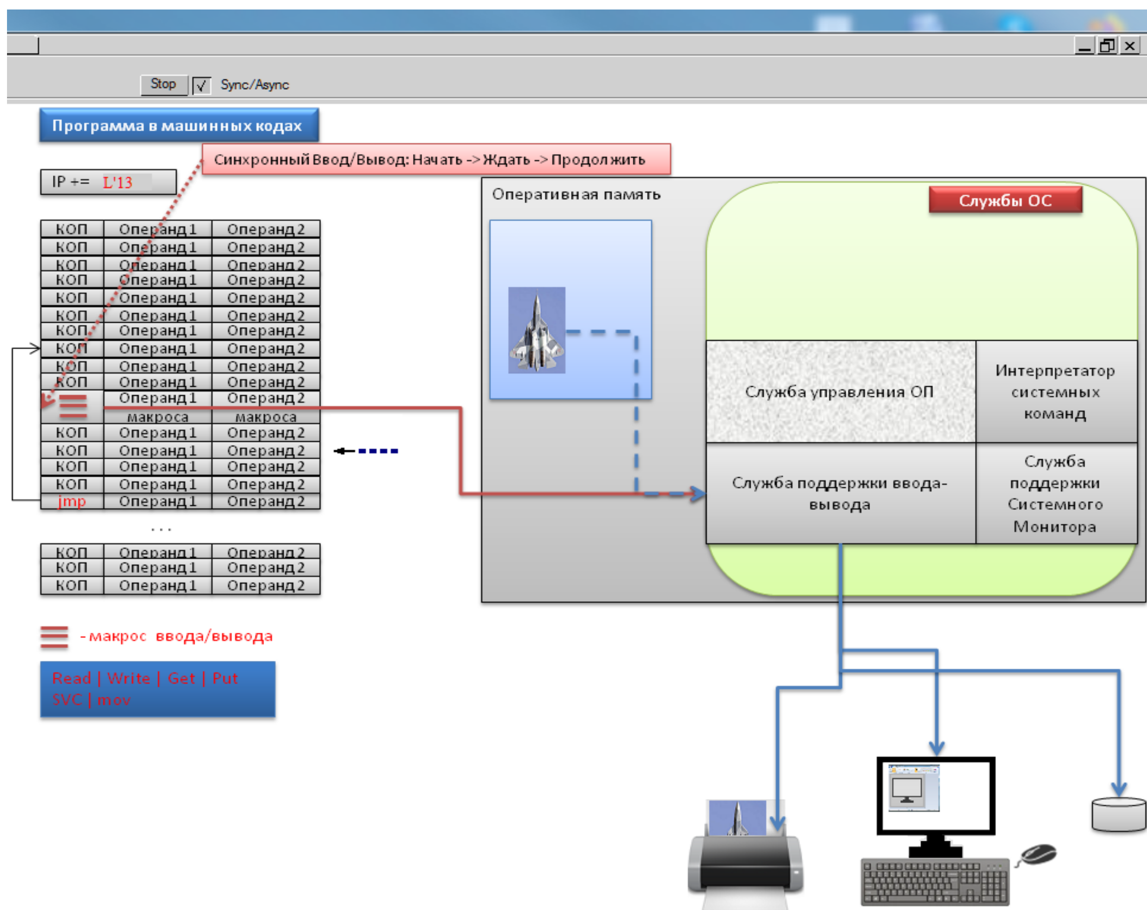


Рис. 10. Службы операционных систем.

Любая операционная система обладает Системным Монитором – программой, позволяющей своему оператору со статусом администратора взаимодействовать с вычислительным

комплексом. Каждый оператор, помимо средств идентификации и пароля, обладает правами доступа к распоряжению и настройкам всех или части ресурсов вычислительной установки.

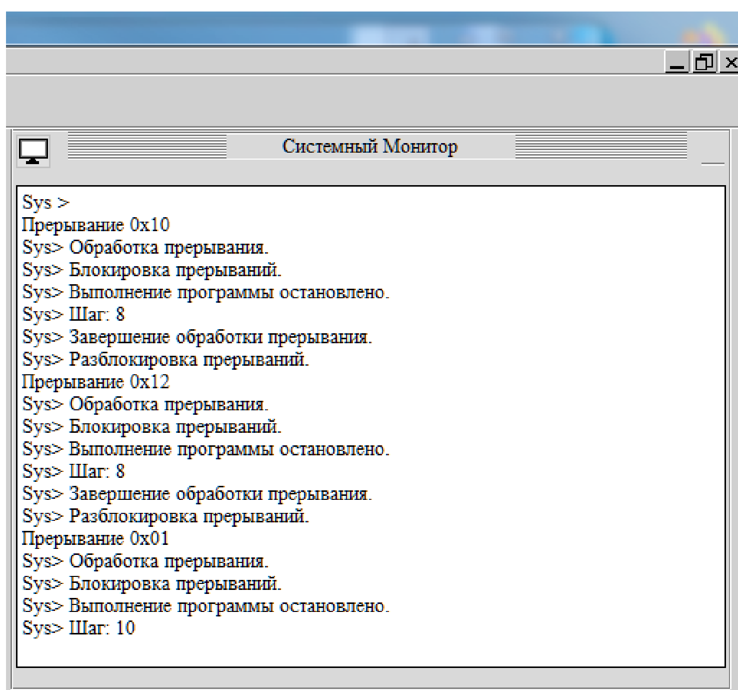


Рис. 11. “Системный монитор” (системная консоль) программы-имитатора.

2.3. Требования к операционным системам.

К операционным системам представляются определенные содержательные требования, которым должны следовать разработчики ОС.

1. Операционная система в своей работе должна занимать минимум времени (по крайней мере, разработчики ОС должны стремиться к этому).
2. Операционная система в своей работе должна занимать и/или использовать минимальное количество ресурсов: оперативную память, дисковое пространство и др.
3. Устойчивость. ОС может войти в аварийное состояние (“рухнуть”) только по причине аппаратных неполадок вычислителя (процессор, или оперативная память, или системные шины, или подобное).
4. ОС контролирует все ресурсы вычислителя и предоставляет процедурное обеспечение доступа прикладным программам к каждому ресурсу.
5. ОС не допускает изменения состояния и информации ресурса, предоставленного одному из приложений (задаче), со стороны другого приложения (задачи).
6. Функции ОС обязаны всегда возвращать значение при обращении из приложений. Аварийное завершение модулей ОС недопустимо.
7. ОС не допускает “взломов” системы и проникновения/воздействия “вирусов” на свою среду.
8. Вся управляющая и служебная информация по ресурсам и обслуживающие их процедуры находятся вне прямого доступа со стороны приложений (посторонних для ОС программ).

Есть примеры некорректных решений, не соответствующих этим требованиям. В частности, ОС Windows допускает “порчу” окон перекрытиями окнами других приложений. При том, что окна являются главными логическими элементами и подконтрольными ОС ресурсами, выделенными приложениям для организации интерфейсов. Тем самым понуждая приложения – владельцев окон “перерисовывать” их посылкой сообщений о порче.
Примерно так, как если бы одно приложение портило файл, занятый другим приложением, а ОС после этого посылало сообщение с предложением владельцу восстановить файл.

Причины таких решений понятны: создавать и поддерживать внутри ОС слепки перекрываемых окон – довольно емкое по памяти занятие. Однако факт остается фактом: ОС не поддерживает сохранность предоставленному приложению ресурсу, а оставляет эту задачу на само приложение. Тем более, что объемы ресурсов современных компьютеров неизмеримо возросли.
Еще одним нарушением канонов является унаследованный еще от UNIX способ создания блоков регистрации динамически выделенных пулов памяти – однонаправленных списков (этот способ реализован в UNIX-подобных ОС, в том числе, в Windows). При вызове средств выделения памяти (в C/C++ - malloc() и его вариации) выделенный участок памяти предваряется фрагментом, содержащим блок регистрации выделенного участка и элементом списка. Тем самым блок управления элементом памяти является размещенным в пользовательской области, что в случае ошибки может приводить к порче блока и списка учета в целом.
Именно с этим связано еще одно нарушение: следующие вызовы системных функций выделения и освобождения памяти завершаются аварийно, без возврата к вызывающей их программе. При отсутствующей диагностике поиск ошибок в приложении (разрушение содержимого элемента списка) становится крайне затруднительным и требует большой изобретательности и времени.
Отметим, что в системах IBM все блоки управления ресурсами располагаются в памяти ОС, что не дает доступа к ним со стороны приложений.
Примечательно и другое: операционные системы IBM не взламываются, вирусы не проникают и нежизнеспособны. Именно по этой причине вычислители от IBM никогда не анализировались на взламываемость и подверженность вирусам, и не упоминались в соответствующих рейтингах.

2.4. Загрузка операционных систем.

Основная работа любой вычислительного комплекса начинается с загрузки его системы управления. Мы остановимся на загрузке операционных систем для одного компьютера, хотя логика загрузки систем для более сложных вычислительных комплексов вполне согласуется с изложенной далее схемой.
Начинается загрузка ОС с выполнения микропрограмм в BIOS, которые тестируют аппаратную часть вычислителя. Следом за ними срабатывает программа BIOS по опросу устройств, заданных в качестве потенциальных носителей операционной системы. Найдя первое же в перечне носителей или по заранее установленному указанию устройства, происходит загрузка программы конкретной ОС. Если указанное устройство не соответствует принципам загрузки (отсутствие или некорректная запись загрузчика (boot-сектор, в частности)) приводит к поиску другого устройства с первоначальным загрузчиком.
В случае найденного модуль BIOS загружает в оперативную память с фиксированного места на внешнем носителе (Boot-сектор) программу первоначальной загрузки (по традиции они обозначаются аббревиатурой IPL - Initial Program Load). И не обязательно, чтобы это была программа операционной системы. IPL загрузит любую правильно организованную (с пониманием действия BIOS) программу, записанную в сектор начальной загрузки на внешнем носителе. Программа загружается в заранее определенное место оперативной памяти (возможно, начиная с нулевого адреса памяти, чаще с фиксированным смещением от начала – нулевые адреса некоторыми системами используются для размещения векторов прерываний – указателей на функции обработки прерываний), в IP формируется указатель на первую команду загруженной программы.
Такой программой может быть программа выбора операционной системы, если на компьютере установлено больше одной ОС (например, программа GRAB для UNIX-подобных).
Загрузка ОС, как правило, осуществляется в несколько этапов: загружается загрузчик ОС, который, в свою очередь, загружает ядро операционной системы, которым инициализируются векторы прерываний и прочие необходимые данные. Далее производится загрузка служб операционной системы. Для отображения процесса загрузки может быть подключен Системный монитор.

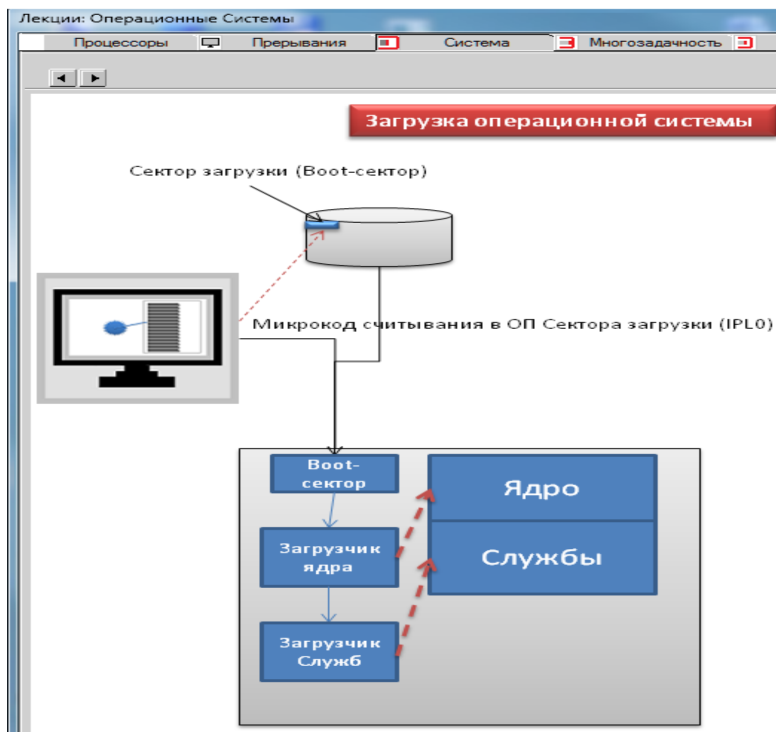


Рис. 12. Первоначальная загрузка ОС.

Одним из этапов и временем загрузки является загрузка драйверов – программ непосредственного взаимодействия с устройствами посредством контроллеров.
Перечень устройств может быть фиксированным (логично его держать в отдельном загружаемом файле) либо формируется опросом через шину о подключенных устройствах. Это – динамический случай.

2.5. Задачи и подзадачи, процессы и потоки.

Во многом благодаря компании Microsoft сформировалась “разноголосица” в терминологии, связанной с пониманием выполняемых задач на вычислительных установках.
Терминология формировалась в компании IBM на протяжении 20-ти с лишним лет, была выверенной и продуманной, но которая стала “перебиваться” с расширением персональных компьютеров и операционных оболочек и операционных систем от Microsoft, иногда запутывая понимание уже сложившихся категорий.
Поясним и упорядочим ее.
Всякое вычисление в IBM начиналось и начинается с задания операционной системе, в котором описывается среда исполнения задачи и ресурсы, которые должны быть привлечены. Такой подход позволяет операционной системе планировать (!) исполнение задачи. Именно по этой причине важнейшей компонентой ОС является планировщик заданий и задач, который оценивает текущие возможности вычислителя и занимается подготовкой среды исполнения. Это нашло отклик в современных механизмах контейнеров, супервизоров и гипервизоров.
Задания позволяют не только готовить необходимые ресурсы, но формировать их защиту от незапланированных действий, а также изолировать среду конкретного вычисления как от вторжений извне, так и проникновений вовне оболочки самого вычисления.
Зафиксировав описание среды ОС запускает задачу. Задача, в свою очередь, может запускать подзадачи. В системах от MS задаче соответствует понятие процесса, группирующего в себе потоки. Оба понятия не вполне удачны, т.к. понятие процесса пересекается с более фундаментальным понятием параллельного процесса в теории и практике параллельных вычислений, в которой всякий процесс рассматривается в группе других процессов, занятых решением одной задачи. Так же неудачен и термин “поток”, т.к. данный термин совпадает с

понятием и терминами одной из концепций параллельных вычислений: т.н. потоковых вычислений (flow, flow-data, stream). Термины теории параллельных вычислений появились значительно раньше, чем решения от MS. По существу, процесс в MS – задача в IBM, поток – параллельный процесс или подзадача.
Подзадачи (потоки) могут работать как с собственной памятью, когда распараллеливаемый поток полностью, вместе с памятью копируется в новую, выделенную ОС память. Так запущенные потоки называются тяжеловесными.
Эта логика наследована от UNIX-подобных систем. При этом очередь событий остается общей (напомню, что в UNIXах очередь событий может быть сформирована с помощью специальных функций). Однако, есть возможность потребовать создание для потока собственной очереди событий.
В то же время, ОС предоставляет возможность оставить память общей для потоков. Такие потоки называются легковесными, т.к. формируются проще и быстрее, без дублирования полей памяти вызвавшего потока-отца.
Каждый поток вызывается Стартером задач ОС, для каждого потока (подзадачи) формируется собственный стек вызовов, в начале которого размещается указания на точку возврата в функцию стартера из функции - точки входа в поток.

2.6. Фоновые и резидентные программы.

Есть определенный класс программ, которые могут работать в т.н. фоновом режиме. Это программы, не взаимодействующие с пользователями, их присутствие никак не отражено или отражено чисто символически каким-нибудь значком на экране или наличием файла.
Обычно такие программы создаются для отслеживания событий или выполнения каких-либо рутинных, служебных функций, допустим, для ведения и/или анализа журнала событий.
В Windows такой пользовательской программой, работающей в фоновом режиме, может считаться программа, не создающая собственного окна интерфейса.
Одновременно, практически все операционные системы, за исключением самых простых (типа MS-DOS или проще), разрешают включать в свой состав (в службы или даже ядро) т.н. резидентные программы, которые, по традиции, имеют привилегированный режим (как программы операционной системы), работают с памятью операционной системы (память с нулевыми ключами), и способные перехватывать и обрабатывать прерывания.
Такие программы имплантируются в ОС только с разрешения и в режиме администратора.
Однако, написание таких программ требует большого внимания, особенно в части реентерабельности (о реентерабельности, ее ограничениях и направленности см. в разделе “Многозадачность”). Они, безусловно, должны быть эффективными как по памяти, так и быстродействию. Они должны быть реентерабельными, если занимаются обслуживанием запросов от множества приложений.

Глава 3. Задачи, задания и управление ими.

Конечной целью вычислителей является очевидное: исполнение программ, возникающих как из внешних источников, так и являющихся внутренним ресурсом установки. В данной главе мы рассмотрим различные схемы исполнения программ и заданий на вычисления, возникающих извне от пользователей. При этом, несмотря на архитектурные и схематические особенности, всем универсальным вычислителям и их операционным системам присущи одинаковые функциональные свойства: размещение задач и заданий, выбор их на исполнение, загрузка исполнимого кода и данных, старт задач, обеспечение их устойчивой и однозначной (вне зависимости от состояния среды вычислителя) работы, размещение результатов и завершение задач.

3.1. Запуск задач и заданий.

Операционные системы по-разному относятся к запуску задач: согласно целям и архитектурной логике использования вычислителей.
Задачи могут быть представлены заданием – комплексом записей (пакетом), предписывающих набор установок в ОС и действий, которые должны принять в качестве параметров, указаний, действий, и которые следует выполнить службам ОС. Каждая ОС имеет собственные наборы предписаний (в IBM с этими целями создали целый язык управления JCL – Job Control Language). В Windows и UNIX-подобных системах предписания и описания могут группироваться в т.н. batch-файлах или скриптах оболочки (shell script).
Одновременно, задача может быть запущена после ввода командной строки для интерпретатора системных команд через системный монитор (или интерпретатор команд операционной системы). В Windows придумали механизм иконок, размещенных на “рабочем столе” (desctop) представляющих задачи и задания, которым, в свою очередь, приписаны строки текстов команд интерпретатора (отображаются и редактируются вызовом окна отображения свойств иконок).
Если вычислительная установка предназначена для обслуживания множества пользователей, то применяемые на ней системы управления сопровождаются службами учета пользователей, разграничения доступа к ресурсам, формирования очередей задач и заданий, информирования пользователей о выполнении заданий.
Остановимся на вариантах очередей задач и заданий (см. Рис. 9).
Как уже отмечалось, для систем немедленного выполнения задач или заданий очереди не создаются. Так, ОС Windows в персональном варианте немедленно приступают к выполнению выбранной пользователем задачи.
А вот многопользовательские системы (серверы, мэинфреймы, др.) такие очереди создают, как правило, в одном или нескольких файлах.
Простейшим случаем является единственный файл, в который последовательно записываются указания на задачи/пакеты заданий, или сами команды или пакеты с ними.
Все записи в файле очереди могут формироваться согласно привилегиям с возможным упорядочением согласно числовым значениям приоритетов и привилегий вне зависимости от времени поступления, назначенными администраторами системы пользователям и их задачам.
Более быстрым вариантом является создание нескольких очередей для задач пользователей.
Одновременно, сами задачи и задания могут обладать собственной приоритетностью, назначаемой их инициаторами, и, возможно, корректирумые системой.
На основе приоритетов пользователей, их задач, возможно, загруженности вычислительной установки, формируется т.н. диспетчерский приоритет, на который опирается диспетчер/супервизор/менеджер задач при предоставлении ресурсов, времени загрузки задач, значения кванта времени в использовании процессора и подобного.

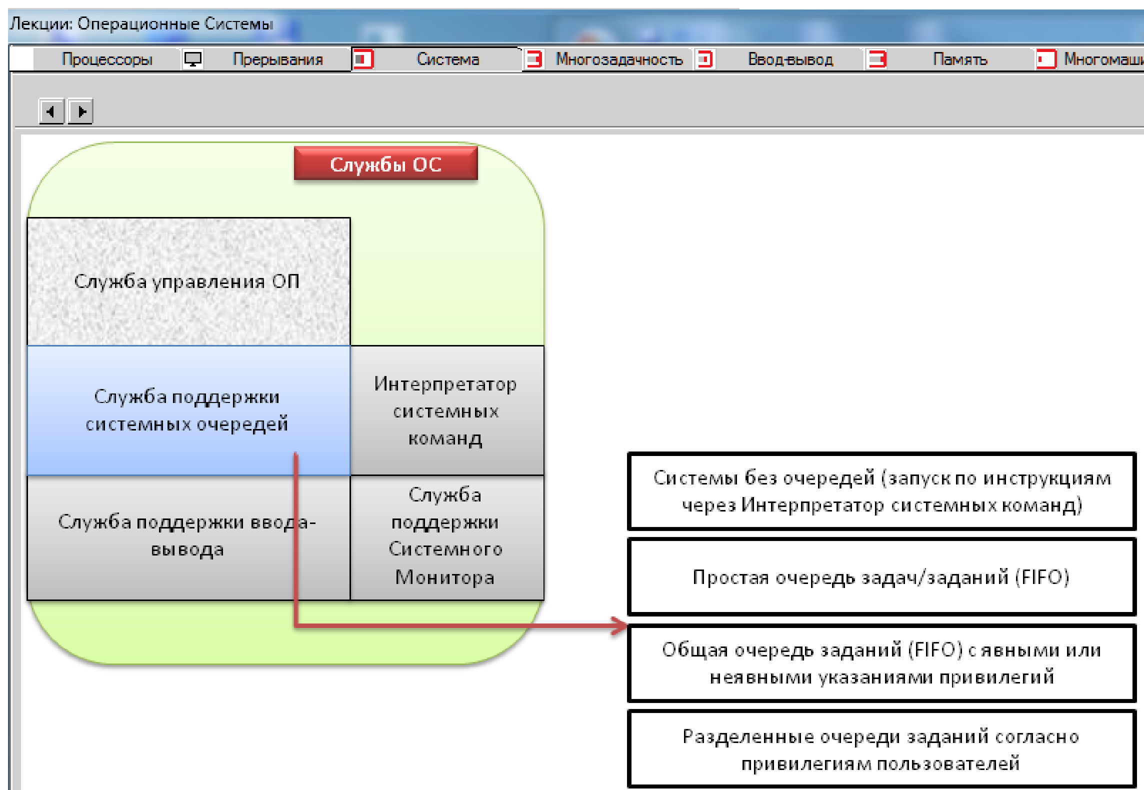


Рис. 13. Варианты организации очередей задач/заданий.

3.2. Загрузчик и Стартер задач.

Опуская случаи, когда есть возможность вызова уже загруженных задач, последовательности шагов в операционных системах, примерно, одинаковы, и могут выполняться двумя компонентами: программой-загрузчиком и программой-стартером. При этом, заметим, что разделение функций между ними довольно условное при выполнении их последовательности. Уже по названию понятно, что Загрузчик выбирает (возможно, указываемую ему) очередную задачу на внешнем носителе согласно логике, принятой в ОС (с очередями или без оных).
В первую очередь выявляется объем оперативной памяти, который потребуется для загрузки задачи. Размер памяти формируется из собственно размера исполнимого кода (размер файла), а также из размеров стека вызовов, динамической памяти, служебной информации (некоторые детали будут отмечены позже), в частности, перечень запрашиваемых приложением динамических библиотек. Обычно, информация о размерах стека (обычно, значения по умолчанию, но пользователь при создании кода может указать собственные величины) и динамической памяти хранится в заголовочной структуре файла с исполнимым кодом, которая формируется на основе выходных данных компилятора и последующей компоновки кода редактором связей. Считав эти данные, Загрузчик запрашивает необходимую память у менеджера памяти, а также загружает необходимые приложению и заявленные заранее динамические библиотеки в среду ОС. Отметим, что приложения могут загружать и выгружать динамические библиотеки самостоятельно, но опять же посредством ОС, загружающей и регистрирующей динамические библиотеки и частично управляющей их использованием. Как правило, динамическая библиотека присутствует в единственном экземпляре (это правило можно обойти), что следует из логики применения библиотек: быть носителем функций для общего применения всеми вычислениями.

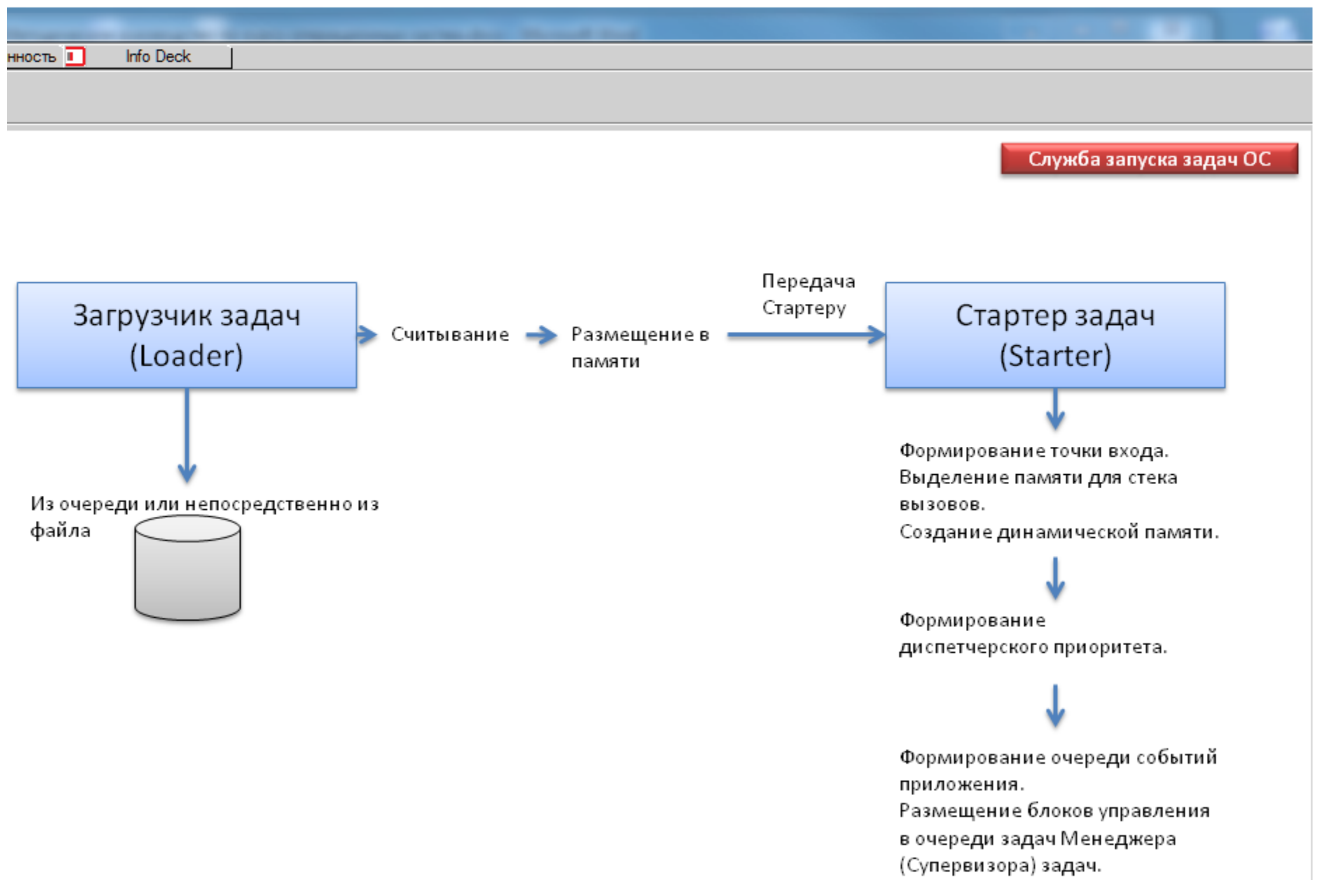


Рис. 14. Последовательность загрузки и старта выполнения задач. В данном варианте выделение памяти под стек вызовов, динамическая память и пр. запрашиваются Стартером задач.

Разместив и подготовив приложение в памяти Стартер задач выбирает из заголовка загруженного файла точку входа – либо прямо определенный абсолютный адрес еще в процессе редактирования связей, либо смещение относительно начала файла. Формируется значение базового регистра. Из Стартера задач осуществляется стандартный вызов загруженной программы, что отражается размещением в стеке вызова указателя на вызвавший модуль стартера и точку возврата в него после завершения работы программы (есть нюансы для многопоточных вариантов). Т.е., опираясь на стилистику и конструкции, к примеру, языка C/C++, адрес начала функции **main(…)** является точкой входа, а оператор **return** из нее возвращает управление в функцию Стартера по адресу возврата в стеке вызовов (адрес следующей за осуществившей переход к вызванной, команды). В машинных кодах откомпилированных программ это выражается секцией кода, назначенной входной, и фрагментом кода, реализующего возврат к вызвавшей секции согласно стеку вызова в модуль Стартера задач.

3.3. Блоки управления задачами (приложения и потоки).

Как уже отмечалось, в операционных системах все ресурсы представляются блоками управления – структурами данных, описывающими ресурс: версия структуры, тип ресурса, характеристики, текущего владельца, если ресурс выделен таковому, и пр. Обычно, блоки управления размещаются в списках (имеются в виду типы данных), таблицах, массивах и пр. Списки удобны в случаях неограниченности ресурсов, более динамичны в построениях. Таблицы и массивы более пригодны при заведомо известных ограничениях по количеству элементов.
Мы будем рассматривать совокупность блоков управления задачами в списочном виде (см. Рис. 11).

В каждом блоке управления задачей указываются

- Адрес загрузки.
- Точка входа в задачу.
- Область сохранения регистров (либо в блоке расположена сама область).
- Область сохранения кэш-памяти или ее необходимых фрагментов.
- Указание на стек вызовов приложения.
- Указание на динамическую память приложения.
- Указание на список блоков регистрации/управления динамически запрошенными пулами памяти приложения (на таблицу страниц в случае организации виртуальной памяти).
- Указание на список подзадач приложения.
- Указание на список открытых файлов приложения.
- Величина кванта времени (для режимов с квантованием времени).
- Величина времени активности программы (время занятости процессора на нее, суммарный объем оперативной памяти на данный момент, пр.).
- Приоритеты задачи.
- Возможное другое, в зависимости от решений, заложенных в комплекс программ управления задачами ОС.

Отдельные элементы таких блоков или все могут быть представлены и в списках потоков (подзадач) приложений. Так, потоки могут иметь

- собственные кванты времени;
- собственные приоритеты;
- обязательно – собственные стеки вызовов;
- точку размещения в случае организации потока через копию программного модуля;
- собственную точку входа;
- пр. служебную информацию.

В ОС Windows при запуске приложений создаются очереди событий фиксированной длины (массивы структур), в которые будут размещаться информация о событиях, относящихся как самому приложению, так и происходящих в целом в среде ОС, но имеющих значение для каждого приложения (изменения на экране PC, требования о закрытии приложения, сообщения от других приложений и многое другое). Потому в блоки управления приложениями и потоками, которые могут иметь собственную очередь событий, размещаются и указатели на очереди событий.
Операционные системы, как правило, позволяют прямо обрабатывать события: назначать модуль (функцию, программу) обработчика событий, перехваченных ядром ОС. В таких случаях в блоке управления приложением фиксируется адрес обработчика в приложении (в некоторых ОС создается таблица переходов к загруженным программам – обработчикам событий, которым управление передается сразу при возникновении прерываний по соответствующим адресам; как правило, это довольно специфические ОС реального времени; аналогичные способы применяются при подключении т.н. виртуальных машин – виртуальных ОС). Заметим, что виртуальной машиной может быть не только ОС, но и исполняющая машина (программа) поддержки языка программирования. В частности, такими машинами обладают язык LISP и LISP-подобные, языки Java и C#, PROLOG, поддерживаемый LISP-машиной, и др.
Возвращаясь к ОС Windows, программа обработки событий указывается в структуре описания характеристик приложения, передаваемая из приложения управлению задачами ОС. Обработчики событий стартуют (подробности в разделе о т.н. “кооперативной многозадачности”) из модуля ОС с размещением в его стеке, как и в случае со Стартерами, указателя на точку возврата в модуль.

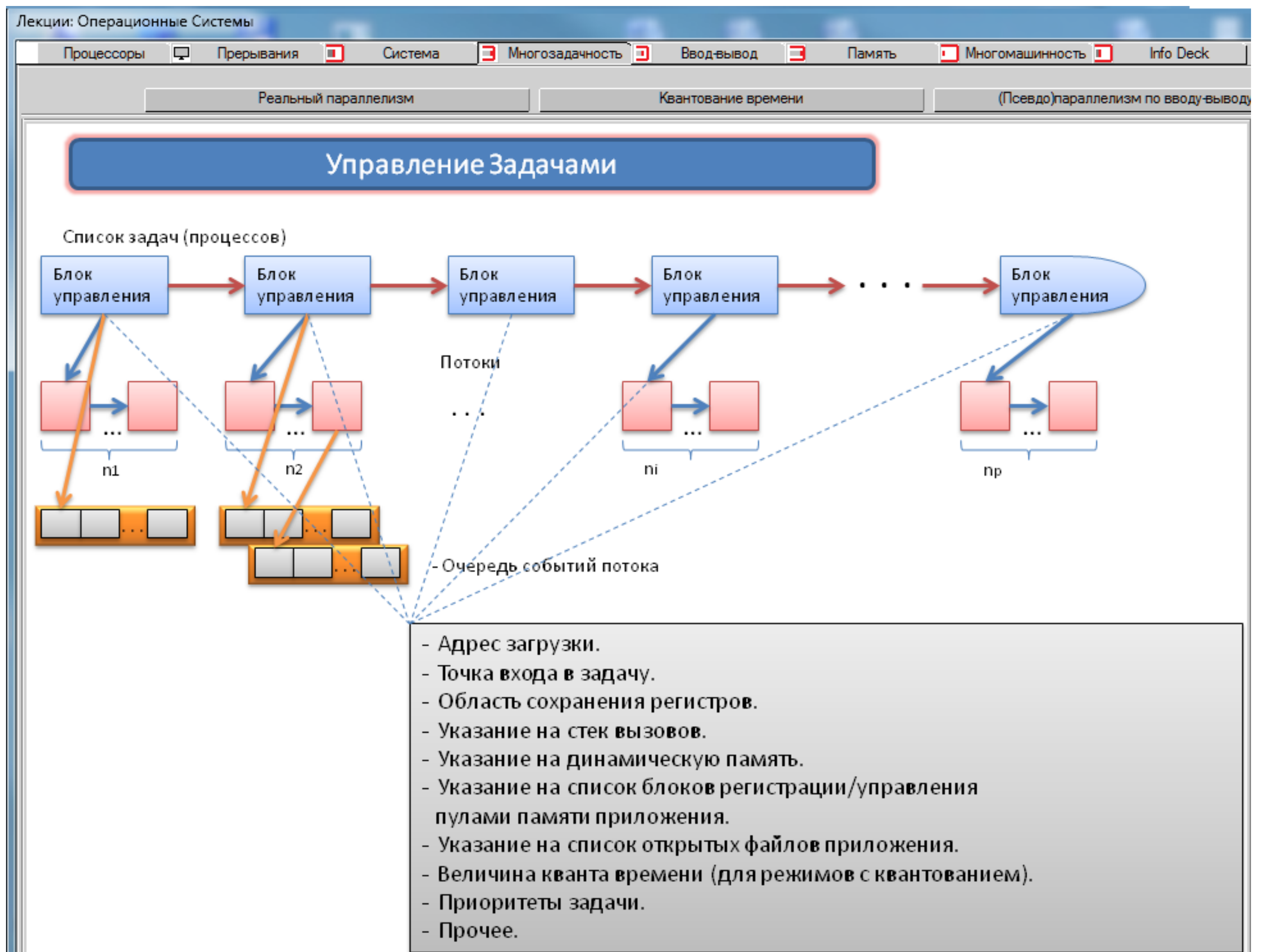


Рис. 15. Списки блоков управления приложений, подзадач. Очереди событий.

Еще одной важной функцией является создание очереди событий в момент запуска приложения, как это принято в ОС Windows. Это действительно не для всех ОС. Например, UNIX-подобные системы очереди как безусловный атрибут, не поддерживают.
Это связано, во многом, с архитектурами вычислителей, для которых разрабатывались данные ОС. В старых архитектурах не были представлены графические дисплеи и средства взаимодействия с ними, потому в настоящее время имеют место программные надстройки над протоколом X11 X Window с надстроенными программными “слоями” вроде Motif, библиотеки Qt и пр.
Архитектуры PC не были предназначены для организации многозадачности (как, в целом, и архитектуры первых поколений PDP, но последние имели хотя бы аппаратные таймеры, позволяющие устраивать многозадачность на основе квантования времени). В то время, как подавляющее число приложений для ОС Windows построено на принципах создания интерфейсов, основанных на логике разнотипных окон (идея не принадлежит компании Microsoft) и множестве событий в окнах приложений. Потому, как правило, первым шагом при программировании таких приложений является описание собственного окна и его вида, указание функции – обработчика событий в окне и в приложении в целом. Обработчики событий могут быть специализированы по типу (например, события от виртуальных таймеров). Для UNIX-подобных схожее организуется посредством, к примеру, библиотек пакета Qt, т.е. надстройки над средствами ОС.
Самыми распространенными событиями здесь являются
- прохождение курсора мыши над окном;
- нажатия/отжатия клавиш на мышке (при нажатиях возможны фиксации длительности), прокрутка колесика мышки, нажатий на него;
- нажатия/отжатия клавиш на клавиатуре;

- прочее, например, создание новых окон, изменение их размеров, графических компонентов на окнах, запуски и терминирования параллельных процессов.

Из важного отметим, что очереди событий опрашиваются синхронно и асинхронно, но с помощью синхронных шагов – обращений из приложения (т.е. инициатором опроса является приложение с помощью вызова необходимой функции опроса, что здесь интерпретируется как синхронный шаг). Однако, вызов функции **GetMessage()** из API Windows приводит к остановке приложения до появления события (синхронизм), в то время, как вызов функции **PeekMessage()** немедленно возвращает управление в вызвавший поток. И в том, и в другом случае обработка события, если таковое случилось и расположено в очереди событий, начинается опять же синхронно после вызова функции **DispatchMessage()**, являющейся обращением к менеджеру задач инициировать обработку указываемого события.

Последнее приводит к вызову менеджером задач функции-обработчика событий, назначенной самим приложением. Отметим, что синхронный вызов обработчика событий легко превратить в асинхронную обработку запуском параллельного процесса (потока в Windows) с передачей ему задачу обработки и возвратом из назначенного обработчика к приложению сразу после **DispatchMessage()**.

Размещение событий в очереди событий может осуществляться согласно назначенным фильтрам. Опрос очереди событий приводит к изъятию событий из очереди вне зависимости от того, обрабатывается событие или игнорируется.

На Рис. 15 представлены основные компоненты и схема обработки событий в ОС Windows. В целом, каждая из функций является средством передачи управления операционной системе и стать причиной переключения на другой поток. **GetMessage()** – в особенности, т.к. ожидание события, практически, гарантированно приведет к активизации другого потока. Именно этот факт послужил основой к появлению термина “кооперативность” в разделении времени процессора: приложение само отдает ресурс (вызовом функции ОС) для использования другими приложениями.

Однако здесь присутствуют подводные камни, которые известны разработчикам операционных систем: приложение может захватить процессор и использовать его бесконечно долго (по крайней мере, до физического износа вычислителя), если его не прервать извне. Единственным способом здесь является аппаратное прерывание. Архитектура процессоров x86 была в этом смысле крайне упрощена, а потому “подвесить” ОС приложением или даже службой самой ОС (“голубые экраны”) было довольно просто.

В частности, в тех случаях, когда виртуальной машине языка Caper [12,13] требовался весь ресурс процессора при псевдопараллельном выполнении большого количества параллельных процессов (десятки и сотни тысяч), то операционная система “подвисала” до тех пор, пока виртуальная машина не завершит работу с таким количеством процессов. Причина банальна: достаточно не обращаться к модулям ОС. Можно привести множество других задач подобного толка, когда ОС отходит на второй план. В частности, это комбинаторные задачи, осуществляющие перебор в оперативной памяти и не обращающиеся к ОС.

Т.е. общая логика “кооперативной многозадачности” заключается в том, что приложение само передает управление менеджменту ОС, которое, “пользуясь моментом”, может переключить задачи и их потоки. Иначе, всякое приложение имеет возможность монополизировать ресурс процессора, если в архитектуру не включены высокоприоритетные средства прерываний, позволяющих перехватить процессор у приложения. Это безусловное нарушение принципов построения ОС, т.к. основной ресурс вычислителя может попасть под безраздельное управление приложения. Во многом причиной этому является отсутствие аппаратного таймера, т.к. при переключении приложений можно (так и происходило) назначать квант времени для выполнения приложение, которое прерывалось при исчерпании таймера.

Заметим, что многоядерные процессоры разрешают такие ситуации, т.к. корректно построенные ОС с момента загрузки захватывают под свои нужды по крайней мере одно ядро с недопущением его использования приложениям. Следует отметить, что это условно “дорогой” способ.

В целом, такая проблема может возникнуть и с другими архитектурами процессоров с куда более разнообразным набором прерываний из-за невнимательности или ошибки системного администратора.

События и их обработка в OS Windows

```
If ( result = PeekMessage( &gMsg, NULL, 0, 0, PM_REMOVE ) )
 {
      . . .
      res = DispatchMessage( &gMsg );
      . . .
 }
If ( (result = GetMessage( &gMsg, NULL, 0, 0 )) >= 0 )
 {
      . . .
      result = DispatchMessage( &gMsg );
      . . .
 }
```

PeekMessage(. . .) – асинхронное обращение за событием в очереди; вне зависимости от результата продолжает работать.

GetMessage(. . .) – синхронное обращение за событием в очереди; если очередь пуста, то программа останавливается до появления события.

DispatchMessage(. . .) – отправка записи с событием на обработку через обращение к модулю управления очередями, который, в свою очередь, вызывает назначенную в качестве обработчика событий функцию приложения.

CALLBACK-функция – функция, вызываемая модулем управления очередью событий с передачей кода события и его атрибутов. Программа останавливается до завершения функции.

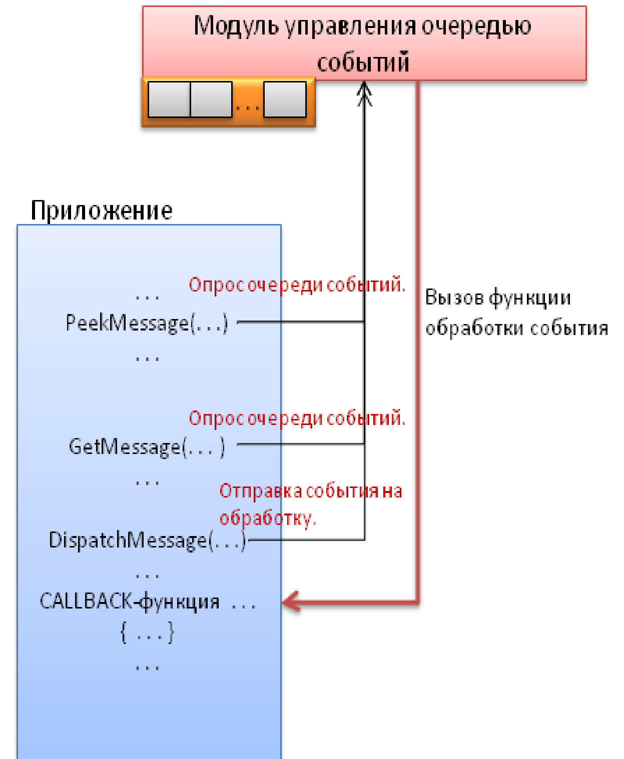


Рис. 16. Схема обработки событий в ОС Windows

Здесь мы не описываем функции создания и размещения событий как в собственной очереди приложения, так и в очередях событий других приложений. Отметим, что таковые есть: **SendMessage()** и **PostMessage()**, осуществляющие, соответственно, синхронное и асинхронное размещение событий как в собственную очередь, так и в очереди других приложений.
В UNIX-подобных системах очереди событий можно организовать “вручную” с помощью специальных функций, работать с ними посредством аналогов указанных выше функций для Windows (подчеркнем, что функции создания очередей событий и их удаления в Windows не присутствуют, т.к. очереди создаются автоматически, а функции обработки представлены в API).

3.4. Переключение задач и потоков.

Переключение с задачи на задачу, с потока на поток осуществляется супервизором (менеджером и подобным) задач операционной системы в привилегированном режиме, как правило, с блокированием или минимизации большинства событий в ОС, т.к. такое переключение сопровождается работой с аппаратным оснащением процессора, что требует минимального отвлечения супервизора.
Механизм переключения на одном процессоре довольно прост: анализируя список (прохождением по нему) блоков регистрации задач и потоков на предмет их приоритетности, согласно логике, установленной в ОС (см. Рис. 15), выбирается активизируемый поток, сохраняется текущее состояние аппаратных средств (регистры управления, общие регистры, кэш-память), загруженных данными исполняемой задачи в области управления задачей, и восстанавливается в аппаратных средствах сохраненные или установленные данные выбранной к

исполнению задачи (или ее остановленного ранее потока), в том числе, содержимое кэш-памяти процессора, бывшее на момент остановки потока.
Возможное отсутствие машинного кода и данных собственно задачи (потока) в оперативной памяти приводит к осуществлению операций свопинга (подкачки в ОП из своп-файлов).
Восстанавливается IP. Следующей исполняемой процессором командой станет команда активизированного потока.

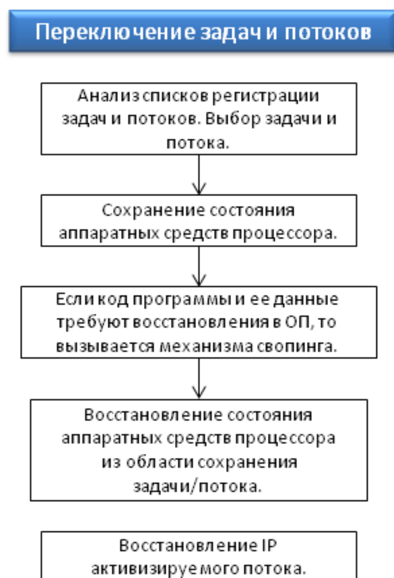


Рис. 17. Переключение задач и потоков.

Межпрограммное взаимодействие может быть обеспечено как операционной системой (в ОС Windows это осуществляется с помощью сообщений, которые размещаются в очередь событий того же или другого приложения, в очередь событий потока, если таковая была создана при старте потока), так и специальными библиотеками программ, поддерживающими межмашинное взаимодействие (в частности, библиотеки на основе стандарта MPI).

3.5. Прикладные задачи и запросы ресурсов операционных систем.

В процессе выполнения пользовательским программам в разные моменты времени могут потребоваться различные ресурсы ОС, которая, напомним, распоряжается всеми ресурсами среды, в которой работают программы. Соответственно, все прикладные программы в случае необходимости того или иного ресурса обращаются к соответствующим службам ОС.
Логика запросов ресурсов укладывается в довольно простую схему вызовом функций:
- запроса ресурса;
- использование ресурса посредством обращений к службе;
- освобождение ресурса.

В известных случаях использование ресурса осуществляется средствами программирования без привлечения служб. Например, можно запросить память у службы управления памятью, но использовать ее прямо из программы, к примеру, посредством указателей. Однако, при работе с файлами после запроса на использование файла и его предоставления различные операции чтения и записи в файл осуществляются посредством службы ОС, при том, что файл или его фрагмент располагается в ОП в буфере.
Если ресурс с определенного момента не нужен, то его необходимо освобождать, при том, что ОС с завершением задания сама высвобождает все ресурсы, предоставленные приложению (по крайней мере, должна). Но правилом хорошего тона и признаком профессионализма следует высвобождать ненужные ресурсы из самой программы. Так необходимо поступать и с полученными ранее участками ОП, и с открытыми файлами, и с прочими.

Еще одним ресурсом, которым может оперировать приложение – это динамические библиотеки, которые также загружаются и освобождаются службой ОС посредством функций обращений из приложений. Однако, данные библиотеки становятся ресурсом ОС (каждая в единственном экземпляре) и могут быть предоставлены другим приложениям. А в этих случаях освобождение и выгрузка библиотек может оказаться невозможными.
Подчеркнем, что отдельные прикладные программы и даже целые приложения могут стать ресурсом ОС. Здесь имеет место двойственная трактовка ресурса, когда прикладная программа, прикладные данные, будучи ресурсами приложений, становятся ресурсами, контролируемым ОС. В общем-то, с определенной точки зрения это относится и к прикладным программам, так как с момента загрузки они, в определенном смысле, становятся контролируемым ресурсом ОС.

Глава 4. Память и управление ею.

Управление оперативной памятью – одна из важнейших и сложнейших компонент любого универсального вычислителя. Важнее, как представляется, любого устройства, в том числе, процессора. Т.к. именно в ОП располагаются программы, именно они требуют размещения своих данных, при этом каждой программе "мешают" другие программы.
Проблемы начинаются с момента загрузки очередной программы: как и куда ее разместить в памяти, как размещать все данные, требуемые для организации вычисления, данные, возникающие в ходе вычислений, и пр. При этом, сохраняется требование: доступ к оперативной памяти должен быть максимально быстрым, т.к. от этого зависит скорость исполнения вычислений, а, следовательно, и общая эффективность вычислителя.

4.1. Формирование памяти вычислителей и приложений.

Во многом, на организацию памяти повлияло решение IBM-360 с составной памятью блоков с электронными ключами и разделение всей памяти между ОС и приложениям по ключам. Длина ключа в данном случае (в первоначальном варианте – 4 бита на четырехкилобайтовый блок) не важна, важна логика блоков в процессе выделения памяти, а не отдельными группами байтов. Электронный ключ памяти носит защитную функцию при использовании всей памяти разными приложениями. Эта логика перешла по наследству в ОС UNIX, а позже в его производные. Поддерживать защиту от постороннего вмешательства каждый байт в случае безблоковой организации, мягко говоря, является сложным (регистрация огромного количества фрагментов и "фрагментиков" может потребовать больше памяти, чем суммарный объем самих фрагментов). Защищаются блоки и их совокупности.
Размеры блоков могут иметь разную величину в зависимости от вычислителей и их возможностей. В случаях памяти с большими блоками динамически запрашиваемая память может выделяться из них, уже зарезервированных для приложения. Есть и риск определенной избыточности: блок может быть больше, чем необходимо приложению. Но идеальных решений не существует, как известно, алгоритмизация и программирование – это всегда поиск компромиссов и балансов.
Как уже отмечалось, списки блоков памяти в некоторых ОС организуются там же в блоках, что делает их уязвимыми (их блоки управления) перед самим приложением (см. ниже).

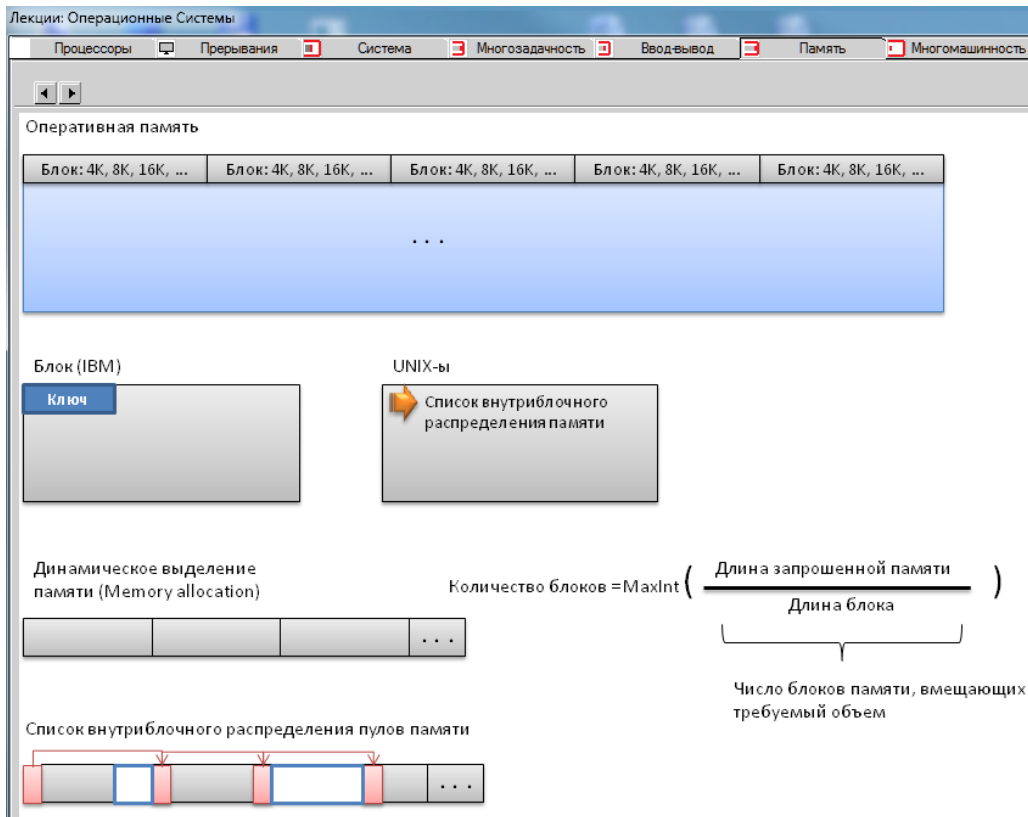
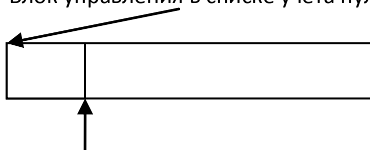


Рис. 18. Блоки памяти и списки внутриблочного распределения элементов памяти.

Электронные ключи блоков памяти и аппаратные средства контроля доступа позволяют более “свободно” распределять память (что делается в системах IBM) и применять разные модели в ее управлении. Однако, нет идеальных схем, каждый раз при создании ОС исследуются и разрабатываются (иногда довольно сложные) алгоритмы по выделению и освобождению участков памяти. Отметим, что произвольное размещение участков памяти получило название несмежной (управление несмежной памятью; существует т.н. управление смежной памятью, когда проверяется соседство освобождаемого участка (смежные), и если хотя бы один из них свободен, то освобожденный объединяется с соседним).
Осуществляя выделение и освобождение памяти мы получаем “дыры” различных размеров – неравномерную фрагментацию всей памяти. Это приводит к постепенно растущим потерям времени по ее управлению. По этой причине некоторые ОС имеют службы дефрагментации – минимизацию количества фрагментов, которые срабатывают эпизодически на основе установленных эвристик. Примерно так же работает служба дефрагментации для внешних носителей памяти, в первую очередь, для дисков.
При выделении памяти служба управления (в языке C/C++ в стандартах прописаны функции **malloc()**, **calloc()**, для изменения участка памяти с сохранением его содержимого или части используется функция **realloc()**) ищет в разрозненных участках фрагмент подходящего размера (если разбиение памяти основано на блоках, то, как уже говорилось, блоки проверяются на возможность выделения из их свободных участков). Функция **calloc()** при этом инициализирует (по умолчанию – обнуляет) все байты выделенного участка. При этом, в UNIX-подобных ОС, как и в Windows, унаследовавшего данное решение, выделенный участок предваряется блоком управления элемента списка.



Pointer - указатель на выделенную память – вертикальная стрелка на рисунке.
pointer = malloc(. . .); // указатель на выделенную область

Смещение по ошибке “влево” от “pointer” приведет к разрушению данных в элементах управления списка, что приведет к аварийному завершению как функций выделения памяти, вызываемых впоследствии, так и функции ее освобождения **free()**. По понятным причинам: списки однонаправленные, прохождение по списку будет нарушено внутри элементов списка при выполнении указанных функций.
Очевидна, как уже указывалось, и причина: нарушено одно из основополагающих требований к ОС, а именно, данные управления системным ресурсом располагаются в области прикладной задачи, т.е. доступны, а вызов системной функции приводит к ее аварийному завершению, что недопустимо для ОС (см. требования к ОС).
Есть и рекомендация: не работать с указателями в выделенном пуле памяти, а только с прямой адресацией элементов массива индексами (со структурами, конечно, тоже), и обязательной установкой контроля индекса, если среда разработки позволяет (контроль на значение получаемого индекса меньше минимального (0 или 1 согласно принятой нумерации в языке программирования)). Иначе, выражение в языке С, к примеру, типа “**array[i] = 10;**” вполне может привести к ошибке, если нет проверки индекса на **i < 0** и **i >= N**, где **N** – размер массива.
Некоторые компиляторы позволяют встраивать проверки автоматически при указании соответствующего параметра.
В целом, указатель – штука потенциально опасная, требует значительного внимания. При этом проблемы могут возникнуть и с переменным в динамической памяти, если, к примеру, взять адрес переменной, а следом заполнять данными по сформированному указателю (естественно, в случае ошибок программиста).

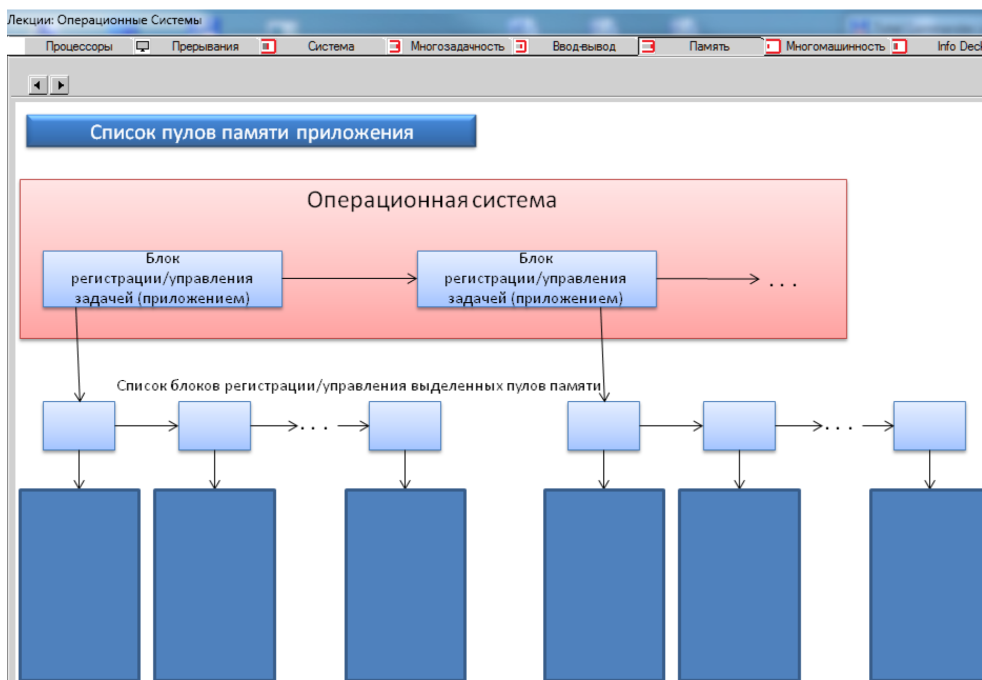


Рис. 19. Блоки памяти приложений с регистрацией в списках.

4.2. Оверлеи, свопинг.

Недостаток памяти в ходе вычислений – проблема давняя, со дня появления первого компьютера, и со временем стала решаться программистами приложений с помощью создания т.н. оверлеев. Что, конечно, влекло дополнительную нагрузку на разработчиков программ.
Суть оверлеев заключается в том, чтобы структурировать программу, ориентируясь на возможность выгружать ее модуль из оперативной памяти и загружать на его место другой модуль. Во многом, концепция Вирта по модульности программ и программирования возникла, в частности, из этой идеи.
На Рис. 20 представлены два варианта оверлеев: “строгая” и “слабая”. Первый вариант предполагает выделение области памяти, в которую поместятся все оверлейные модули, с последующими загрузками модулей из внешней памяти в данную область, их использованием и загрузки нового модуля в освобожденную область (выгрузка необходима в случае незавершенности работы модуля и необходимости сохранения его текущего состояния с последующим восстановлением).
После определенных практик в программировании подобная технология с вариациями была включена в ОС от IBM. Так поначалу ОС работала со страницами памяти фиксированной длины, хранимыми на дисках. Под их подкачку и последующую выгрузку выделялся фиксированный участок ОП.
Слабая оверлейность предполагает ту же схему загрузки-выгрузки, но необязательно в ту же область памяти, с освобождением или резервированием предыдущей. Это связано со случаями, когда первоначальная область недостаточна по размеру для нового загружаемого модуля. Резервирование используется, когда предполагается возвращение модуля в память. Однако, повторим, такая схема недостаточно “оверлейна”.
Для поддержания строгого варианта оверлейная область может быть заказана еще до начала выполнения задачи в задании для ОС с учетом максимального размера загружаемых в нее модулей.

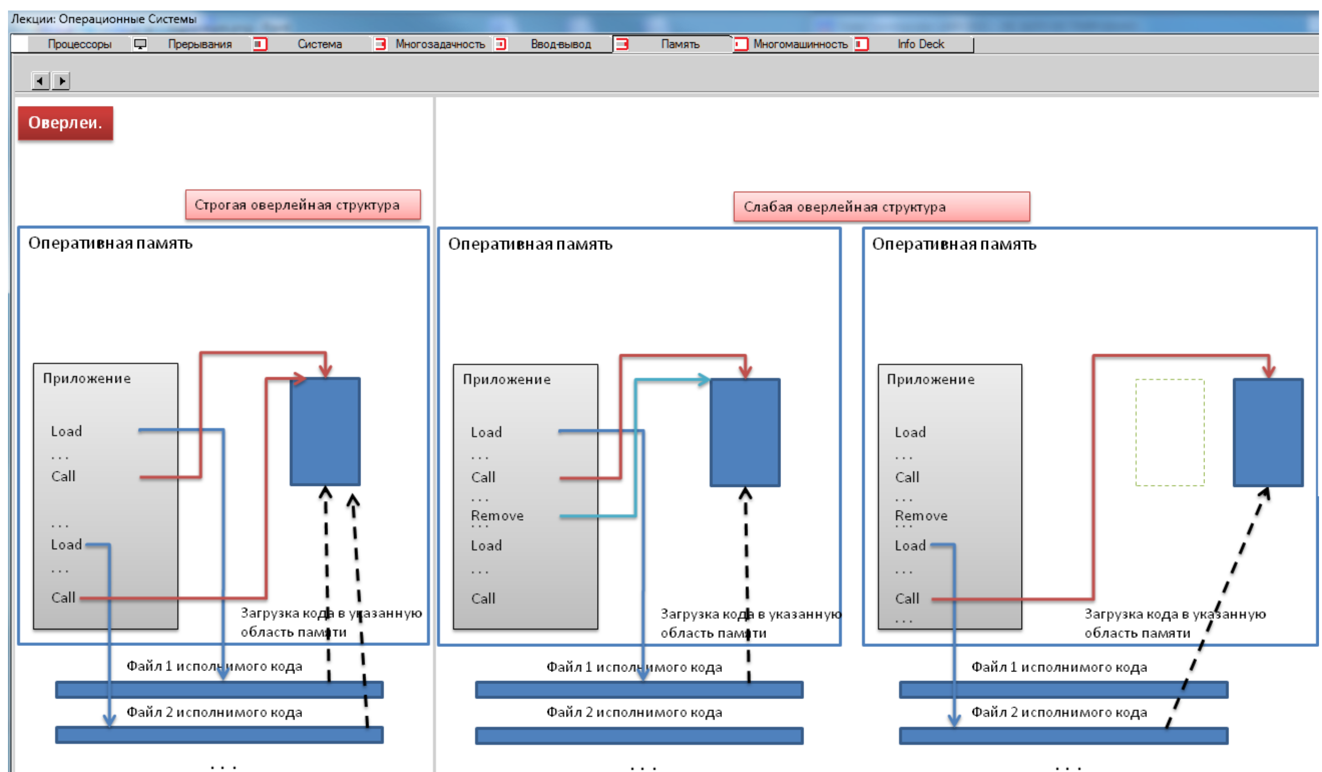


Рис. 20. Оверлеи.

В качестве примера на Рис. приведен фрагмент кода на языке Caper, который обладает средствами загрузки (**import** <имя файла> **as** <имя процедурного блока>) программного модуля в указанном файле в качестве именованной совокупности процедур, так же являющейся процедурой, и выгрузки (**remove** <имя процедурного блока>) программных модулей, и обладает свойствами слабой оверлейности.

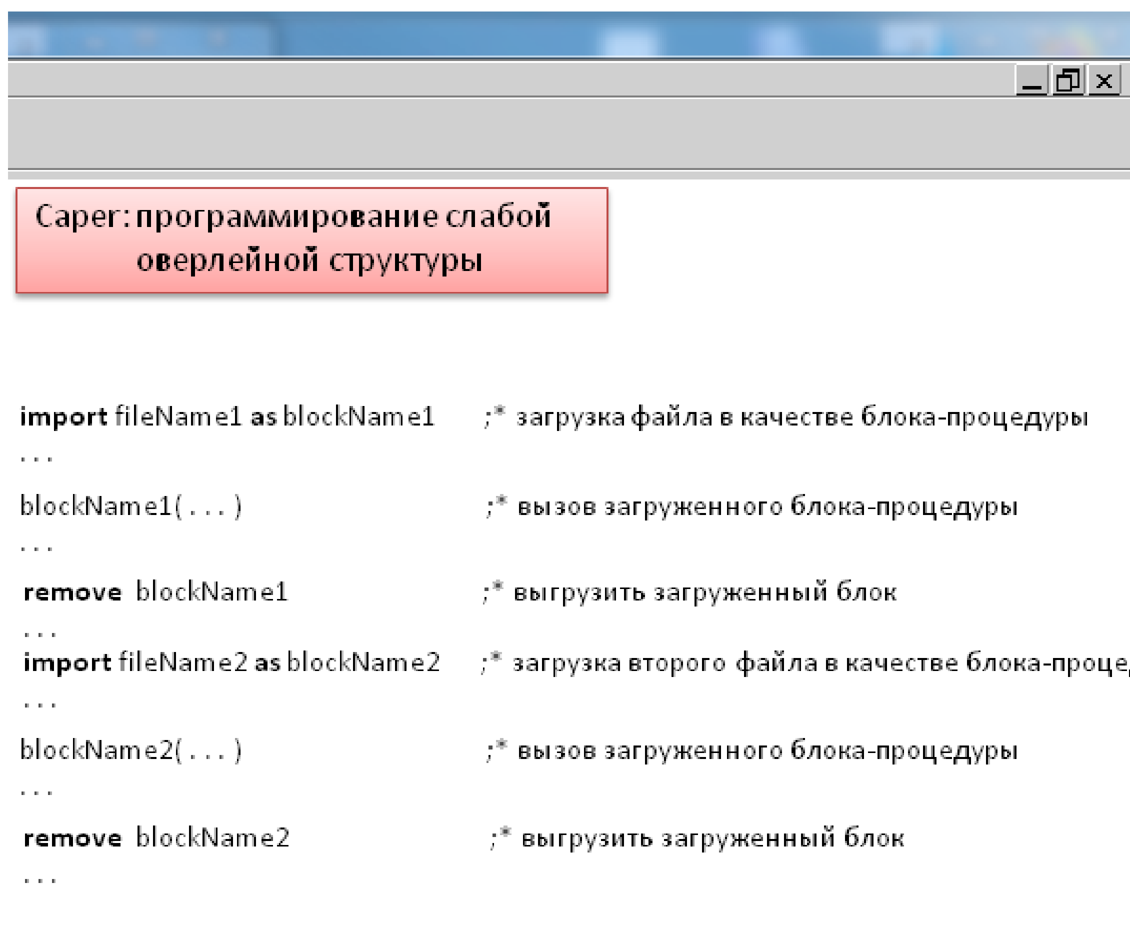


Рис. 21. Программирование слабой оверлейности в Caper.

4.3. Организация виртуальной памяти.

Определенным наследником оверлейной организации вычислений является способ построения операционными системами виртуальной памяти.
Одним из распространенных методов, используемом в распределении памяти, является т.н. страничный метод организации.
Подход был разработан компанией IBM при создании варианта SVS (система виртуальной памяти) ОС, которая быстро трансформировалась в VM (система виртуальных машин в советской терминологии).
Малые машины с процессорами вроде x86 компенсировали отсутствие “сложной”, а следовательно - дорогой памяти с электронными ключами, виртуальными адресами, которые транслировались в реальные адреса с помощью специальных преобразователей на основе таблиц сегментов памяти, задаваемых в двухбайтовых регистрах (см. Рис.). В целом, такой подход возник с появлением ОС для компьютеров VAX компании DEC.
Идея страничной памяти укладывается следующую логику:
 - разбиение всей реальной (физической) памяти на фиксированные по длине непересекающиеся

области;
- загрузка программ и ее данных в выделенную для нее область;
- все манипуляции с памятью ограничиваются выделенной областью и адресацией внутри нее;
- в случаях нехватки размеров области для всего кода программы или ее данных организуется частичные загрузки фрагментов программы и данных.

С этими целями формируются файлы на диске для выгрузки в разделы файлов и фактического освобождения оперативной памяти под новые подгружаемые фрагменты программ и данных. Такая процедура называется свопингом (от SWAP (не аббревиатура), swap, своп - обмен).
Еще более ранняя процедура, поддерживаемая ОС, - это дамп (dump) памяти – прямая выгрузка содержимого оперативной памяти или ее фрагментов на внешний носитель. Как правило, она использовалась и используется для отображения памяти с целью поиска ошибок. Однако, автор данного текста использовал данную процедуру для сохранения во внешней памяти текущего состояния приложений с определенной периодичностью на случаи аварийных состояний вычислителя (отключения питания, в частности), и последующего восстановления. Такая процедура важна для вычислений большой длительности.
Процессы swapping-а и дампирования могут быть организованы по-разному и на разных носителях. Актуально, чтобы носитель был с высоким быстродействием (используются HDD и SSD; последние при высокой скорости имеют недостаток, связанный с ограниченностью количества перезаписей), т.к. быстродействие внешнего носителя безусловно влияет на величину пауз, возникающих из-за обмена.
Одним из ключевых преимуществ логики страниц является фиксированность размеров страниц в ОП и возможность сохранять их “одним движением”, за одну операцию записи и чтение.
Часто в ОС применяется уже описанная логика объединенных последовательных блоков памяти, и так же называемых страницами.
Так или иначе, в страницы могут добавляться новые элементы данных, могут удаляться из них. Динамика изменений со временем приводит к фрагментации памяти как в ОП, так и в swap-файлах, что начинает тормозить процесс вычислений при поисках необходимых данных. В таких случаях на основе эвристических правил проводится дефрагментация, процедура затратная по времени.
Учитывая этот факт, помимо собственно упрощения манипуляциями с переменными и упрощения их контроля (напомним, что в C/C++ динамическое создание составных переменных требует их освобождение по завершении работы с ними, что требует дополнительного внимания) в некоторых распространенных языках программирования создается собственная память значительных объемов с возможностью расширения, в которой учитываются переменные и размещаются все, связанные с ними, данные. Такие языки основаны на наличии собственных виртуальных машин, которые, в том числе, управляют собственной памятью приложений. В этих случаях применяется процедура “сборки мусора”, логика которой наследована от языка LISP.
Во многом процедура дефрагментации схожа с такими процедурами, усложненных учетом общих полей данных для различных параллельных потоков (ведется их учет, используются счетчики пользователей конкретными данными, пр.).
Виртуальная память с виртуальными адресами формируется из нескольких слоев с вариациями. Обычно все начинается с таблицы страниц (фреймов) в физической памяти. Всякий адрес формируется из номера блока (фрейма), записанного в таблице и смещения внутри страницы. Из номера блока и смещения формируется абсолютный адрес в реальной оперативной памяти. Таким образом, формируются адресные пространства приложений, замкнутые в наборы страниц. В целом, мы имеем дело с вариантом относительной адресации.

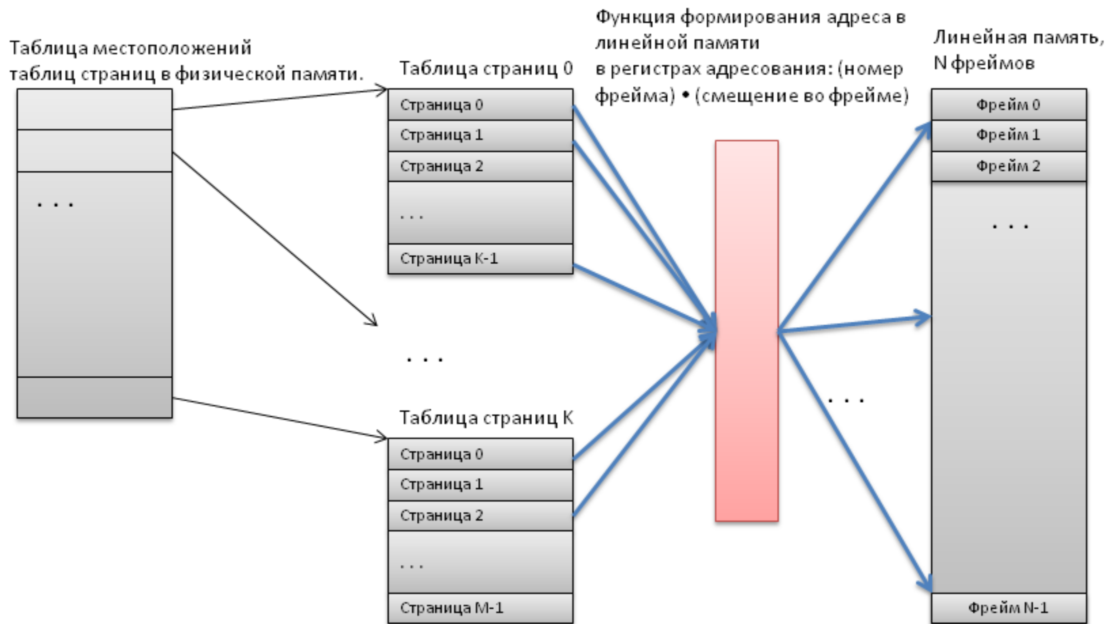


Рис. 22. Схема преобразователей виртуальных адресов в реальные физические.

В целом, в современных 32-битных архитектурах мы имеем 16-ть бит для нумерации страниц и 16-ть битов для смещения: имеем 4 Гб адресуемого пространства (иногда ограничивается до 2 Гб).

4.4. Виртуальные машины и виртуальные операционные системы.

Виртуальная машина (ВМ или VM) – программный комплекс, реализующий собственный подход к описанию и реализации вычислений, заданных прочими программами на языках программирования. Как правило, виртуальные машины снабжаются собственным языком, схожим с ассемблерами, в различных случаях называемым байткодом (Java), промежуточным языком (Intermediate Language, как в C#, F#). Автор предпочитает собственное название, введенное для языка Caper – псевдо-ассемблер. Главное предназначение ВМ и их отличие от ассемблеров процессоров – обработка низкоуровневых инструкций на основе собственной логики их интерпретации.
Виртуальные ОС обладают всеми основными атрибутами обычных операционных систем, однако опираются на наличие базовой операционной системы и пользуются ее рутинной частью, в частности, элементами ядра, средствами обслуживания физических устройств, первичной обработки прерываний и пр. (заметим, что большинство прерываний транслируются на виртуальную ОС).
Подчеркнем также, что виртуальная ОС предназначена для управления ресурсами вычислительной установки, управления организации вычислений и пр. по собственной логике, отличающейся от логики базовой ОС (иначе, как говорится, нечего и огород городить).

Глава 5. Особенности разработки программ ОС.

Особым звеном в представлении операционных систем и их многозадачности/многопоточности выделим технику программирования компонент, которые должны реагировать на те или иные события или обращения к программам ОС со стороны приложений. Вопрос очевиден: как программы операционных систем должны быть организованы для обработки множества обращений к ним из разных одновременно выполняемых задач/потоков?

Здесь могут быть применены следующие варианты: реентерабельность (reenterability - повторновходовость) программ, метод создания копий программных модулей и создание мьютексов.
Проблема заключается в том, что в разные моменты времени к одной и той же программе или ее фрагменту в оперативной памяти (в частности, принадлежащих ОС) могут возникнуть обращения (вызовы) из других программ. Данная проблема возникает только для многозадачных систем, в которых организуется параллельное или псевдопараллельное исполнение программ, т.е. формируется набор активных исполняемых последовательностей машинного кода.
Подчеркнем, что представленные ниже методы разработки и программирования вполне применимы и в прикладных программах.

5.1. Реентерабельные программы.

В случаях, когда вызываемая программа использует носители данных (переменные, участки ОП и пр.), имеющие статичный характер (т.е. были выделены программе в момент ее старта и существующие до конца вычислений) или оперирует с каким-либо устройством, то ее новый вызов может привести к незавершенности уже текущих операций, изменению переменных и прочих полей данных уже при исполнении нового вызова всевозможными новыми размещениями, изменяя те, что были при предыдущем вызове (см. Рис. 12).
По этим причинам к программам, обслуживающим множества вызовов, предъявляется требование быть реентерабельными, готовыми к повторным вызовам.
Решения здесь применяются довольно простые: не используются статичные переменные и пр. поля данных. Или, если используются, то не имеют влияния на суть алгоритма. В качестве примера можно привести статичную переменную – счетчик количества входов в программу, т.е. статичная переменная, инкрементируемая при каждом входе и декрементируемая при выходе.
В реентерабельных программах для каждого вызова создаются новые копии переменных и соответствующие им поля данных (простые локальные переменные формируются в стеке вызовов, в то время, как массивы данных и структуры требуют запросов на динамическое выделение новых участков).
Одновременно, для реентерабельных программ исключено применение изменения или самоизменения в ходе вычислений– еще одно свойство ассемблеров: программа расположена в ОП, а, следовательно, ее код доступен для изменений как изнутри программы, так и снаружи из программных модулей с теми же ключами памяти. Изменение программы в машинных кодах, как и самоизменение, невозможно для архитектур с разделенной памятью для программных кодов и данных, с блокированием записи в поля с программными кодами.
Итак, для обслуживания множества запросов может оказаться достаточной одна копия программного кода.

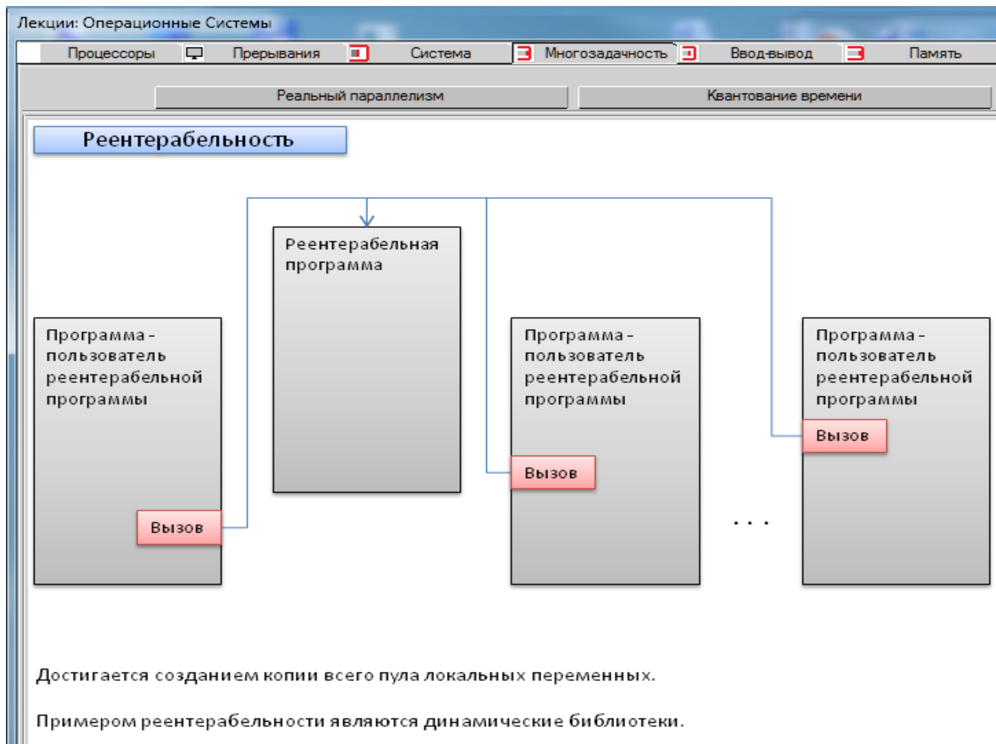


Рис. 23. Разномоментные вызовы реентерабельных программ.

5.2. Копии программных модулей.

Другим способом является более очевидное: для каждого вызова создается собственная копия обрабатывающего программного кода. При этом оригинал может уже находиться в оперативной памяти или быть загружен с внешнего носителя.
На Рис.13 представлены средства загрузки программных модулей макрокоманды Load (загрузка в ОП необходимого модуля), Copy (копирование уже загруженного в память модуля в другую область) и Call (вызов модуля, находящегося в памяти) в ОС от IBM, и fork и exec в UNIX-подобных ОС.
На этой основе формировалась многозадачность в UNIX-ах, когда создавалась копия не только пользовательской программы, но и текущего состояния ядра ОС, а также некоторых ее служб.

5.3. Мьютексы.

Еще одним способом корректного решения проблемы множественных обращений к одному и тому же программному коду является более поздний по времени создания механизм мьютексов (фнукции, процедуры, секции кода, подобное).
Мьютексы регистрируются операционной системой из прикладной программы вызовом функции ОС с указанием собственной функции в качестве мьютекса (ничто не препятствует тем же механизмом пользоваться и самой ОС для создания мьютексов в собственных целях) программы, которая считается собственником мьютекса (освобождает его от регистрации она же, или ОС в случаях “забывчивости” или аварийного завершения владельца мьютекса).
Обычно, мьютексы предназначены для изменения общих для различных потоков вычислений переменных, но, в целом, этот способ пригоден для любых целей.
Вызов мьютекса во время его работы (по сути, занятости) по вызову из другого потока приводит к блокированию (остановке) выполнения вызвавшего потока.
Кроме того, помимо остановок (псевдо)параллельных потоков этот механизм сопровождается задержками, связанными с контролем обращений к мьютексам.

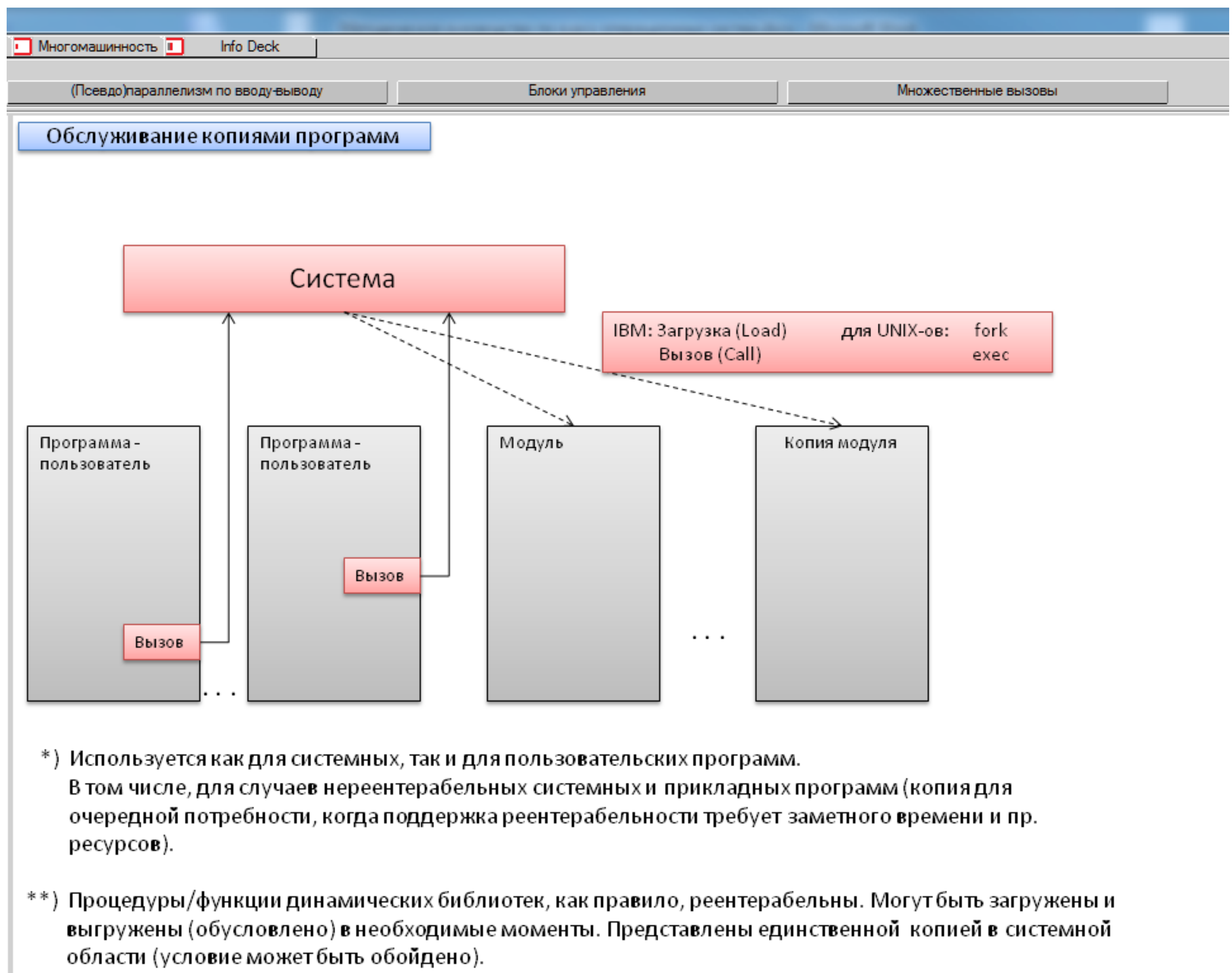


Рис. 24. Организация обработки множества вызовов созданием копий программного кода.

Глава 6. Ввод-вывод. Файловые системы. Методы доступа.

Одной из основных компонент любого вычислителя являются средства долговременного хранения и отображения информации. Данные средства отличаются большим многообразием и способами работы с ними.

6.1. Устройства ввода-вывода и методы доступа к ним.

Взаимодействие с вычислительными устройствами начиналось с пульта устройства с помощью всевозможных переключателей, кнопок и т.п., а результаты отображались лампочками того же пульта (индикацией).
Вскоре появилась печатающая машинка, исполняющая функции ввода и вывода, позже – специальные печатающие устройства, устройства ввода на перфокартах и перфолентах, еще позже – алфавитно-цифровые дисплеи с клавиатурой, плоттеры, графические дисплеи, сканнеры и еще много разновидностей специальной техники. Все это стало содержимым понятия “периферия” – совокупности подключенных раздельными или совмещенными (данные + управление) шинами передачи данных устройств ввода-вывода информации.

Появление разнообразных устройств и решений в их исполнении привело к стандартизации интерфейсов устройств и вычислительной машины.
В программной части, обеспечивающей взаимодействие устройств и компьютера, законодателями мод стали, опять же, специалисты компании IBM, которые разработали исчерпывающий набор методов и вариантов организации ввода-вывода.
Дисководы и дисковые накопители, накопители на магнитных лентах, перфокарты, перфоленты и простая бумага стали средством накопления выводимой информации. Одновременно, те же средства являются источниками для ввода информации в память компьютеров (бумага, конечно посредством сканнеров).
Обеспечение соединений с устройствами и ввода-вывода осуществляется посредством контроллеров (по существу - микропроцессоры), или специализированных процессоров, обладающих собственными средствами программирования (упрощенными ассемблерами).
Устройства разделяются
- по быстродействию: быстрые и медленные;
- по способу адресации: последовательная и прямая;
- по функционалу: только вывод, только ввод, ввод и вывод;
- по наличию или отсутствию внутренних буферов.
Клавиатура, мышь – устройства, безусловно, медленные, что согласуется со скоростью действий человека.
Жесткие диски, графические карты – быстрые устройства. Прочие устройства можно отнести к промежуточным скоростям, вплоть до низких (струйные принтеры, плоттеры и подобные).
Устройства разделяются по способу доступа к данным. Их два: устройства с прямым методом доступа (Random Access Method; присутствие в названии слова “Random” не является удачным, т.к. никакой “случайности” в доступе к устройству не имеется, потому в советской литературе применялось слово “прямой”, что значительно ближе к сути), и устройства с последовательным методом доступа.
Прямой метод доступа, безусловно, присущ оперативной памяти. Однако ОП обычно не относится к устройствам ввода-вывода (за исключением виртуальных дисков, создаваемых в ОП).
Были разработаны дисковые накопители, адресация в которых формировалась из трех составляющих: номера цилиндра, номера дорожки в цилиндре и номера записи на дорожке.
Дисковое устройство представляет из себя надстроенные друг над другом диски с магнитным покрытием:

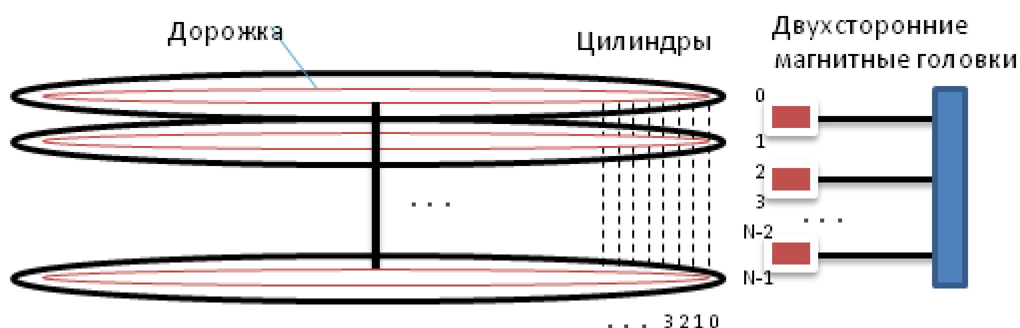


Рис. 25. Структура магнитного диска.

Каждый диск на магнитной поверхности (две внешние пластины с наружной стороны таких покрытий не имеют) имеет исчисляемое сотнями количество окружностей – магнитные дорожки. На каждой дорожке располагаются магнитные записи, которые, как правило, разбиваются на участки фиксированной длины, называемые кластерами, секторами, блоками.
Окружности (дорожки), расположенные друг над другом на пластинах, образуют цилиндр. Нумерация дорожек ведется от нуля с первой до последней в цилиндре (вертикальные штрихованные линии на Рис. 25). Спаренные магнитные головки пронумерованы так же. Потому адрес в диске формируется как тройка (Cyl, Head, Record) – номер цилиндра, номер головки,

номер записи на дорожке. Начало записи на дорожке (или начало дорожки) обозначается с помощью специальных маркеров.
Комплекс головок имеет возможность вдвигаться на всю глубину до центра пластин, от 0-ого цилиндра до последнего.
Диск разгоняется до больших скоростей так, что создается воздушная подушка между головкой и поверхностью пластин, предохраняя их от физического контакта. Выдвинувшись по адресу до указанного цилиндра, указанная номером головка считывает запись с дорожки или записывает на дорожку в запись. Именно потому такой принцип прямой установки на нужный цилиндр и работа соответствующей головки был назван прямым методом доступа.
Дорожки и записи на дисках могут повреждаться или уже быть поврежденными сразу после производства в допустимых количествах. Потому, при форматировании (разметка магнитных записей) дисков оставляются резервные цилиндры, которые могут быть использоваться для переадресации испорченной дорожки или блоков записей на резервную дорожку. Такая процедура осуществляется средствами диагностики ОС или независимыми утилитами.
Естественно, информация о переадресации фиксируется в памяти дискового устройства или в контроллере. Т.е., при адресовании (c_1, h_1, r_1) будет выбрана (c_2, h_2, r_2).
Время движения головок от первого цилиндра до последнего у современных дисков измеряется, примерно, в диапазоне от 5 до 15 мсек. Резервные цилиндры в количестве от 20-ти и более расположены в числе последних, как и время доступа к ним, по вполне очевидным причинам.
Как правило, 0-ая запись 0-ой дорожки 0-ого цилиндра содержит сектор загрузки операционной системы (см. раздел о загрузке ОС).
Заметим, что даже при прямом доступе для дисков чтение или запись, как чтение и запись для файлов, осуществляется последовательным способом.
Иным способом адресации является последовательный метод доступа, название которого говорит само за себя, и применяемый на информационных носителях с последовательным расположением данных, и в которых доступ к данным осуществляется через их номер расположения в последовательности. Без прохода по предыдущим данным к требуемой информации не добраться. К классическим примерам можно отнести сильно устаревшие перфоленты, магнитные ленты (используются до сих пор) и подобное.
Аналогично при конструировании данных: в массивах – прямой доступ по индексу: array[i] – прямое указание элемента; в списках – последовательный доступ, пробегая по указателям от начала списка к требуемому элементу.
В программировании мы имеем аналоги. К примеру, массивы с указанием и немедленным доступом к элементам массива, или списки, которые требуют последовательного продвижения вправо или влево (для двунаправленных списков; для однонаправленных списков продвижения осуществляются только в одну сторону) по элементам до искомого (с целью ускорения работы со списками иногда используют карты списков – массивы с указателями на элементы списков, превращая последовательный метод в прямой).
Ну и совсем очевидный пример: оперативная память с адресованием и прямым доступом к ячейкам памяти или к словам выборки.

6.2. Файловые структуры.

В операционных системах каждое устройство представляется символическим образом, а его содержимое, если это носитель наборов данных, представляется в виде древовидной логической структуры, которая отражает размещение совокупностей и отдельных файлов на носителе.

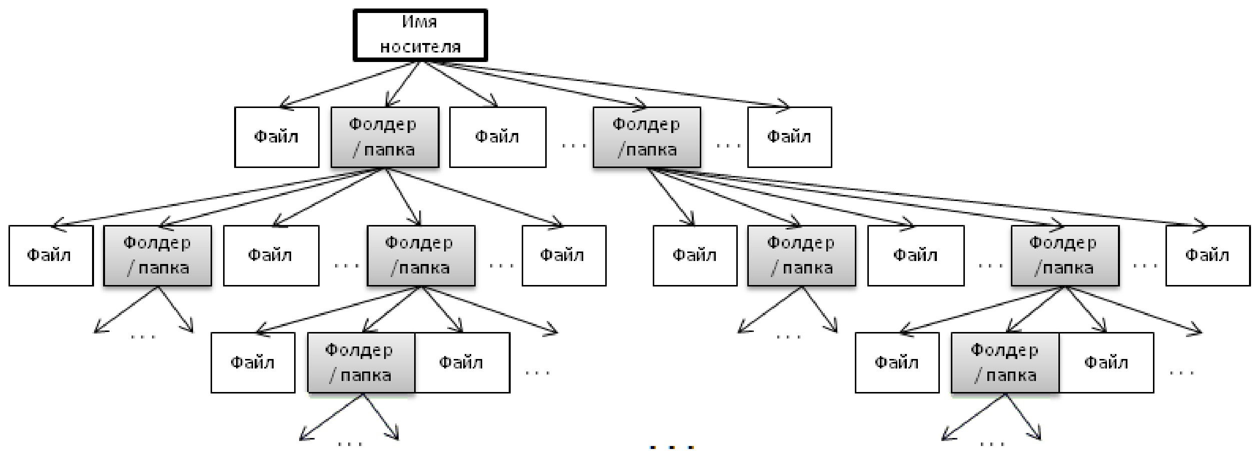


Рис. 26. Древовидная файловая структура.

Для множества носителей формируется либо граф - дерево, вершинами следующего за корнем уровня являются устройства, а уже следом – файлы (изначально – набор данных на внешнем носителе), либо граф – лес: совокупность деревьев.
Перечень графовых структур можно содержать либо в виде списков, либо в таблицах, в которых каждая регистрирующая запись для файла имеет поле указания на фолдер, которому принадлежит данный файл; это же относится и к фолдерам; в некоторых ОС для обеих разновидностей используется термин "файл", составной или простой.
Ну, а далее с каждой вершиной в графе связывается структура или таблица, в которой, как правило, перечисляются
- тип файла: фолдер или простой файл;
- адрес на носителе;
- время создания;
- время последнего обновления;
- пользователь, осуществивший обновление;
- время последнего использования (отличать от обновления, т.к. файл может быть использован только на чтение);
- пользователи, которым разрешен доступ к файлу и режимы доступа;
- прочее, что определяется направленностью ОС и фантазией ее разработчиков.

Т.е. древовидная структура в виде размеченного направленного графа позволяет описывать вложенные друг в друга наборы данных, регистрировать и контролировать их использование.
Из важного отметим, что файл может записан как в виде непрерывной последовательности байтов, так и состоять из множества фрагментов в силу того, что носитель мог быть фрагментирован удалениями файлов, и необходимого непрерывного участка для записи файла не нашлось (актуален регулярный запуск процедуры дефрагментации для перезаписей файлов и создания больших свободных участков).
Фрагмент файла может быть представлен либо записью в таблице с указанием файла, продолжением которого является фрагмент, либо во фрагменте файла формированием указания на продолжение. Заметим, такие же методы могут использоваться и при размещении массивов данных в оперативной памяти.

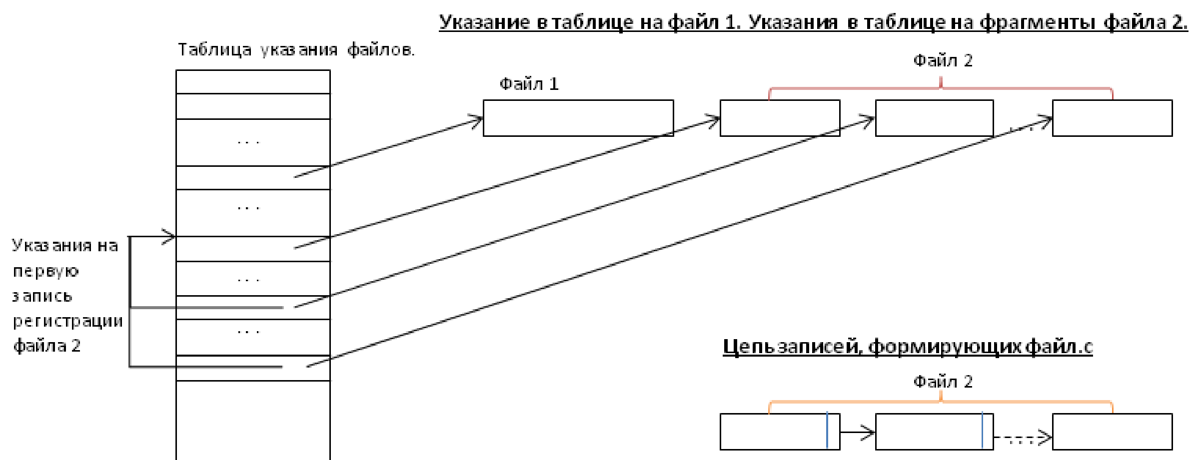


Рис. 27. Варианты указаний на фрагментированные файлы. В цепи записей фрагментов в каждом фрагменте формируется указание на адрес следующего фрагмента.

6.3. Системы ввода-вывода с буферизацией.

В зависимости от конкретных решений, часть устройств современных компьютеров
- передают или принимают по одному байту информации;
- часть устройств отправляют или принимают информацию от вычислителя в размерах его слова выборки.

Такие варианты связаны либо с упрощениями ввода-вывода, либо в тех случаях, когда получение или вывод информации требуют немедленной реакции как от программ компьютера, так и программного или микропроцессорного обеспечения со стороны устройств.
Устройства с внутренним буфером обеспечивают накопление информации в байтах внутренней памяти самого устройства и присутствуют, как правило, в довольно медленной периферии: принтеры, сканнеры и подобные. Так, на принтер с буфером можно отправить текст или изображение, которые могут быть приняты устройством в буфер, а уже позже быть автономно распечатаны.
Буферизация используется и для быстрых устройств (к примеру, некоторые дисковые накопители): минимизируется ожидание перемещений головок, в буфере HDD накапливается считываемая информация, и только после этого содержимое отправляется в ОП.
Буферизация на устройствах предназначена для минимизации количества операций чтения-записи, особенно в случаях, когда в них участвуют центральные процессоры с их отвлечением от основной задачи по выполнению программных кодов.
Помимо буферов в устройствах буферизация применяется в программных системах: организуются поля памяти в качестве буферов для отдельных программ. Здесь понятие “буфер” довольно условное, содержательное, согласуется с целями использования как участок памяти для промежуточных накоплений информации. Они не имеют какой-то особой физической реализации, только логически выделенные области памяти для реализации взаимодействий как с устройствами, так и для межпрограммных взаимодействий.
Контроль за заполнением, опорожнением буфера может быть организован создателем буфера. Часто буферы создаются приложениями, но передаются под контроль другим программным модулям (в частности, службам ОС). При этом надо помнить, что если буферы используются двумя и более параллельными процессами, где и кем бы они ни были организованы, необходимо учитывать решение проблем синхронизации работы и событий, связанных с буферами: факты создания, заполнения, переполнения и удаления.
Определенные проблемы начинаются тогда, когда буфер используется асинхронно.
К примеру, если буфер используется для асинхронного наполнения информацией, то возникает необходимость асинхронного же информирования о завершении операции, т.к. программа,

запросившая информацию посредством буферизации, может продолжать свою работу. Наиболее эффективным способом здесь является инициирование прерывания (события, в общем случае) со стороны процесса наполнения буфера или считывания из него. Как правило, эти обязанности возлагаются на программы службы ввода/вывода ОС. Однако такой же способ можно использовать и в обычных потоках: если операция асинхронного заполнения буфера была запрошена из одного потока к другому, при этом первый продолжил работу, то из второго потока по завершении загрузки буфера может быть создано событие с размещением в очереди событий первого. Даже если оба потока имеют разные очереди событий.

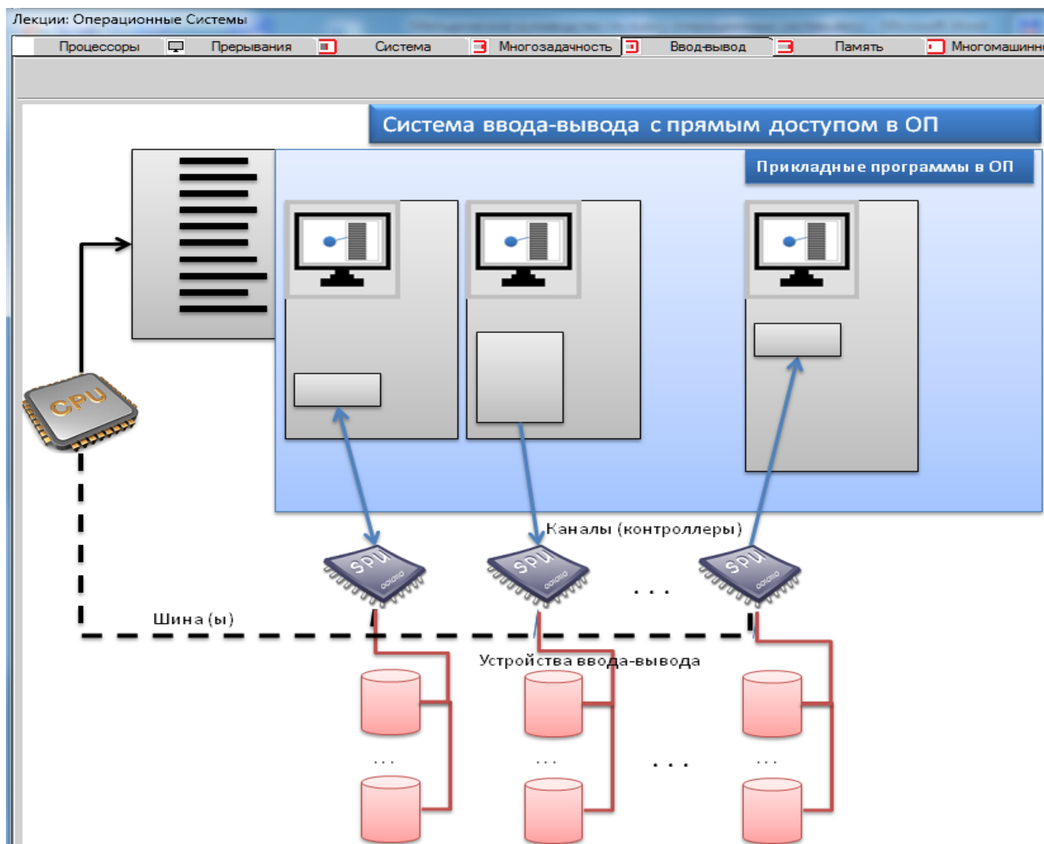


Рис. 28. Устройство ввода-вывода в вычислительных машинах.

На Рис. 15 представлен вариант считывания с устройств ввода/вывода непосредственно в память или запись на устройство из нее без участия центрального процессора, с помощью специальных подпроцессоров (обозначены на рисунке как SPU). Очевидно, что здесь следует позаботиться о защите участков памяти в случае записей в нее (да и чтение из чужих областей не всегда допустимо и может регулироваться специальными флагами разрешения чтения/записи).
Ввод-вывод с буферизацией появился, в первую очередь, как средство минимизации количества операций непосредственно с устройствами (накопить порциями выводимое в буфере, и в случае его заполнения или командой вывести одноразовым действием), организации асинхронных операций В/В без задержек на ожидание их завершения.
Заполнение и разгрузка буферов, как правило, сопровождается прерываниями (в общем случае – событиями ОС), если буферы обслуживаются операционными системами или имеют аппаратную поддержку, что способствует правилам асинхронной работы с ними.
Буферы создаются и используются и для целей промежуточного накопления и хранения информации. К примеру, при работе с изображениями, расположенными в оперативной памяти. Отметим несколько характеристик, связанных с отображением видеоинформации на мониторы. В первую очередь – это специальный процессор (видеопроцессор), обладающий собственной памятью. Современные графические процессоры – это, как правило, т.н. GPU (Graphics Processing

Unit), являющиеся матричными процессорами, относящимися к архитектуре SIMD по Флинну. Видеопроцессоры с определенной частотой сканируют видеопамять или ее фрагмент (объемы современной видеопамяти позволяют одномоментно хранить множество изображений установленного формата) и отображают ее содержимое на мониторе. Обычно, изображения представлены в формате RGB с последующей генерацией видеосигналов. GPU позволяют с большой скоростью проводить преобразования изображений непосредственно в собственной памяти (шейдинг, растеризацию и пр., относящееся к рендерингу).
Несмотря на высокие скорости преобразований, реализованных для графического процессора, ими, преобразованиями, задачи обработки изображений никак не исчерпываются. А потому их формирование часто осуществляется в программах, исполняемых на CPU.
В этой связи в промежуточных буферах можно хранить различные фрагменты изображений, хранить несколько изображений для последующей их композиции и манипуляций, и пр. Частое копирование фрагментов изображений из ОП в графическую память порождают известную проблему: “блинкование” (промаргивание) картинки на экране.
Формирование изображения в ОП (фактически, в буфере) и пересылка его завершенной версии в видеопамять решает данную проблему.
Файловые системы многих ОС построены на принципах копирования открываемых файлов в буферы (фрагментарно или целиком) в ОП и осуществление всех операций чтения/записи именно с содержимым буферов, а не прямо на устройствах (все с той же целью минимизации операций с устройствами). При этом операции записи на устройство имеют устанавливаемую задержку. По этой причине всегда рекомендуется отключать устройства через ОС (не выдергивать, к примеру, из портов USB), т.к. запись на устройство была осуществлена в буфер в ОП и по таймауту еще не была выгружена в реальный файл на устройстве. Т.е. условно записанные данные в буфер могут быть утеряны. При отключении устройства через службу ОС буфер принудительно выгружается на устройство без ожидания завершения таймаута.
Возможно установление нулевого таймаута, и тогда каждая операция записи сопровождается немедленной записью и на устройство. Что, конечно, приводит к замедлению работы программы, осуществляющей ввод/вывод.

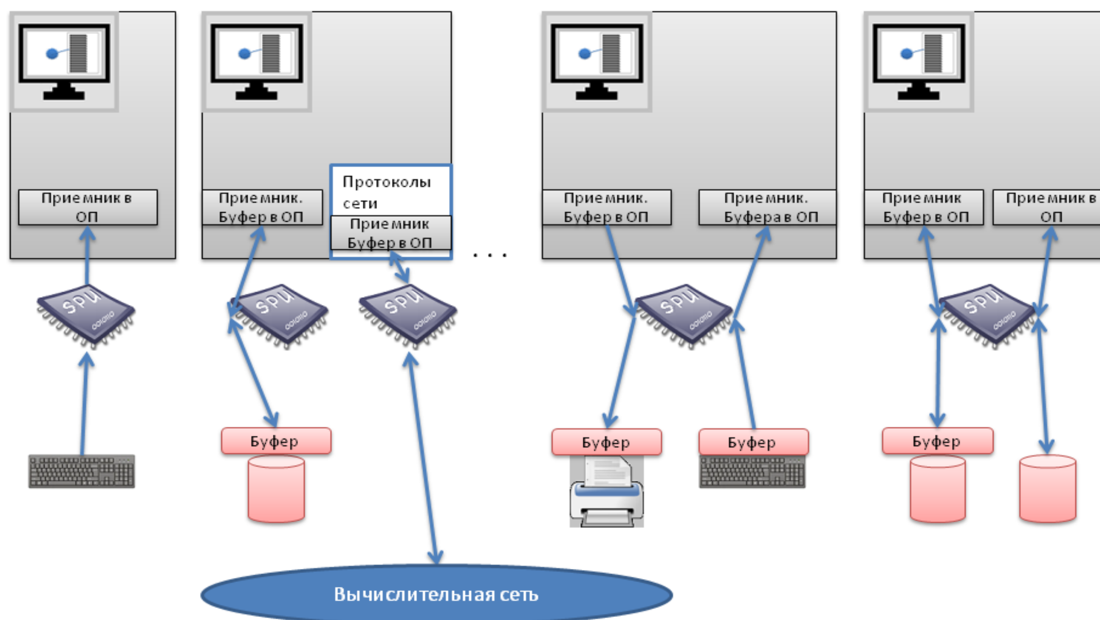


Рис. 29. Устройства ввода-вывода с буферами и без них. Буферы в ОП.

Глава 7. Многомашинные и многопроцессорные комплексы.

Идея соединить два и более вычислителя является, практически, ровесницей появления компьютеров. Поначалу средствами соединения являлись специальные кабели – шины с возможностью передачи цифровой информации. Позже стали проводится эксперименты и разработка аналого-цифровых и цифроаналоговых преобразователей. Последние дали возможность использовать распространенные в те времена аналоговые средства связи (телефонную, телеграфную связь, радиоканалы и пр.).
Взаимодействующие вычислители позволяли, передавая информацию друг другу, обеспечивать более надежную ее сохранность (хранить на двух компьютерах надежнее, чем на одном), передавать, в том числе, коды программ, обеспечивать дублирование выполнения вычислений, их сверку и синхронизацию (первые поколения вычислителей были не очень надежны, сопровождались расчетами вероятностей на сбой на основе собираемых статистических данных).
Но главная цель создания многопроцессорных и многомашинных ассоциаций – распределение вычислений с возможностью их ускорения, а также поддержка параллельных программ, реализующих модели совокупностей объектов с независимым или частично зависимым поведением. Множество исполнителей различных вычислений, безусловно, приводит не только к их ускорению, но и к упрощению алгоритмизации и программирования.

7.1. Формирование многомашинных и многопроцессорных систем.

Наиболее продуманной и реализованной она стала уже в концепции IBM/360.
Так, были осуществлены следующие типы соединений:

1. Процессор – процессор;
2. Канал – канал.
3. Общее устройство и носитель данных.
4. Сетевое соединение.
5. Телекоммуникационные соединения.

В первом случае процессоры соединяются посредством кабеля-шины. В системе команд присутствуют команды инициирования межпроцессорной связи и обмена данными по соединению.

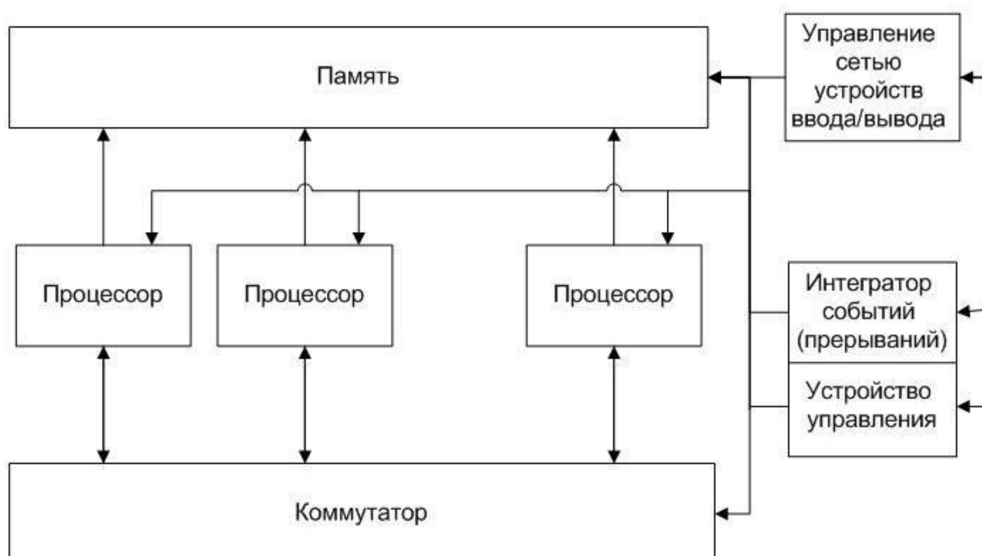


Рис. 30. Многопроцессорная архитектура с общей памятью и централизованным управлением.

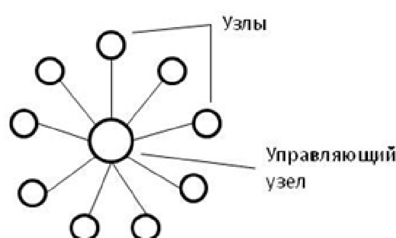
Позже были разработаны коммуникаторы, позволяющие соединять не только процессоры, но и устройства периферии, все или выделенные.
Соединения "канал-канал" являются средством подключения одного вычислителя в качестве устройства ввода-вывода для другого. Последующая работа осуществляется через установки "Master/Slave" – главенствующей машины и подчиненной, управляющей и управляемой.
Соединение через общий носитель данных осуществляется присоединением устройства ввода-вывода к двум и более вычислителям. Как правило, таким устройством выступает накопитель на жестком диске как наиболее быстрое устройство. Аналогом подобного являются, в частности, т.н. "рейд-массивы". Взаимодействие при подобном соединении происходит посредством записей на носитель и их чтением разными участниками.
С появлением вычислительных сетей (средства коммуникации на основе общепринятых протоколов – правил организации взаимодействий) соединения в необходимых случаях осуществляются через них.
Наконец, телекоммуникационные соединения осуществляются посредством телекоммуникационных станций, относящихся к периферии отдельного вычислителя. Т.е. вычислительные установки взаимодействовали через соединения (проводная связь, радиосвязь, оптическая связь и возможное др.) между телекоммуникационными устройствами, расположенными на значительных расстояниях друг от друга.
Управление каждым видом соединения осуществляется посредством соответствующей компоненты ОС, ее службы.
В целом, все виды соединений могут осуществляться как с разделениями на управляющие и подчиненные машины, так и без выделений таковых, в режиме равноправных, "договаривающихся" о взаимодействии вычислителей.
Ну, а в логической основе взаимодействий должен лежать общий для всех участников язык и его однозначная интерпретация. Позже эта совокупность выразилась в понятии протоколов: объединении символизма, общих и понятных структур данных и единых правил реагирования (интерпретации) перечисленного.
Изначально многомашинные комплексы стали подразделяться на однородные (гомогенные), т.е. состоящие из вычислителей с одинаковой архитектурой и одинаковыми операционными системами, и неоднородные (гетерогенные), т.е. имеющие в своем составе вычислители разных архитектур (не обязательно) с разными операционными системами. И здесь, пожалуй, главенствующую роль играет операционная система, т.к. собственно взаимодействие, обмен информацией и поддержание самого процесса вычислений осуществляется именно под управлением ОС. Одинаковость архитектур связана с возможностями исполнения одного и того же машинного кода (иначе пришлось бы осуществлять компиляцию и редактирование связей для различных ассемблеров, что, безусловно, является усложнением).
Присутствие различных операционных систем влечет дополнительные проблемы как по организации их согласованной работы, так и по прикладным задачам, т.к. каждая разновидность ОС может иметь те или иные особенности по использованию ее функционала, что потребует адаптации к нему и внутри прикладных задач.
Отметим, что появление виртуальных машин – систем, устанавливаемых на разные платформы (к примеру, POSIX), но ведущие совершенно одинаково, является определенным решением проблем разнородности.
Еще в 1980-е в IBM была разработана ОС Виртуальных машин, которая на сегодняшний день стала основной и имеющей в своем составе варианты UNIX-а, Linux-а, OS-2 и др.
Работа под виртуальной машиной с очевидностью несколько медленнее (насколько медленнее зависит от логики виртуальной ОС, количества промежуточных этапов между базовой ОС и виртуальной, некоторых других технических деталей), чем под базовой ОС: так или иначе виртуальная ОС будет работать с аппаратной частью через прослойку ядра базовой ОС (в системе может присутствовать несколько виртуальных машин, которые, конечно же опираются на собственные ядра, подходы к обработке прерываний и пр.).
Примером довольно широко распространенных многомашинных вычислительных систем являются кластеры – однородная совокупность вычислителей, объединенных специальной

вычислительной сетью, которая, в силу свой узкой направленности (быстрое распределение и обмен сообщениями компьютерами) значительно упрощена в части протоколов: здесь не нужны все семь стандартизованных уровней в силу ограниченности функционала.
Кластеры, по сути, относятся к архитектурам SPMD (Single Procedure Multiple Data – единственная процедура для множества данных) по расширению классификации Флинна (Single Instruction Multiple Data): код одной задачи (приложения) распространяется по всем выделенным для задачи узлам кластера, а далее фрагментами задачи обрабатываются различные данные.
Качественный скачок концепция кластеров получила с появлением персональных компьютеров, которые были значительно дешевле, и которые позволяли создавать большие ассоциации компьютеров, соединенных вычислительной сетью. Таким образом, кластерами стали называть однородные структуры соединенных компьютеров, управляемых одинаковыми операционными системами. Очевидно, что кластеры значительно менее производительны, чем многопроцессорные системы. Если в многопроцессорных системах межпроцессорные коммуникации образованы внутренними соединениями или средствами быстрых коммутаторов (общая системная шина, прямые соединения, транспьютеры и подобное), то в кластерах соединения построены на протоколах специализированных вычислительных сетей. Даже некоторые интеграции и упрощения в реализациях уровней протоколов не спасают ситуацию со скоростями обмена данными: отличия с многопроцессорными системами могут составлять несколько порядков.
Как правило, кластеры имеют централизованное управление.
Ярким примером кластеров является Titan с 18 688 узлами, и построенный на гибридной платформе Cray XK7. Гибридность архитектуры заключается в том, что 16-ядерные процессоры AMD Opteron интегрированы с GPU NVIDIA Tesla K20x. Таким образом, разработчики Titan обеспечили эффективное аппаратное исполнение как операторов с плавающей точкой, так и 32- и 64-битовых целочисленных операций на Opteron-ах, и матричных операций на GPU. Распределение исполняемого кода по тому или иному виду процессоров определяется на этапе компиляции.
Отметим, что на данный момент Titan в рейтинге производительности TOP-500 отошел с 1-ого места в 2012 г. на 2-ое место китайским кластером Тяньхэ-2 на модификациях процессоров Intel Xeon.
У обоих кластеров программное управление построено на модификациях ОС Linux. Выбор Linux в качестве ОС узлов многомашинных систем сопряжен вовсе не с какими-то особенными достоинствами самой ОС, а в первую очередь, ее, ОС, дешевизной или и вовсе бесплатностью. Немалую роль здесь играет значительная простота и открытость исходных кодов.
Наиболее распространенной платформой суперкомпьютеров из списка TOP-500 является Blue Gene производства компании IBM (существует множество модификаций и поколений). В основе платформы Blue Gene/Q – процессоры PowerPC A2 с 18-ью ядрами (одно ядро отвечает за управление, второе обеспечивает надежность, а 16 ядер используются непосредственно для вычислений).
На узлах Blue Gene размещается ОС CNK (for Compute Node Kernel), а для поддержки операций ввода-вывода на соответствующих ядрах размещается модификация ядра Linux INK (for I/O Node Kernel).

7.2. Структуры управления в многомашинных комплексах и задачи ОС.

Одной из принципиальных задач в многопроцессорных и многомашинных системах является организация управления вычислительным комплексом. Безусловно, при построении систем управления разработчики ориентируются на архитектуру вычислителя (но не только на нее; в отдельных случаях разрабатывается слой поддержки различных моделей организации вычислений).
Ключевым здесь является выбор между централизованным и распределенным управлением.

На Рис. представлена логическая схема централизованного управления (звездообразная структура) без указания способов физического подключения участвующих компьютеров (это могут быть отдельные шины, специальные каналы, вычислительная сеть; соединения могут быть разнородными). Центральная машина (узел) интегрирует управляющие действия по распределению задач, передаче информации, отслеживанию состояний периферийных узлов, интеграции событий, касающихся всего комплекса, включению или исключению отдельных узлов, и пр.



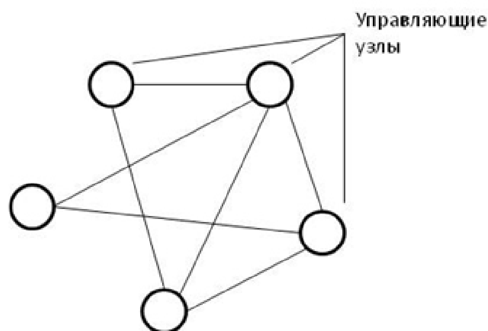


Рис. 31. Схемы с централизованным и распределенным управлением.

Системы с распределенным управлением включают в себя равноправные по функционалу и возможностям управления узлы. Так, каждый из узлов имеет возможность инициировать задания для других, в то время, как такие задания некоторыми узлами могут быть отклонены. Взаимодействия между узлами организуются через диалог и установление режима работы.
Качественным отличием между указанными двумя способами управлением являются те факты, что при централизованном управлении управляющий узел имеет распорядительные (императивные) права (периферийный узел обязан выполнить задание центрального), периферийные узлы, как правило, заняты только заданиями, поступившими от центрального узла, в то время, как при распределенном – посредством обращений за возможностью выполнить то или иное действие, не имеющих статуса распоряжений (т.е. узел на основе собственных, как правило, эвристических правил может отказать).
Каждый узел может иметь подключенные и подчиненные ему периферийные вычислители (т.е. быть вариантом хост-компьютера, и не обязательно в статусе сервера).
В качестве узла могут выступать вычислительные установки любой сложности, в том числе, отдельные кластеры. Важен статус узла в задачах управления всей системой.
Главным преимуществом при централизованном управлении является снятие задач управления с периферийных машин и скорость при распределении заданий.
Системы с распределенным управлением являются (в силу равноправия и наличия средств управления на всех узлах) более гибкими и живучими: выход из строя любого узла или средств связи с ним не приводит к каким-либо катастрофическим последствиям, т.к. многомашинный комплекс просто останется без одной машины. При этом выход из строя средств связи может быть компенсирован наличием альтернативных соединений.
В то же время выход из строя управляющего узла при централизованном управлении приведет к остановке работы всего комплекса. Эти случаи, как правило, компенсируются расположением компьютеров (одного, двух), дублирующих работу центрального на принципах “зеркала”, и которые автоматически подключаются к управлению вычислительным комплексом в случае выхода из строя управляющего узла.
В целом, многомашинные комплексы применяют резервирование компьютеров для подключения в общую систему при выходе из строя узлов и/или управляющих машин.
Очевидно, что вызов для параллельного исполнения того или иного машинного кода [5]) сопряжен с распределением всего исполнимого кода по вычислительным узлам. [6]) Передавать код от узла к узлу посредством механизма сообщений или аналогичный является нерациональным, т.к. загружает сеть соединений, а размеры файлов исполнимого кода могут быть очень велики.

По этой причине наиболее распространенным является способ считывания исполнимого кода каждым из узлов с общего внешнего носителя (сервера), на который исполнимая программа размещается заблаговременно.
Заметим, что в указанной выше логике построены и современные т.н. “графические процессоры” GPU (наиболее известны производители NVIDIA и AMD). По существу, они имеют построение матричных процессоров, содержащих множество процессоров исполнителей и центральный процессор, осуществляющий управление ими и прочими ресурсами: внутренней памятью, взаимодействием с CPU перекачкой информации из собственной памяти к оперативной CPU, и обратно.

[5]) Не описываем экзотику вроде загрузки исходного кода программы, его компиляцию, редактирование и последующее исполнение, в первую очередь, потому, что это сопряжено с чрезмерным расходом ресурсов кластеров, занятостью их узла(ов) техническими операциями, в то время, как основное предназначение многоузловых вычислительных комплексов – выполнение сложных задач для множества пользователей. Компиляция и редактирование связей вполне осуществимы на клиентских машинах.

[6]) Языки программирования, позволяющие модульное построение программ, предполагают и модульное построение исполнимого кода в отдельных загружаемых файлах. Отчасти, то же самое можно сказать о динамически загружаемых библиотеках, создание и/или использование которых поддерживается, практически, всеми распространенными языками программирования.
Язык параллельного программирования Caper устроен по модульному принципу. Его виртуальная машина позволяет динамически загружать модули с объектным кодом, динамически связывать их и исполнять. Так же поддерживается и работа с динамическими библиотеками ОС. В этом связи, Caper позволяет размещать модули на сервере для использования отдельных из них на том или ином узле. Т.е. формирования некоего целостного кода, реализующего все приложение, не требуется. Возможна и передача модулей с узла на узел, т.к., как правило, объектные модули языка имеют небольшой размер.

7.3. Организация общей памяти в многомашинных комплексах.

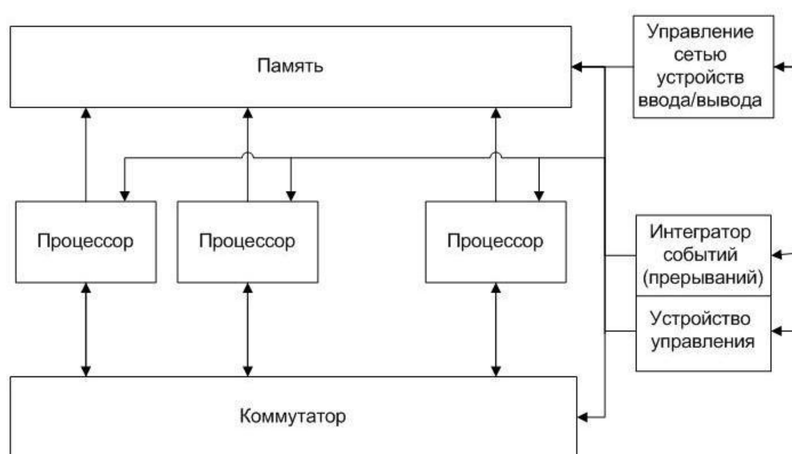


Рис. 32. Архитектуры с общей памятью.

Отметим, что чтение и, особенно, размещение данных от множества процессоров/ядер в общую ОП регулируется аппаратными средствами, требует эффективных методов синхронизации процессов при работе с памятью (на Рис. не отображено). Коммутатор обеспечивает управление

процессорами и межпроцессорное взаимодействие. Интегратор событий необходим при наличии у процессоров собственной системы прерываний.
Как правило, для эффективной работы многопроцессорного вычислителя (позже эта методология перешла, в частности, к кластерам) исходят из необходимости общего управляющего узла. Такая схема особенно необходимо при организации вычислений класса SIMD|SPMD, к которым относятся процессоры GPU и прочие матричные процессоры. Они выполняют роль умножителей и распространителей операций с различными данными. Так, для процессоров NVidia указываемая функция итерации операций передается на устройство управления (тоже процессор) для распространения инструкций тела итерации по процессорам всего устройства. В качестве средства программирования управления в NVIDIA был создан язык Cuda. Ту же роль играет программная платформа OpenCL, которая стандартизирована для кроссплатформенных приложений и поддержана, практически, всеми производителями программного обеспечения.

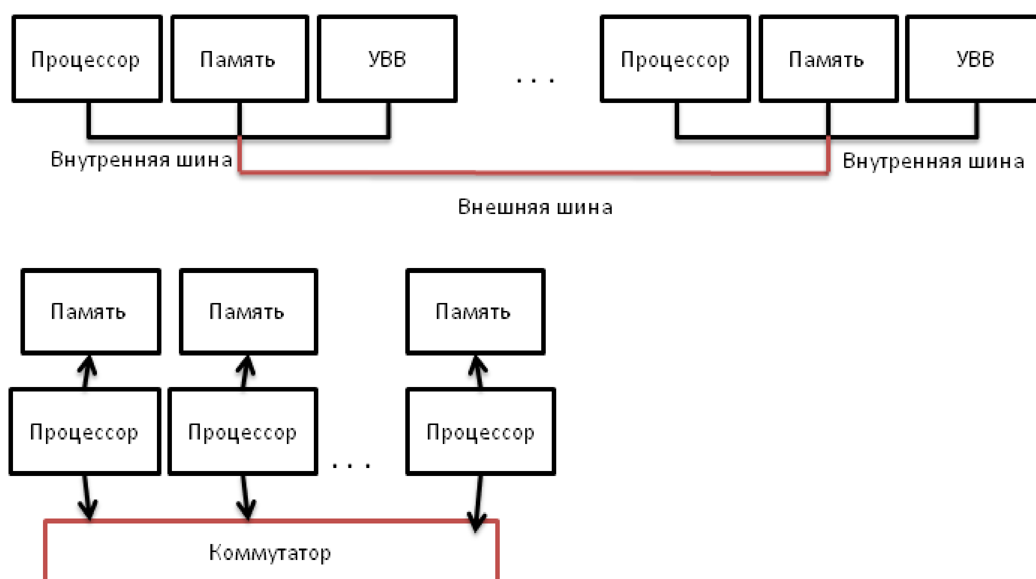


Рис. 33. Архитектуры с разделенной памятью.

Заметим, что элементы оперативной памяти тоже могут быть снабжены коммуникатором. В таких случаях каждый и выделенные процессоры могут обращаться не только к “своей” памяти, но и к чужой. Здесь часто используется сквозная адресация (нумерация) байтов или слов памяти.
Одной из актуальных проблем при использовании многомашинных и многопроцессорных комплексов с разделенной памятью (типа Motorola 68 0xx; см. Рис.33) является необходимость создание общей оперативной памяти для поддержки общих переменных и пр. структур данных для распределенных по процессорам или машинам вычислений.
Одним из способов является создание сквозной нумерации выделенных элементов памяти во всех вычислителях. Если под общую память у компьютера с номером **a** выделяется память объемом **L**, то получив число – адрес мы имеем алгебраический вычет **A** = **a*****L** + **b**. По существу, задав адрес ячейки общей памяти, мы получаем номер компьютера и смещение в выделенном на нем участке. В целом, этот способ схож с адресацией в виртуальной памяти, только с указанием еще и компьютера.
Доступ и обращение к конкретному элементу осуществляется с помощью функции–монитора, управляющей созданной памятью как на запись, так и на чтение. Возможна более сложная организация, в частности, подобная управлению памятью внутри одного компьютера.
Иным способом является выделение, условно, одного компьютера в качестве хранилища общих данных в его оперативной памяти. Переменные вычисления организуются с помощью обращений к программе-монитору хранилища (как вариант, переменные представляются символическими именами, а их учет формируется парами (строка имени, адрес расположения в общей памяти)).

В целом, организация общей памяти на внешних носителях ничем особым, кроме некоторых технических нюансов, не отличается от организации в ОП. В данном случае данные организуются и хранятся в файлах.
Сами же операции в общей памяти инициируются сообщениями, содержащие инструкции.
Наконец, возможен еще один вариант создания общей памяти с выделением отдельного компьютера с соответствующим программным управлением, который отвечает для создания, сохранения, изменения и передачу “переменных” (с соответствующей сигнатурой) – полей памяти в собственной ОП, и создаваемых по требованию прочих компьютеров многомашинного комплекса и фиксируемых в переменных в символическом представлении. Обращение к переменной транслируется в функцию обращения к выделенному компьютеру выполнить необходимую функцию: записать передаваемое значение, считать (и передать запрашивающему) текущее значение “переменной”, создать или удалить ее.
В качестве заключения к данной главе отметим, что всякие соединения с обращениями по ним, безусловно, требуют времени и замедляют работу процессов, исполняемых на процессорах внутри вычислителей. То же касается и коммуникаций внутри одной машины, в том числе, между процессором и его ОП. Однако, эти затраты многократно компенсируются в обратных случаях, когда все вычисления ведутся внутри единственной вычислительной машины.

Заключение.

Данный текст формировался на основе курса по операционным системам, проводимому в ИПИиА НАН Республики Армения.
В данном тексте осуществлена попытка охватить основные архитектурные особенности вычислительных установок, их ресурсную базу и методы решений управления ими при наличии множества вычислительных процессов.
Особое внимание уделено событийным механизмам вычислителей, схемам обработки прерываний и событий в целом.
Основное внимание уделено операционным системам – программным комплексам, осуществляющими распределение ресурсов между прикладными задачами, реализуемыми вычислительными установками.
Рассмотрены периферийные средства, синхронные и асинхронные методы ввода и вывода информации, способы хранения и доступа к информации.
Представлены устройство многомашинных комплексов, в частности, устройство кластеров, и способы организации управления ими.
Представлены некоторые проблемы программирования для тех или иных вычислителей и их операционных систем.

Литература

1. Алгоритмы, математическое обеспечение и архитектура многопроцессорных вычислительных систем. М: “Наука”, 1982, 336 с.
2. Карп Р.М., Миллер Р.Е. Параллельные схемы программ. Москва, “Мир”: Кибернетический сборник. Вып. 13. 1976. сс. 5-61.
3. Flynn M. J. Very high speed computers. Proc IEEE, 1966, 54. — P. 1901—1901.
4. Джермейн К. Программирование на IBM/360. Пер. с англ. Изд. 2, 1973. 870 с.
5. Документация по zOS: https://www.ibm.com/docs/en/zos

6. М.Б. Кузьминский. Z-архитектура. https://www.osp.ru/os/2001/10/180514
7. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. СПб.: Питер, 2015, 1120 с.
8. Стивенс У. Р., Раго С. UNIX. Профессиональное программирование. 3-е изд. СПб.: Питер, 2018, 944 с.
9. Устройство графических процессоров. https://club.dns-shop.ru/blog/t-99-videokartyi/122040-ustroistvo-graficheskih-protsessorov-gpu/?utm_referrer=https%3A%2F%2Fwww.google.com%2F
10. Digital Signal Processing Solutions. Texas Instruments Incorporated, 2000.
11. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. СПб.: Питер, 2001, 752 с.
12. Вартанов С.Р. Язык программирования CAPER. - Киев, 1997 (Препр. 97-5, Национальная Академия Наук Украины, Институт Кибернетики им. Глушкова).
13. Vartanov S.R. On Parallel Programming Language Caper. Lect. Notes in Computer Sci., HCPN-2001, p. 501-503.