**Раздел 5. Программное обеспечение информационных технологий**

**Глава 1. Общие сведения о программном обеспечении**

* 1. **Основные понятия и определения**

Совокупность программ регулярного применения и сопровождающей их документации, предназначенная для решения задач пользователей на компьютерной технике, называется *программным обеспечением* (ПО). Программное обеспечение в англоязычной литературе (сейчас и повсеместно) принято называть словосочетанием software. Этот термин является игрой слов и может быть дословно переведен как “мягкий товар”. Однако термин закрепился, о чем свидетельствует использование его составных частей в названиях известных компаний и фирм, производящих и распространяющих ПО, как зарубежных, так и отечественных (Microsoft, Netware, СофтБит, AutoSoft и др.).

Основной компонент ПО – *программа* – упорядоченная в соответствии с некоторым алгоритмом последовательность команд (инструкций) компьютера для решения задачи пользователя. Чаще всего образ программы хранится в виде [исполняемого модуля](http://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C) (отдельного [файла](http://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) или группы файлов).

*Пользователь* – лицо, заинтересованное в решении некоторой задачи средствами вычислительной техники. По отношению к программному обеспечению компьютерные пользователи делятся на следующие группы:

* системные программисты, занимающиеся разработкой, эксплуатацией и сопровождением системного программного обеспечения (см. ниже);
* прикладные программисты. Выполняют разработку и отладку программ решения задач из различных прикладных сфер деятельности пользователей;
* конечные пользователи. Используют прикладное программное обеспечение для решения задач в своей повседневной деятельности. Различаются по уровню своей подготовки в части знания и использования компьютерной техники.
* администраторы. Как правило, это высококвалифицированные компьютерные специалисты, отвечающие за работу вычислительной сети, баз данных, корпоративной информационной системы в целом, безопасность и защиту данных. Могут иметь определенную специализацию: управление сетевым каталогом, политикой учетных записей, политикой аудита и т.п.

*Задача* (problem, task) – проблема, подлежащая решению в интересах пользователя.

Термин “задача” в программировании означает единицу работы вычислительной системы, требующую выделения вычислительных ресурсов (процессорного времени, оперативной и внешней памяти, файлов и т.п.).

*Приложение* (application) – программная реализация решения задачи на компьютере. Приложение может состоять из одной или нескольких взаимосвязанных и взаимодействующих программ.

Принято (весьма условно) делить программы на небольшие (простые), средней сложности и большие.

Программа считается “небольшой”, как по размерам, так и по другим признакам, если она удовлетворяет следующим признакам:

1) решает одну четко поставленную задачу в хорошо известных ограничениях, к тому же, не очень существенную для какой-либо практической или исследовательской деятельности;

2) неважно, насколько быстро она работает;

3) ущерб от неправильной работы программы практически нулевой (за исключением возможности обрушения ею системы, в которой выполняются и другие, более важные задачи);

4) не требуется дополнять программу новыми возможностями, практически никому не нужно разрабатывать ее новые версии или исправлять найденные ошибки;

5) в связи со сказанным выше не очень нужно прилагать к программе подробную и понятную документацию – для человека, который ею заинтересуется, не составит большого труда понять, как ею пользоваться, просто по исходному коду.

Сложные или “большие” программы, называемые также программными системами, программными комплексами, программными продуктами, отличаются от “небольших” не столько по размерам (хотя обычно они значительно больше), сколько наличием дополнительных факторов. Эти факторы связаны с их востребованностью и готовностью пользователей платить деньги, как за приобретение самой программы, так и за ее сопровождение и даже за специальное обучение работе с ней.

Обычно сложная программа обладает следующими свойствами [ Кулямин]:

1) программа решает одну или несколько связанных задач, зачастую сначала не имеющих четкой постановки, настолько важных для каких-либо лиц или организаций, что те приобретают значимые выгоды от ее использования;

2) существенно, чтобы программа была удобной в использовании. В частности, она должна включать достаточно полную и понятную пользователям документацию, возможно, также специальную документацию для администраторов, а также набор документов для обучения работе с программой;

3) низкая производительность программы на реальных данных приводит к значимым потерям для пользователей;

4) неправильная работа программы наносит ощутимый ущерб пользователям и другим организациям и лицам, даже если сбои происходят не слишком часто;

5) для выполнения своих задач программа должна взаимодействовать с другими программами и программно-аппаратными системами и обеспечивать работу на разных платформах;

6) пользователи, работающие с программой, приобретают дополнительные выгоды от того, что программа развивается, в нее вносятся новые функции и устраняются ошибки. Необходимо наличие проектной документации, позволяющей развивать ее, возможно, вовсе не тем разработчикам, которые ее создавали, без больших затрат на обратную разработку (реинжиниринг);

7) в разработку программы вовлечено значительное количество людей (более 5-ти человек). Большую программу практически невозможно написать с первой попытки, с небольшими усилиями и в одиночку;

8) большая программа имеет намного большее количество ее возможных пользователей по сравнению с небольшими программами, и еще больше тех лиц, деятельность которых будет так или иначе затронута ее работой и результатами.

Примером большой программы может служить стандартная библиотека классов Java или C#, соответствующих систем программирования.

Строго говоря, ни одно из указанных свойств не является обязательным для того, чтобы программу можно было считать большой, но при наличии двух-трех из них достаточно уверенно можно утверждать, что она большая. На основании некоторых из перечисленных свойств можно сделать вывод, что большая программа или программная система чаще всего представляет собой не просто код или исполняемый файл, а включает еще и набор проектной и пользовательской документации.

Процесс создания программ можно представить как последовательность следующих действий:

1. постановка задачи;
2. алгоритмизация решения задачи;
3. программирование.

*Постановка задачи* (problem definition) – это точная формулировка требований (функциональных и нефункциональных), предъявляемых к работе программы, с описанием входной и выходной информации, и, возможно, описание подходов к решению задачи.

*Алгоритм –* точный набор [инструкций](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), описывающих порядок действий исполнителя (компьютера), от допустимых исходных данных для достижения результата [решения задачи](http://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B7%D0%B0%D0%B4%D0%B0%D1%87) за конечное время. В старой трактовке вместо слова “порядок” использовалось слово “последовательность”, но по мере развития параллельности в работе компьютеров слово “последовательность” стали заменять более общим словом “порядок”. Это связано с тем, что работа каких-то инструкций алгоритма может быть зависима от других инструкций или результатов их работы. Таким образом, некоторые инструкции должны выполняться строго после завершения работы инструкций, от которых они зависят. Независимые инструкции или инструкции ставшими независимыми из-за завершения работы инструкций, от которых они зависят, могут выполняться в произвольном порядке, параллельно или одновременно, если это позволяет используемые процессор и операционная система.

*Программирование* (programming) – теоретическая и практическая деятельность, связанная с созданием программ.

Разработка программных систем (ПС), т.е. программирование, имеет ряд специфических особенностей. Прежде всего, следует отметить некоторое противостояние: неформальный характер требований к ПС (постановки задачи) и понятия ошибки в нем, но формализованный основной объект разработки − программы ПС. Тем самым разработка ПС содержит определенные этапы формализации, а переход от неформального к формальному существенно неформален.

Разработка ПС носит творческий характер (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым эта разработка ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого ее конца.

Следует отметить также особенность продукта разработки. Он представляет собой некоторую совокупность текстов (т.е. статических объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является динамическим). Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств. Продукт разработки имеет и другую специфическую особенность: ПС при своем использовании (эксплуатации) не расходуется и не расходует используемых ресурсов.

* 1. **Технологии программирования**

В процессе разработки программных систем используются различные *технологии программирования.* В соответствии с обычным значением слова “технология” под технологией программирования (programming technology) понимается совокупность производственных процессов, приводящая к созданию требуемого ПС, а также описание этой совокупности процессов. Другими словами, технология программирования понимается здесь в широком смысле как технология разработки программных средств, включая в нее все процессы, начиная с момента зарождения идеи этого средства до создания необходимой программной документации. Каждый процесс этой совокупности базируется на использовании каких-либо методов и средств, например, компьютера (в этом случае речь идет о компьютерной технологии программирования).

В литературе имеются и другие, несколько отличающиеся, определения технологии программирования. Используется в литературе и близкое к технологии программирования понятие программной инженерии, определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств. Именно программной инженерии (software engineering) посвящена упомянутая работа [1.3]. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала.

В технологии программирования акцент делается на изучении процессов разработки ПС (технологических процессов) и порядке их прохождения − методы и инструментальные средства разработки ПС используются в этих процессах, их применение и образуют технологические процессы. Тогда как в программной инженерии изучаются различные методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей – эти методы и средства могут использоваться в разных технологических процессах (и в разных технологиях программирования).

Не следует также путать технологию программирования с методологией программирования. В технологии программирования методы рассматриваются “сверху” − с точки зрения организации технологических процессов, а в методологии программирования методы рассматриваются “снизу” − с точки зрения основ их построения. Например, в работе [1.9] методология программирования определяется как совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом.

В историческом аспекте в развитии технологии программирования можно выделить несколько этапов [ Гагарина].

1. Первый этап - *“стихийное”* программирование – отсутствие сформулированной технологии, когда программирование было, по сути, искусством. Этап охватывает период от появления первых ЭВМ до середины 60-х годов 20 века. Развитие программирования шло по пути замены машинных языков ассемблерами, а затем алгоритмическими языками (Fortran, Algol) и повторного использования подпрограмм, что повысило производительность труда программиста.

Стихийно использовалась разработка “снизу-вверх” – подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых потом пытались построить сложную программу. В начале 60-х годов 20 века разразился кризис программирования. Он выражался в том, что фирмы превышали все сроки завершения программных проектов и их стоимость. В результате многие проекты так и не были завершены.

2. Второй этап – *структурный подход* к программированию. Этот подход сложился в 60 – 70 годы 20 века и представлял собой совокупность рекомендуемых технологических приемов, охватывающих все этапы разработки программного обеспечения. В основе структурного подхода лежит декомпозиция сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. В отличие от используемого ранее процедурного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры.

Проектирование осуществлялось “сверху-вниз” и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов – метод пошаговой детализации. Поддержка принципов структурного программирования была заложена в основу процедурных языков программирования (PL/1, Algol-68, Pascal, C).

Появилась и начала развиваться технология модульного программирования, которая предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули. Практика показала, что структурный подход в сочетании с модульным программированием позволяет получить достаточно надежные программы, размер которых не превышает 100000 операторов [Л10 Гагарина]. Узким местом модульного программирования стали межмодульные интерфейсы, ошибки в которых трудно обнаружить по причине раздельной компиляции модулей (ошибки выявляются только при выполнении программы).

3. Третий этап – *объектный* подход к программированию. Сложился с середины 80-х до конца 90-х годов 20 века. *Объектно-ориентированное программирование* (ООП) определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности *объектов*, каждый из которых является экземпляром определенного типа (*класса*), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов осуществляется путем передачи сообщений.

Основное достоинство объектно-ориентированного программирования по сравнению с модульным программированием – более естественная декомпозиция программного обеспечения, которая существенно облегчает его разработку. Кроме того, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции. Это позволяет существенно увеличить показатель повторного использования кодов и создавать библиотеки классов для различных применений.

Развитие объектного подхода в технологии программирования привело к созданию сред визуального программирования. Появились языки визуального объектно-ориентированного программирования, такие как Delphi, C++ Builder, Visual C++, C# и т. д. Однако технология ООП имеет и недостатки. Главный из них – зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь, к ресурсам друг друга.

4. Четвертый этап – *компонентный* подход и CASE-технологии (с середины 90-х годов 20 века до нашего времени). Этот подход предполагает построение программного обеспечения из отдельных компонентов – физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через *стандартизованные двоичные интерфейсы*. В отличие от обычных объектов объекты-компоненты можно собирать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. В настоящее время рынок компонентов – реальность, поддерживаемая Интернетом и массовой рекламой и публикациями.

Основы компонентного подхода были разработаны компанией Microsoft, начиная с технологии OLE (*Object Linking and Embedding* – связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Ее развитием стало появление COM-технологии (*Component Object Model* – компонентная модель объектов), а затем ее распределенной версии – DCOM, на основе которых были разработаны компонентные технологии, решаются различные задачи разработки программного обеспечения.

Среди них следуют отметить OLE-*automation* – технологию создания программируемых приложений, обеспечивающую доступ к внутренним службам этих приложений. На основе OLE-automation создана технология *Active*X, предназначенная для создания программного обеспечения, как сосредоточенного на одном компьютере, так и распределенного. Безопасность и стабильная работа распределенных приложений обеспечивается еще двумя технологиями, заложенными в COM. Это – MTS (Multitier Distributed Application Sever- сервер многозвенных распределенных приложений) и MTS (Microsoft Transaction Server) – сервер управления транзакциями.

Компонентный подход лежит также в основе технологии CORBA (Common Object Request Bracer Architecture – общая архитектура с посредником обработки запросов объектов). Эта технология, реализующая подход, аналогичный COM, разработана группой компаний OMC (Object Management Group – группа внедрения объектной технологии программирования). Программное ядро CORBA реализовано для всех основных аппаратных и программных платформ и обеспечивает создание программного обеспечения в гетерогенной вычислительной среде.

Важнейшая особенность современного этапа технологии программирования - широкое использование компьютерных технологий создания и сопровождения программных систем на всех этапах их жизненного цикла. Эти технологии получили название CASE-технологий (Computer-Aided Software/System engineering – разработка программного обеспечения/программных систем с использованием компьютерной поддержки). Сегодня существуют CASE-технологии, поддерживающие как структурный, так и объектный, в том числе, компонентный подходы к программированию.

* 1. **Качество и характеристики программного обеспечения**

Рассматривая, оценивая и анализируя программные системы и отдельные программы, останавливаются на показателях качества программ и их основных характеристиках. *Качество* ПО - это совокупность свойств, определяющих полезность изделия (программы) для пользователей в соответствии с функциональным назначением и предъявлёнными требованиями. Характеристика качества программы - понятие, отражающее отдельные факторы, влияющие на качество программ и поддающиеся измерению.

*Критерий качества* - численный показатель, характеризующий степень, в которой программе присуще оцениваемое свойство. Критерии качества могут включать множество различных характеристик: экономичность, документированность, гибкость, модульность, надёжность, обоснованность, тестируемость, ясность, точность, модифицируемость, эффективность, легкость сопровождения и т.д.

Для измерения характеристик и критериев качества используют *метрики*. Метрика качества программ – это система измерений качества программ. Эти измерения могут проводиться на уровне критериев качества программ или на уровне отдельных характеристик качества. В первом случае система измерений позволяет непосредственно сравнивать программы по качеству. При этом сами измерения не могут быть проведены без субъективных оценок свойств программ. Во втором случае измерения характеристик можно выполнить объективно и достоверно, но оценка качества ПО в целом будет связана с субъективной интерпретацией получаемых оценок.

Каждое ПС должно выполнять определенные функции, т.е. делать то, что задумано. Хорошее ПС должно обладать еще целым рядом свойств, позволяющим успешно его использовать в течение длительного периода, т.е. обладать определенным качеством. Качество (quality) ПС − это совокупность его характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей [3.6]. Это не означает, что разные ПС должны обладать одной и той же совокупностью таких свойств в их наивысшей степени. Этому препятствует тот факт, что повышение качества ПС по одному из таких свойств часто может быть достигнуто лишь ценой изменения стоимости, сроков завершения разработки и снижения качества этого ПС по другим его свойствам. Качество ПС является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.

Совокупность свойств ПС, которая образует удовлетворительное для пользователя качество ПС, зависит от условий и характера эксплуатации этого ПС, т.е. от позиции, с которой должно рассматриваться *качество* этого ПС. Поэтому при описании качества ПС, прежде всего, должны быть фиксированы критерии отбора требуемых свойств ПС. В настоящее время критериями качества ПС (criteria of software quality) принято считать:

1) *функциональност*ь, способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПС;

2) *надежность* – устойчивость, точность выполнения предписанных функций обработки, возможность диагностики возникающих ошибок. Надежность (reliability) ПС − это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под отказом в ПС понимают проявление в нем ошибки. Таким образом, надежное ПС не исключает наличия в нем ошибок − важно лишь, чтобы эти ошибки при практическом применении этого ПС в заданных условиях проявлялись достаточно редко. Убедиться, что ПС обладает таким свойством можно при его испытании путем тестирования, а также при практическом применении.

3) *легкость применения,* это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПС и оценке полученных результатов, а также вызывать положительные эмоции определенного или подразумеваемого пользователя. Сюда следует также отнести дружественный интерфейс, контекстно-зависимую подсказку, хорошую документацию;

4) *эффективность* − это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов.

5) *сопровождаемость*, *модифицируемость* – это характеристики ПС, которые позволяют минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей, переходу на новые версии и т.п.;

6) *мобильность (многоплатформенность)* – независимость от технического комплекса вычислительных средств, операционной системы, сетевых возможностей, специфики предметной области задачи и т.д.;

7) *коммуникативность* - степень возможной интеграции с другими программами, обеспечение обмена данными между программами.

Во многих случаях функциональность и надежность являются обязательными критериями качества ПС, причем обеспечение надежности красной нитью проходит по всем этапам и процессам разработки ПС. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПС.

К основным характеристикам программ и программных систем относится *сложность* программной системы. При оценке сложности программ, как правило, выделяют три основные группы метрик:

1. метрики размера программ,
2. метрики сложности потока управления программ,
3. метрики сложности потока данных программ.

Оценки первой группы наиболее просты и, очевидно, поэтому получили широкое распространение. Традиционной характеристикой размера программ является количество строк исходного текста. Под строкой понимается любой оператор программы, поскольку именно оператор, а не отдельно взятая строка является тем интеллектуальным "квантом" программы, опираясь на который можно строить метрики сложности ее создания.

Непосредственное измерение размера программы, несмотря на свою простоту, дает хорошие результаты. Конечно, оценка размера программы недостаточна для принятия решения о ее сложности, но вполне применима для классификации программ, существенно различающихся объемами. При уменьшении различий в объеме программ на первый план выдвигаются оценки других факторов, оказывающих влияние на сложность. Таким образом, оценка размера программы есть оценка по номинальной шкале, на основе которой определяются только категории программ без уточнения оценки для каждой категории.

К группе оценок размера программ можно отнести также и метрику Холстеда. Основу метрики Холстеда составляют четыре измеряемых характеристики программы:

n1 - число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);

n2 - число уникальных операндов программы (словарь операндов);

N1 - общее число операторов в программе;

N2 - общее число операндов в программе.

Опираясь на эти характеристики, получаемые непосредственно при анализе исходных текстов программ, М. Холстед вводит следующие оценки:

словарь программы n1=n1+n2,

длину программы N=N1+N2, (1)

объем программы V=N\*log2(n) (бит). (2)

Под битом подразумевается логическая единица информации - символ, оператор, операнд.

Далее М. Холстед вводит n\* - теоретический словарь программы, т.е. словарный запас, необходимый для написания программы, с учетом того, что необходимая функция уже реализована в данном языке и, следовательно, программа сводится к вызову этой функции. Используя n\*, Холстед вводит оценку V\*:

V\* = n\* \* log2 n\*, (3)

с помощью которой описывается потенциальный объем программы, соответствующий максимально компактному тексту программы, реализующей данный алгоритм.

Вторая наиболее представительная группа оценок сложности программ - метрики *сложности потока управления* программ. Как правило, с помощью этих оценок оперируют либо плотностью управляющих переходов внутри программ, либо взаимосвязями этих переходов. И в том и в другом случае стало традиционным представление программ в виде управляющего ориентированного графа G = (V,E), где V - вершины, соответствующие операторам, а E - дуги, соответствующие переходам.

Впервые графическое представление программ было предложено Маккейбом. Основной метрикой сложности он предлагает считать цикломатическую сложность графа программы, или, как ее еще называют, цикломатическое число Маккейба, характеризующее трудоемкость тестирования программы.

Для вычисления цикломатического числа Маккейба Z(G) применяется формула

Z(G) = e – v + 2p,

где e - число дуг ориентированного графа G;

v - число вершин;

p - число компонентов связности графа.

Число компонентов связности графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильно связный. Cильно связным называется граф, любые две вершины которого взаимно достижимы. Для графов корректных программ, т. е. графов, не имеющих недостижимых от точки входа участков и "висячих" точек входа и выхода, сильно связный граф, как правило, получается путем замыкания дугой вершины, обозначающей конец программы, на вершину, обозначающую точку входа в эту программу.

Одной из наиболее простых, но, как показывает практика, достаточно эффективных оценок сложности программ является метрика Т. Джилба, в которой логическая сложность программы определяется как насыщенность программы выражениями типа IF-THEN-ELSE. При этом вводятся две характеристики: CL - абсолютная сложность программы, характеризующаяся количеством операторов условия; cl - относительная сложность программы, характеризующаяся насыщенностью программы операторами условия, т. е. cl определяется как отношение CL к общему числу операторов.

Другая группа метрик сложности программ – метрики *сложности потока данных*, т.е. использования, конфигурации и размещения данных в программах. Рассмотрим метрику, связывающую сложность программ с обращениями к глобальным переменным.

Пара "модуль, глобальная переменная" обозначается как (p,r), где p - модуль, имеющий доступ к глобальной переменной r. В зависимости от наличия в программе реального обращения к переменной r формируются два типа пар "модуль, глобальная переменная": фактические и возможные. Возможное обращение к r с помощью p показывает, что область существования r включает в себя p.

Характеристика Aup говорит о том, сколько раз модули up действительно получали доступ к глобальным переменным, а число Pup - сколько раз они могли бы получить доступ. Отношение числа фактических обращений к возможным определяется как Rup = Aup/Pup. Эта формула показывает приближенную вероятность ссылки произвольного модуля на произвольную глобальную переменную. Очевидно, чем выше эта вероятность, тем выше вероятность "несанкционированного" изменения какой-либо переменной, что может существенно осложнить работы, связанные с модификацией программы.

Кроме метрик сложности программ, используются метрики *стилистики и понятности программ*. Наиболее простой метрикой стилистики и понятности программ является оценка уровня комментированности программы F:

F = Nком / Nстр, (4)

где Nком - количество комментариев в программе; Nстр - количество строк или операторов исходного текста.

Таким образом, метрика F отражает насыщенность программы комментариями. Исходя из практического опыта, принято считать, что F >= 0.1, т. е. на каждые десять строк программы должен приходиться минимум один комментарий. Как показывают исследования, комментарии распределяются по тексту программы неравномерно: в начале программы их избыток, а в середине или в конце - недостаток. Это объясняется тем, что в начале программы, как правило, расположены операторы описания идентификаторов, требующие более "плотного" комментирования. Кроме того, в начале программы также расположены "шапки", содержащие общие сведения об исполнителе, характере, функциональном назначении программы и т. п. Такая насыщенность компенсирует недостаток комментариев в теле программы, и поэтому формула (4) недостаточно точно отражает комментированность функциональной части текста программы.

К другим характеристикам программ следует отнести:

1. состав функций обработки информации, определенный функциональными требованиями к программной системе;
2. объем файлов, используемых программной системой;
3. требования к операционной системе и компьютерной технике и др.
   1. **Варианты использования и распространения программных продуктов**

Все программы по характеру использования и категориям пользователей можно разделить на два класса – *утилитарные* программы и *программные продукты*. Первые предназначены для удовлетворения нужд их разработчиков (программы для себя) и не предназначены для широкого распространения. Вторые (программные продукты) используются для удовлетворения потребностей пользователей, широкого распространения и продажи.

Существует большое количество различных компаний, занимающиеся разработкой *проприетарного* программного обеспечения (*proprietary software*). Этим термином обозначают программное обеспечение, которое имеет собственника, осуществляющего контроль над этим программным обеспечением и определяющего собственные лицензионные соглашения по использованию программного продукта. Наиболее типичными ограничениями проприетарного ПО являются:

1) ограничение на коммерческое использование. Существует большое количество проприетарных программных продуктов, которое можно использовать [бесплатно](http://ru.wikipedia.org/wiki/Freeware) в некоммерческих целях для частных лиц, медицинских и учебных заведений, для некоммерческих организаций и т.д. Такое программное обеспечение очень популярно и широко используется, а за счёт своей бесплатности имеет хорошую [техническую поддержку](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%85%D0%BD%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D0%BE%D0%B4%D0%B4%D0%B5%D1%80%D0%B6%D0%BA%D0%B0) со стороны специалистов, у которых отсутствует необходимость дополнительных затрат на обучение;

2) ограничение на распространение. Этот вид ограничений сопровождает обычно крупные программные проекты, когда правообладатель требует оплаты за каждую копию программы. Обычно с таким ограничением используются программные продукты, ориентированные на узкий «профессиональный» сегмент рынка или у программного обеспечения, требующегося большому числу пользователей. Примером может служить пакет программ [Adobe CS3](http://ru.wikipedia.org/wiki/Adobe_Creative_Suite) или операционная система [Microsoft Windows XP](http://ru.wikipedia.org/wiki/Microsoft_Windows_XP);

3) ограничение на модификацию. Этот вид ограничения используется только в программных пакетах с закрытыми исходными кодами и может запрещать или ограничивать любую модификацию программного кода, дизассемблирование и декомпиляцию.

В настоящее время существуют и другие варианты легального распространения программных продуктов, которые появились с использованием Интернета:

1) freeware – бесплатные, свободно распространяемые программы. Очень много серьезных компаний писали, пишут и будут писать freeware-программы. Дело в том, что freeware – прекрасный инструмент в продвижении новых технологий и продуктов. Например, всем известна программа общения ICQ. Это популярный бесплатный продукт, который имеет очень сильные позиции в сравнении с платными программами. Некоторые shareware-программы становятся бесплатными. Всем известный мультимедиапроигрыватель Winamp первоначально был shareware-программой стоимостью в $10. Однако после того, как сайт winamp.com стал привлекать большое количество посетителей, разработчики, получая солидные доходы от рекламы, решили сделать свой продукт бесплатным, еще больше увеличив его популярность и свои кошельки;

2) shareware – условно-бесплатные программы. Употребляется и еще одно наименование этого типа ПО ”пробное” (trial). Основное достоинство shareware “попробуй, прежде чем купить” (try before you buy). Пользователю предоставляется продукт с некоторыми ограничениями, пока он его не приобретет. Ограничения могут быть функциональными и/или временными (чаще всего программа работает 30 дней или определенное количество запусков). Если пользовать решает, что это программа ему нужна, он должен зарегистрироваться, заплатив автору определенную сумму. В противном же случае обязан прекратить использование программы и удалить ее со своего компьютера;

3) public domain software - очень похожие на freeware программы. Распространяются бесплатно. Однако, в отличие от freeware, где автор программы имеет все права на программу, в случае с public domain у него эти права отсутствуют. Программа распространяется вместе с исходным кодом, и автор отказывается от своих прав. Главной идеей было развитие программы в дальнейшем. Однако в силу того, что программа была “ничья”, кто угодно мог слегка модифицировать код, откомпилировать и распространять ее как платную. По этой причине таких программ в настоящее время просто не найти;

4) open source. Представляет собой    развитие концепции public domain software, в которой учтены ошибки этого варианта. Программа, как и раньше, распространяется на бесплатной основе вместе с исходным кодом. Однако автор уже не отказывается от своих прав. Существует система требований к лицензии на программный продукт, который называется The Open Source Definition (OSD), которая представлена на сайте. К программе обязательно должен быть приложен исходный код. Модифицированное ПО должно распространяться на тех же условиях, что и исходный продукт. Автор исходного продукта даже имеет право требовать, чтобы исходный код его программы распространялся без изменений, но в комплекте с соответствующими модифицирующими исправлениями.

Существуют и другие, можно сказать экзотические варианты распространения программ. К ним можно отнести:

1) adware.  К этой категории относятся программы, которые во время своей работы демонстрируют пользователю рекламу, обычно графические баннеры размером 468x60 точек. Adware сочетает в себе freeware и shareware. С одной стороны, пользователь не обязан оплачивать программу, и может ею пользоваться сколь угодно долго, с другой, у него есть стимул оплатить программу, ведь в этом случае он избавится от баннера, который так долго его нервирует. Наибольшее развитие этот тип получил в программах, которые работают в Интернете, ведь именно оттуда скачивается новая реклама. Например, браузеры, download-менеджеры и программы дозвона;

2) donationware.  Такое ПО также распространяются бесплатно, однако разработчик программы в лицензионном соглашении указывает, что, если пользователю программа нравится, то он может (а не обязан) выслать денежное вознаграждение. К сожалению, как показывает практика, пользователи очень вяло реагируют на такие просьбы и очень редко высылают деньги.

Ряд производителей использует OEM-программы (Original Equipment Manu-facturer), т.е. встроенные программы, устанавливаемые на компьютеры или поставляемые вместе с компьютером.

Программный продукт должен быть соответствующим образом подготовлен к эксплуатации (отлажен), иметь необходимую техническую документацию, предоставлять сервис и гарантию надежной работы программы, иметь товарный знак изготовителя, а также наличие кода государственной регистрации.

**Глава 2. Классификация программного обеспечения**

2.1. Классы программного обеспечения

*Под программным обеспечением* понимается совокупность программ, выполняемых вычислительной системой. К программному обеспечению относится также вся область деятельности по проектированию и разработке ПО: технология проектирования программ; методы тестирования программ; методы доказательства правильности программ; анализ качества работы программ; документирование программ; разработка и использование программных средств, облегчающих процесс проектирования программного обеспечения, и многое другое.

Программное обеспечение – неотъемлемая часть компьютерной системы. Оно является логическим продолжением технических средств. Сфера применения конкретного компьютера определяется созданным для него программным обеспечением. Сам по себе компьютер не обладает знаниями ни в одной области применения. Все эти знания сосредоточены в выполняемых на компьютерах программах. Программное обеспечение современных компьютеров включает миллионы программ – от игровых до научных.

Программные продукты можно классифицировать по различным признакам. Рассмотрим классификацию, в которой основополагающим признаком является сфера (область) исполь­зования программных продуктов:

* аппаратная часть автономных компьютеров и сетей ЭВМ;
* функциональные задачи различных предметных областей;
* технология разработки программ.



Рис. 1. Классы программных продуктов

Для поддержки информационной технологии в этих областях выделим соответственно три класса программных продуктов, представленных на рис. 1:

* системное программное обеспечение;
* пакеты прикладных программ;
* инструментарий технологии программирования.

Системное программное обеспечение (System Software) — совокупность программ и программных комплексов, предназначенная для обеспечения работы компьютера и сетей ЭВМ.

*Системное программное обеспечение* выполняет следующие задачи:

* создание операционной среды функционирования других программ;
* обеспечение надежной и эффективной работы самого компьютера и вычислитель­ной сети;
* проведение диагностики, локализации сбоев, ошибок и отказов и профилактики аппаратуры компьютера и вычислитель­ных сетей;
* выполнение вспомогательных технологических процессов (копирование, архивиро­вание, восстановление файлов программ и баз данных и т.д.).

Данный класс программных продуктов тесно связан с типом компьютера и является его неотъемлемой частью. Программные продукты в основном ориентированы на квалифи­цированных пользователей – профессионалов в компьютерной области: системного про­граммиста, администратора сети, прикладного программиста, оператора. Однако знание базовой технологии работы с этим классом программных продуктов требуется и конечным пользователям персонального компьютера, которые самостоятельно не только работают со своими программами, но и выполняют обслуживание компьютера, программ и данных.

Программные продукты данного класса носят общий характер применения, независи­мо от специфики предметной области. К ним предъявляются высокие требования по надеж­ности и технологичности работы, удобству и эффективности использования.

*Пакет прикладных программ* (application program package) представляет собой комплекс взаимосвязанных программ, предназначенный для решения задач определенного класса кон­кретной предметной области. Пакеты прикладных программ (ППП) служат программным инструмента­рием решения функциональных задач и являются самым многочисленным классом про­граммных продуктов. В данный класс входят программные продукты, выполняющие обработку информации различных предметных областей.

Установка пакетов прикладных программ на компьютер выполняется системными администраторами, системными программистами, а также (в некоторых случаях) квалифицированными пользователями. Непосредственную эксплуатацию программных продуктов осуществляют, как правило, конеч­ные пользователи – потребители информации, во многих случаях, деятельность которых весьма далека от компьютерной области. Данный класс программных продуктов может быть весьма специфичным для отдельных предметных областей.

*Инструментарий технологии программирования* представляет собой совокупность про­грамм и программных комплексов, обеспечивающих технологию разработ­ки, отладки и внедрения создаваемых программных продуктов.

Инструментарий технологии программирования включает специализированные программные продукты, ко­торые являются инструментальными средствами разработчика. Программные продукты данного класса поддерживают все технологические этапы процесса проектирования, программирования (кодирования), отладки и тестирования создаваемых программ. Пользовате­лями технологии программирования являются системные и прикладные программисты.

**2.2. Структура системного программного обеспечения**

Системное программное обеспечение (рис.2) можно разделить на *базовое программное обеспечение,* которое, как правило, поставляется вместе с компью­тером, и *сервисное программное* обеспечение, которое может быть приоб­ретено дополнительно.

Базовое программное обеспечение (base software) – минимальный набор программных средств, обеспечивающих работу компьютера. Сервисное программное обеспечение включает программы и программные ком­плексы, которые расширяют возможности базового программного обеспе­чения и организуют более удобную среду работы пользователя.



Рис. 2

В базовое программное обеспечение входят:

* операционная система;
* операционные оболочки (обычно текстовые и графические);
* сетевая операционная система.

*Операционная система* предназначена для управления выполнением пользо­вательских программ, планирования и управления вычислительными ресурсами ЭВМ.

Наиболее традиционное сравнение ОС осуществляется по следующим характеристи­кам процесса обработки информации:

* управление памятью (максимальный объем адресуемого пространства, типы памяти, технические показатели использования памяти);
* функциональные возможности вспомогательных программ (утилит) в составе опера­ционной системы;
* наличие компрессии диска;
* возможность архивирования файлов;
* поддержка многозадачного режима работы;
* поддержка сетевого программного обеспечения;
* наличие качественной документации;
* условия и сложность процесса инсталляции;
* мобильность (переносимость), безопасность, надежность и др.

Операционные системы, учитывая их центральное положение в программном обеспечении компьютеров, подробно рассматриваются в следующей главе учебника.

*Сетевые операционные системы —* комплекс программ, обеспечиваю­щий обработку, передачу и хранение данных в сети. Сетевая ОС предоставляет пользовате­лям различные виды сетевых служб (управление файлами, электронная почта, аудио- и видеоконференции, распределенные вычисления, процессы управления сетью и др.), поддерживает работу в абонентских системах. Сетевые опе­рационные системы используют архитектуру *клиент-сервер* или одноранговую архитектуру. Вначале сетевые операционные системы поддерживали лишь локальные вы­числительные сети (ЛВС), сейчас эти операционные системы распространяются на ассоциа­ции локальных сетей (см. раздел 4).

*Операционные оболочки —* специальные программы, предназначенные для облегчения общения пользователя с командами операционной системы. Операционные обо­лочки имеют текстовый и графический варианты интерфейса конечного пользователя, а в будущем возможны варианты речевого интерфейса и распознавание рукописного ввода данных. Эти программы существенно упрощают задание управляющей информации для вы­полнения команд операционной системы, уменьшают напряженность и сложность работы конечного пользователя.

Расширением базового программного обеспечения компьютера является набор *сервис­ных,* дополнительно устанавливаемых программ (или программ, поставляемых непосредственно с операционными системами), которые можно классифицировать по функциональному признаку следующим образом:

* программы диагностики работоспособности компьютера;
* антивирусные программы, обеспечивающие защиту компьютера, обнаружение и вос­становление зараженных файлов;
* программы обслуживания дисков, обеспечивающие проверку качества поверхности магнитного диска, контроль сохранности файловой системы на логическом и физичес­ком уровнях, сжатие дисков, создание страховых копий дисков, резервирование дан­ных на внешних носителях и др.;
* программы архивирования данных, которые обеспечивают процесс сжатия информа­ции в файлах с целью уменьшения объема памяти для ее хранения;
* программы обслуживания сети.

Эти программы часто называются утилитами. Утилиты – программы, служащие для выполнения вспомогательных опе­раций обработки данных или обслуживания компьютеров (диагностики, тестирования аппаратных и программных средств, оптимизации использо­вания дискового пространства, восстановления разрушенной на магнитном диске информации и т.п.).

В современных операционных системах такие утилиты могут быть представлены, как например, в Windows группами программ Стандартные и Служебные. В них входи ряд полезных программ: калькулятор, звукозапись, блокнот и др. В группе Служебные имеется ряд программ, расширяющих возможности операционной системы: очистка и дефрагментация диска, восстановление системы и т.п.

**2.3. Прикладное программное обеспечение**

Прикладные программы предназначены для того, чтобы обеспечить применение вычислительной техники в различных сферах деятельности человека. Помимо создания новых программных продуктов разработчики прикладных программ большие усилия тратят на совершенствование и модернизацию популярных систем, создание их новых версий. Новые версии, как правило, поддерживают старые, сохраняя преемственность, и включают в себя базовый минимум (стандарт) возможностей.

Один из возможных вариантов классификации программных средств (ПС), составляющих прикладное программное обеспечение (ППО), отражен на рис. 3. Как и почти всякая классификация, приведенная на рисунке, не является единственно возможной. В ней представлены даже не все виды прикладных программ. Тем не менее, использование классификации полезно для создания общего представления о ППО.

Рис.3

Несмотря на широкие возможности использования компьютеров для обработки самой разной информации, самыми популярными являются программы, предназначенные для работы с текстами - текстовые редакторы и издательские системы. Текстовыми редакторами называют программы для ввода, обработки, хранения и печатания текстовой информации в удобном для пользователя виде. Эксперты оценивают использование компьютера в качестве печатающей машинки в 80%.

Большую популярность приобрели программы обработки графической информации. Компьютерная графика в настоящее время является одной из самых динамично развивающихся областей программного обеспечения. Она включает в себя ввод, обработку и вывод графической информации - чертежей, рисунков, фотографий, картин, текстов и т.д. - средствами компьютерной техники. Различные типы графических систем позволяют быстро строить изображения, вводить иллюстрации с помощью сканера или видеокамеры, создавать анимационные ролики. Графические редакторы позволяют пользоваться различным инструментарием художника, стандартными библиотеками изображений, наборами стандартных шрифтов, редактированием изображений, копированием и перемещением фрагментов по страницам экрана и др.

Для выполнения расчетов и дальнейшей обработки числовой информации существуют специальные программы – электронные таблицы. В процессе деятельности любого специалиста часто требуется представить результаты работы в виде таблиц, где одна часть полей занята исходными данными, а другая – результатами вычислений и графического анализа. Характерными для них является большой объем перерабатываемой информации, необходимость многократных расчетов при изменении исходных данных. Автоматизацией подобной рутинной работы и занимаются электронные таблицы.

Одним из наиболее перспективных направлений развития вычислительной техники является создание специальных аппаратных средств для хранения гигантских массивов информационных данных, и последующей нечисловой обработки их, чаще всего – поиска и сортировки. Для компьютерной обработки подобных баз данных используют системы управления базами данных (СУБД). Последние представляют собой набор средств программного обеспечения, необходимых для создания, обработки и вывода записей баз данных. Различают несколько типов СУБД: *иерархические, сетевые, реляционные.* При работе с СУБД выделяют несколько последовательных этапов:

• проектирование базы данных;

• создание структуры базыданных;

• заполнение базы данных;

• просмотр и редактирование базыданных**;**

• сортировку базы данных;

• поиск необходимой записи;

*•* выборку информации по определенным признакам (критериям);

• создание отчетов.

Как правило, большинство популярных систем управления базами данных поддерживают эти этапы и предоставляют удобный инструментарий для их реализации.

Желание объединить функции различных прикладных программ в единую систему привело к созданию интегрированных систем. Современная концепция интеграции программных средств – кооперация отдельных прикладных программных систем по типу широко известного пакета Microsoft Office. Универсальные интегрированные системы разрабатывались по принципу единой системы, содержащей в качестве элементов множество программ, полезных практически любому пользователю. К таким программам относятся: текстовые и графические редакторы, электронные таблицы, пакеты для разработки презентаций, почтовые программы, органайзеры, системы управления базами данных и др.

Сами системы, входящие в пакет, являются независимыми, более того, они сами представляют локально интегрированный пакет, поскольку помимо основной своей задачи поддерживают функции других систем. Например, текстовый редактор Word обладает возможностью манипулировать с электронными таблицами и базами данных, а в электронной таблице Excel встроен мощный текстовый редактор. Для сопряжения информационных данных из различных программных систем в них предусматривают импортно-экспортную систему обмена с перекодировкой форматов представления данных.

К прикладному программному обеспечению относятся также инструментальные программные средства специального назначения.В настоящее время создаются различные специальные программные системы целевого назначения, предназначенные для работы специалистов в некоторой предметной области. Такие программы называют авторскими инструментальными системами.Авторская система представляет интегрированную среду с заданной интерфейсной оболочкой, которую пользователь может наполнить информационным содержанием своей предметной области.

Среди таких систем получили распространение экспертные системы. Такие программы ведут себя подобно эксперту в некоторой узкой прикладной области. Экспертные системы призваны решать задачи с неопределенностью и неполными исходными данными, требующие для своего решения экспертных знаний. Кроме того, эти системы должны уметь объяснять свое поведение и свое решение.

Принципиальным отличием экспертных систем от других программ являетсяихадаптивность, т.е. изменчивость в процессе самообучения.

Принято выделять в экспертных системах три основных модуля:

• модуль базы знаний;

• модуль логического вывода;

• интерфейс пользователя.

Экспертные системы, являющиеся основой искусственного интеллекта, получили широкое распространение в различных областях науки (например, для классификации животных и растений по видам, химического анализа), в медицине (постановка диагноза, анализ электрокардиограмм, определение методов лечения), в технике (поиск неисправностей в технических устройствах, слежение за полетом космических кораблей и спутников), в политологии и социологии, криминалистике, лингвистике и т.д.

В последнее время широкую популярность получили программы обработки гипертекстовой информации. Гипертекст – это форма организации текстового материала не в линейной последовательности, а в форме указании возможных переходов (ссылок), связей между отдельными его фрагментами. В обычном тексте используется обычный линейный принцип размещения информации, и доступ к нему (тексту) осуществляется последовательно. В гипертекстовых системах информация напоминает текст энциклопедии, и доступ к любому выделенному фрагменту текста осуществляется произвольно по ссылке. Организация информации в гипертекстовой форме используется при создании справочных пособий, словарей, контекстной помощи (Help) в прикладных программах.

Расширение концепции гипертекста на графическую и звуковую информацию приводит к понятию гипермедиа. Идеигипермедиа получили распространение в сетевых технологиях, в частности в Интернет-технологиях. Технология WWW (World Wide Web) позволила структурировать громадные мировые информационные ресурсы посредством гипертекстовых ссылок. Разработаны программные средства, позволяющие создавать подобные Web-странички. Стали высокоразвитыми механизмы поиска нужной информации в лабиринте информационных потоков. Популярными поисковыми средствами в Интернет являются Yandex, Gogle, Yahoo, AltaVista, Magellan, Rambler и др.

Мультимедиа (multimedia) – это взаимодействие визуальных и аудиоэффектов под управлением интерактивного программного обеспечения. Появление и широкое распространение компакт-дисков (CD-ROM и DVD) сделало эффективным использование мультимедиа в рекламной и информационной службе, сетевых телекоммуникационных технологиях, в обучении.

Мультимедийные игровые и обучающие системы начинают вытеснять традиционные “бумажные библиотеки”. Сегодня в DVD-библиотеках можно “гулять” по музеям, Московскому Кремлю и т.д. с помощью “электронного путеводителя”, изучать различные учебные дисциплины, языки программирования и др.

Отдельную группу прикладного ПО составляют программные средства профессионального уровня. Каждая прикладная программа этой группы ориентируются на достаточно узкую предметную область, но проникает в нее максимально глубоко. Так функционируют АСНИ – автоматизированные системы научных исследований, каждая из которых “привязана” к определенной области науки, САПР – системы автоматизированного проектирования, каждая из которых также работает в узкой области, АСУ - автоматизированные системы управления (которых в 60 - 70 годах были разработаны тысячи).

Следует отметить не только условность предложенной выше классификации, но и наличие пересечений. Так, каждую конкретную экспертную систему вполне можно отнести к ППО профессионального уровня; принцип гипертекста реализован в ряде авторских систем и т.д. В тоже время в связи с важностью отдельных групп прикладного ПО для широкого круга пользователей компьютерных систем прикладные программные системы общего назначения рассматриваются более подробно в разделе 8.

**Глава 3. Операционные системы**

**3.1. Понятие операционной системы**

Современный компьютер – сложнейшая аппаратно-программная система. Написание программ для компьютера, их отладка и последующее выполнение представляет собой сложную трудоемкую задачу. Во многом это связано с тем, что существует огромная разница между тем, что удобно для людей, и тем, что удобно для компьютеров. Компьютер понимает только свой машинный язык (назовем его Я0), для человека наиболее удобен – разговорный или хотя бы язык описания алгоритмов – алгоритмический язык. Проблему можно решить двумя способами. Оба способа связаны с разработкой команд, которые были бы боле удобны для человека, чем встроенные машинные команды компьютера. Эти новые команды в совокупности формируют некоторый язык, который назовем Я1.

Упомянутые два способа решения проблемы различаются тем, каким образом компьютер будет выполнять программы, написанные на языке Я1. Первый способ – замена каждой команды языка Я1 на эквивалентный набор команд в языке Я0. В этом случае компьютер выполняет новую программу, написанную на языке Я0, вместо программы, написанной на языке Я1. Эта технология называется *трансляцией*.

Второй способ - написание программы на языке Я0, которая берет программы, написанные на языке Я1, в качестве входных данных, рассматривает каждую команду по очереди и сразу выполняет эквивалентный набор команд языка Я0. Эта технология не требует составления новой программы на Я0. Она называется *интерпретацией*, а программа, которая осуществляет интерпретацию, называется *интерпретатором*.

В подобной ситуации проще представить себе существование гипотетического компьютера или *виртуальной* машины, для которой машинным языком является язык Я1, чем думать о трансляции и интерпретации. Назовем такую виртуальную машину М1, а виртуальную машину с языком Я0 – М0. Для виртуальных машин можно будет писать программы, как будто они действительно существуют.

Очевидно, можно пойти дальше – создать еще набор команд, который в большей степени ориентирована человека и в меньшей степени на компьютер, чем Я1. Этот набор формирует язык Я2 и соответственно виртуальную машину М2. Так можно продолжать до тез пор, пока не дойдем до подходящего нам языка уровня n.

Большинство современных компьютеров состоит из двух и более уровней. Уровень 0 – аппаратное обеспечение машины. Электронные схемы этого уровня выполняют программы, написанные на языке уровня 1. Следующий уровень – *микроархитектурный* уровень.

На этом уровне можно видеть совокупности 8 или 32 (иногда и больше) регистров, которые формируют локальную память и АЛУ (арифметико-логическое устройство). Регистры вместе с АЛУ формируют тракт данных, по которому поступают данные. Основная операция этого тракта заключается в следующем. Выбирается один или два регистра, АЛУ производит над ними какую-то операцию, а результат помещается в один из этих регистров. На некоторых машинах работа тракта контролируется особой программой, которая называется микропрограммой. В других машинах такой контроль выполняется аппаратным обеспечением.

Следующий (второй) уровень составляет уровень *архитектуры системы команд*. Команды используют регистры и другие возможности аппаратуры. Команды формируют уровень ISA (Instruction Set Architecture), называемый машинным языком. Обычно машинный язык содержит от 50 до 300 команд, служащих преимущественно для перемещения данных по компьютеру, выполнения арифметических операций и сравнения величин.

Следующий (третий) уровень обычно гибридный. Большинство команд в его языке есть также и на уровне архитектуры системы команд. У этого уровня есть некоторые дополнительные особенности: набор новых команд, другая организация памяти, способность выполнять две и более программы одновременно и некоторые другие. С течением времени набор таких команд существенно расширился. В нем появились так называемые макросы операционной системы или вызовы супервизора, называемые теперь системными вызовами.

Новые средства, появившиеся на третьем уровне, выполняются интерпретатором, который работает на втором уровне. Этот интерпретатор был когда-то назван *операционной системой.* Команды третьего уровня, идентичные командам второго уровня, выполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Иными словами, одна часть команд третьего уровня интерпретируется операционной системой, а другая часть – микропрограммой. Вот почему этот уровень операционной системы считается гибридным.

Операционная система была создана для того, чтобы автоматизировать работу оператора и скрыть от пользователя сложности общения с аппаратурой, предоставив ему более удобную систему команд. Нижние три уровня (с нулевого по второй) конструируются не для того, чтобы с ними работал обычный программист. Они изначально предназначены для работы интерпретаторов и трансляторов, поддерживающих более высокие уровни. Эти трансляторы и интерпретаторы составляются системными программистами, которые специализируются на разработке и построении новых виртуальных машин.

Над операционной системой (ОС) расположены остальные системные программы. Здесь находятся интерпретатор команд (оболочка), компиляторы, редакторы и т.д. Подобные программы не являются частью ОС (иногда оболочку пользователи считают операционной системой). Под операционной системой обычно понимается то программное обеспечение, которое запускается в *режиме ядра* или, как еще его называют, *режиме супервизора*. Она защищена от вмешательства пользователя с помощью специальных аппаратных средств.

Четвертый уровень представляет собой символическую форму одного из языков низкого уровня (обычно ассемблер). На этом уровне можно писать программы в приемлемой для человека форме. Эти программы сначала транслируются на язык уровня 1, 2 или3, а затем интерпретируются соответствующей виртуальной или фактически существующей (физической) машиной.

Уровни с пятого и выше предназначены для прикладных программистов, решающих конкретные задачи на языках высокого уровня (C, C++, C#, VBA и др.) Компиляторы и редакторы этих уровней *запускаются в пользовательском режиме*. Еще на более высоких уровнях располагаются прикладные программы пользователей.

Большинство пользователей компьютеров имеют опыт общения с операционной системой, по крайней мере, в той степени, чтобы эффективно выполнять свои текущие задачи. Однако они испытывают затруднения при попытке дать определение операционной системе. В известной степени проблема связана с тем, что операционные системы выполняют две основные, но практически не связанные между собой функции: расширение возможностей компьютера и управление его ресурсами.

С точки зрения пользователя ОС выполняет функцию расширенной машины или виртуальной машины, в которой легче программировать и легче работать, чем непосредственно с аппаратным обеспечением, составляющим реальный компьютер. Операционная система не только устраняет необходимость работы непосредственно с дисками и предоставляет простой, ориентированный на работу с файлами интерфейс, но и скрывает множество неприятной работы с прерываниями, счетчиками времени, организацией памяти и другими компонентами низкого уровня.

Однако, концепция, рассматривающая операционную систему, прежде всего как удобный интерфейс пользователя, - это взгляд сверху вниз. Альтернативный взгляд, снизу вверх, дает представление об операционной системе как о механизме, присутствующем в компьютере для управления всеми компонентами этой сложнейшей системы. В соответствии с этим подходом работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти, дисков, принтеров, устройств ввода-вывода, датчиков времени и т.п. между различными программами, конкурирующими за право их использовать.

**3.2. Операционная среда и операционная оболочка**

Операционные системы (ОС) в современном их понимании (их назначении и сущности) появились значительно позже первых компьютеров (правда и, по всей видимости, исчезнут в этой сущности в компьютерах будущего). Почему и когда появились ОС? Считается[[1]](#footnote-1), что первая цифровая вычислительная машина ENIAC (Electronic Numerical Integrator and Computer) была создана в 1946 году по проекту «Проект РХ» Министерства обороны США. На реализацию проекта затрачено 500 тыс. долларов. Компьютер содержал 18000 электронных ламп, массу всякой электроники, имел 12 десятиразрядных сумматоров, а для ускорения некоторых арифметических операций имел умножитель и “делитель-извлекатель” квадратного корня. Программирование сводилось к связыванию различных блоков проводами. Конечно, никакого программного обеспечения, и, тем более, операционных систем, тогда еще не существовало [4, 37].

Интенсивное создание различных моделей ЭВМ относится к началу 50-х годов прошлого века. В эти годы одни и те же группы людей участвовали и в проектировании, и в создании, и в программировании, и в эксплуатации ЭВМ. Программирование осуществлялось исключительно на машинном языке (а затем на Ассемблере), не было никакого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм. Операционные системы еще не появились, а все задачи организации вычислительного процесса решались вручную каждым программистом с примитивного пульта управления ЭВМ.

С появлением полупроводниковых элементов вычислительные возможности компьютеров существенно выросли. Наряду с этим заметно прогрессировали достижения в области автоматизации программирования и организации вычислительных работ. Появились алгоритмические языки (Алгол, Фортран, Кобол) и системное программное обеспечение (трансляторы, редакторы связи, загрузчики и др.). Выполнение программ усложнилось и включало в себя следующие основные действия:

* загрузка нужного транслятора (установка нужных МЛ и др.);
* запуск транслятора и получение программы в машинных кодах;
* связывание программы с библиотечными подпрограммами;
* загрузка программы в оперативную память;
* запуск программы;
* вывод результатов работы программы на печатающее или другое периферийное устройство.

Для организации эффективного использования всех средств компьютера в штаты вычислительных центров ввели должности специально обученных операторов, профессионально выполнявших работу по организации вычислительного процесса для всех пользователей этого центра. Однако, как бы ни был подготовлен оператор, ему тяжело состязаться в производительности с работой устройств компьютера. И поэтому большую часть времени дорогостоящий процессор простаивал, а, следовательно, использование компьютеров не было эффективным.

С целью исключения простоев были предприняты попытки разработки специальных программ – мониторов – прообразов первых операционных систем, которые осуществляли автоматический переход от задания к заданию. Считается, что первую операционную систему создала в 1952 году для своих компьютеров IBM – 701 исследовательская лаборатория фирмы General Motors [ 8 ]. В 1955 году эта фирма и North American Aviation совместно разработали ОС для компьютера IBM – 704.

В конце 50-х годов прошлого века ведущие фирмы изготовители поставляли операционные системы со следующими характеристиками:

* пакетная обработка одного потока задач;
* наличие стандартных программ ввода-вывода;
* возможности автоматического перехода от программы к программе;
* средства восстановления после ошибок, обеспечивающие автоматическую «очистку» компьютера в случаи аварийного завершения очередной задачи и позволяющие запускать следующую задачу при минимальном вмешательстве оператора;
* языки управления заданиями, предоставляющие пользователям возможность описывать свои задания и ресурсы, требуемые для их выполнения.

Пакет представляет собой набор (колоду) перфокарт, организованную специальным образом (задание, программы, данные). Для ускорения работы он мог переноситься на магнитную ленту или диск. Это позволяло сократить простой дорогой аппаратуры. Надо сказать, что в настоящее время в связи с прогрессом микроэлектронных технологий и методологий программирования значительно снизилась стоимость аппаратных и программных средств компьютерной техники. Поэтому сейчас основное внимание уделяется тому, чтобы сделать работу пользователей и программистов более эффективной, поскольку затраты труда квалифицированных специалистов сейчас представляют собой гораздо большую долю общей стоимости вычислительных систем, чем аппаратные и программные средства компьютеров.

Расположение операционной системы в иерархической структуре программного и аппаратного обеспечения компьютера можно представить, как показано на рис. 3.1.



Рис. 3.1

Самый нижний уровень содержит различные устройства компьютера, состоящие из микросхем, проводников, источников питания, электронно-лучевых трубок и т.п. Этот уровень можно разделить на подуровни, например контроллеры устройств, а затем сами устройства. Возможно деление и на большее число уровней. Выше расположен микроархитектурный уровень, на котором физические устройства рассматриваются как отдельные функциональные единицы.

На микроархитектурном уровне находятся внутренние регистры центрального процессора (их может быть несколько), и арифметико-логические устройства со средствами управления ими. На этом уровне реализуется выполнение машинных команд. В процессе выполнения команд используются регистры процессора и устройств, а также другие возможности аппаратуры. Команды, видимые для работающего на ассемблере программиста, формируют уровень ISA (Instruction Set Architecture – архитектура системы команд), часто называемый машинном языком.

Операционная система предназначена, для того чтобы скрыть все эти сложности. Конечный пользователь обычно не интересуется деталями устройства аппаратного обеспечения компьютера. Компьютер ему видится как набор приложений. Приложение может быть написано программистом на каком-либо языке программирования. Для упрощения этой работы программист использует набор системных программ, некоторые из которых называются утилитами. С их помощью реализуются часто используемые функции, которые помогают работать с файлами, управлять устройствами ввода-вывода и т.п. Программист использует эти средства при разработке программ, а приложения во время выполнения обращаются к утилитам для выполнения определенных функций. Наиболее важной из системных программ является операционная система, которая освобождает программиста от необходимости глубокого знания устройства компьютера и представляет ему удобный интерфейс для его использования. Операционная система выступает в роли посредника, облегчая программисту, пользователям и программным приложениям доступ к различным службам и возможностям компьютера [4,18].

Таким образом, *операционная система* – это набор программ, контролирующих работу прикладных программ и системных приложений и исполняющих роль интерфейса между пользователями, программистами, прикладными программами, системными приложениями и аппаратным обеспечением компьютера.

Образно можно сказать, что аппаратура компьютера предоставляет «сырую» вычислительную мощность, а задача операционной системы заключается в том, чтобы сделать использование этой вычислительной мощности доступным и по возможности удобным для пользователя. Программист может не знать детали управления конкретными ресурсами (например, диском) компьютера, а должен обращаться к операционной системе с соответствующими вызовами, чтобы получить от нее необходимые сервисы и функции. Этот набор сервисов и функций и представляет собой операционную среду, в которой выполняются прикладные программы.

Таким образом, *операционная среда* – это программная среда, образуемая операционной системой, определяющая интерфейс прикладного программирования (API) как множество системных функций и сервисов (системных вызовов), предоставляемых прикладным программам. Операционная среда может включать несколько интерфейсов прикладного программирования. Кроме основной операционной среды, называемой естественной (native), могут быть организованы путем эмуляции (моделирования) дополнительные программные среды, позволяющие выполнять приложения, которые рассчитаны на другие операционные системы и даже другие компьютеры.

Еще одно важное понятие, связанное с операционной системой, относится к реализации пользовательских интерфейсов. Как правило, любая операционная система обеспечивает удобную работу пользователя за счет средств пользовательского интерфейса. Эти средства могут быть неотъемлемой частью операционной среды, например, графический интерфейс Windows или текстовый интерфейс командной строки MS DOS, а могут быть реализованы отдельной системной программой – оболочкой операционной системы (например, Norton Commander для MS DOS). В общем случае под *оболочкой операционной* *системы* понимается часть операционной среды, определяющая интерфейс пользователя, его реализацию (текстовый, графический и т.п.), командные и сервисные возможности пользователя по управлению прикладными программами и компьютером.

Перейдем к рассмотрению эволюции операционных систем.

**3.3. Эволюция операционных систем**

Рассматривая эволюцию ОС, следует иметь в виду, что разница во времени реализации некоторых принципов организации отдельных операционных систем до их общего признания, а также терминологическая неопределенность не позволяет дать точную хронологию развития ОС. Однако сейчас уже достаточно точно можно определить основные вехи на пути эволюции операционных систем.

Существуют также различные подходы к определению поколений ОС. Известно разделение ОС на поколения в соответствии с поколениями вычислительных машин и систем [28, 36. 37]. Такое деление нельзя считать полностью удовлетворительным, так как развитие методов организации ОС в рамках одного поколения ЭВМ, как показал опыт их создания, лежит в достаточно широком диапазоне. Другая точка зрения не связывает поколение ОС с соответствующими поколениями ЭВМ. Так, например, известно определение поколений ОС по уровням входного языка ЭВМ, режимам использования центральных процессоров, формам эксплуатации систем и т.п. [4, 8, 20, 39].

Видимо, наиболее целесообразным следует считать выделение этапов развития ОС в рамках отдельных поколений ЭВМ и ВС.

Первым этапом развития системного программного обеспечения можно считать использование библиотечных программ, стандартных и служебных подпрограмм и макрокоманд. Концепция библиотек подпрограмм является наиболее ранней и восходит к 1949 году [37]. С появлением библиотек получим развитие автоматические средства их сопровождения - программы-загрузчики и редакторы связей. Эти средства использовались в ЭВМ первого поколения, когда операционных систем, как таковых еще не существовало (рис. 3.2).

Стремление устранить несоответствие между производительностью процессоров и скоростью работы электромеханических устройств ввода-вывода, с одной стороны, и использование достаточно быстродействующих накопителей на магнитных лентах и барабанах (НМЛ и НМБ), а затем на магнитных дисках (НМД), с другой стороны, привело к необходимости решения задач буферизации и блокирования-деблокирования данных. Возникли специальные программы методов доступа, которые вносились в объекты модулей редакторов связей (впоследствии стали использоваться принципы полибуферизации). Для поддержания работоспособности и облегчения процессов эксплуатации машин создавались диагностические программы. Таким образом, было создано базовое системное программное обеспечение.

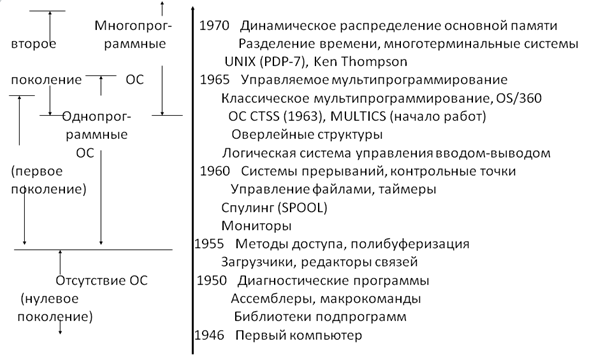


Рис. 3.2

С улучшением характеристик ЭВМ и ростом их производительности стала ясной недостаточность базового программного обеспечения (ПО). Появились операционные системы ранней пакетной обработки – мониторы. В рамках системах пакетной обработки во время выполнения любой работы в пакете (трансляция, сборка, выполнение готовой программы) никакая часть системного ПО не находились в оперативной памяти, так как вся память предоставлялась текущей работе. Затем появились мониторные системы, в которых оперативная память делилась на три области: фиксированная область мониторной системы, область пользователя и область общей памяти (для хранения данных, которыми могут обмениваться объектные модули).

Началось интенсивное развитие методов управления данными, возникала такая важная функция ОС как реализация ввода-вывода без участия центрального процесса – так называемый спулинг (от англ. SPOOL – Simultaneous Peripheral Operation on Line).

Появление новых аппаратных разработок (1959-1963 гг.) – систем прерываний, таймеров, каналов – стимулировало дальнейшее развитие ОС [4, 18, 39]. Возникли исполнительные системы, которые представляли собой набор программ для распределения ресурсов ЭВМ, связей с оператором, управления вычислительным процессом и управления вводом-выводом. Такие исполнительные системы позволили реализовать довольно эффективную по тому времени форму эксплуатации вычислительной системы – однопрограммную пакетную обработку. Эти системы давали пользователю такие средства, как контрольные точки, логические таймеры, возможность построения программ оверлейной структуры, обнаружение нарушений программами ограничений, принятых в системе, управления файлами, сбор учетной информации и др.

Однако однопрограммная пакетная обработка с ростом производительности ЭВМ не могла обеспечить экономически приемлемый уровень эксплуатации машин. Решением стало мультипрограммирование – способ организации вычислительного процесса, при котором в памяти компьютера находиться несколько программ, попеременно выполняющихся одним процессором, причем для начала или продолжения счета по одной программе не требовалось завершения других. В мультипрограммной среде проблемы распределения ресурсов и защиты стали более острыми и трудноразрешимыми.

Теория построения операционных систем в этот период обогатилось рядом плодотворных идей. Появились различные формы мультипрограммных режимов работы, в том числе разделение времени – режим, обеспечивающий работу многотерминальной системы. Была создана и развита концепция виртуальной памяти, а затем и виртуальных машин. Режим разделения времени позволил пользователю интерактивно взаимодействовать со своими программами, как это было до появления систем пакетной обработки.

Одной из первых ОС, использующих эти новейшие решения, была операционная система МСР (главная управляющая программа), созданной фирмой Burroughs для своих компьютеров В5000 в 1963 году. В этой ОС были реализованы многие концепции и идеи, ставшие впоследствии стандартными для многих операционных систем (рис. 3.3):

* мультипрограммирование;
* мультипроцессорная обработка;
* виртуальная память;
* возможность отладки программ на исходном языке;
* написание операционной системы на языке высокого уровня.

Известной системой разделения времени того периода стала система CTSS (Compatible Time Sharing System) – совместимая система разделения времени, разработанная в Массачусетском технологическом институте (1963 год) для компьютера IBM-7094 [37]. Эта система была использована для разработки в этом же институте совместно с Bell Labs и General Electric системы разделения времени следующего поколения MULTICS (Multiplexed Information And Computing Service). Примечательно, что эта ОС была написана в основном на языке высокого уровня EPL (первая версия языка PL/1 фирма IBM).

Одним из важнейших событий в истории операционных систем считается появление в 1964 году семейства компьютеров под названием System/360 фирмы IBM, а позже System/370. Это было первой в мире реализацией концепции семейства программно и информационно совместимых компьютеров, ставших впоследствии стандартной для всех фирм компьютерной отрасли.

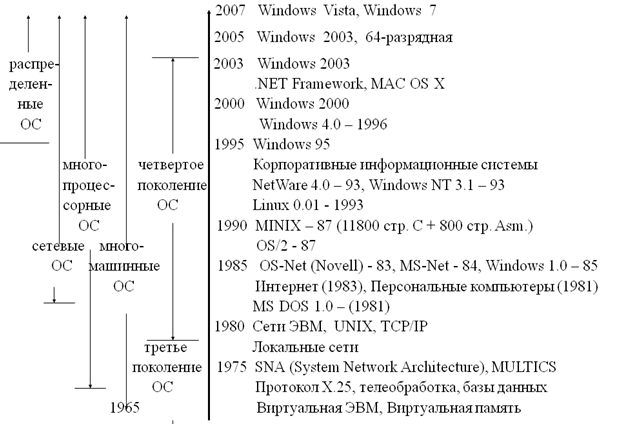


Рис. 3.3

Нужно отметить, что основной формой использования ЭВМ, как в системах разделения времени, так и в системах пакетной обработки, стал многотерминальный режим. При этом не только оператор, но и все пользователи получали возможность формулировать свои задания и управлять их выполнением со своего терминала. Поскольку терминальные комплексы скоро стало возможным размещать на значительных расстояниях от компьютера (благодаря модемным телефонным соединениям), появились системы удаленного ввода заданий и телеобработки данных. В ОС добавились модули, реализующие протоколы связи [26, 28, 36,37].

К этому времени произошло существенное изменение в распределении функций между аппаратными и программными и программными средствами компьютера. Операционная система становится «неотъемлемой частью ЭВМ», как бы продолжением аппаратуры. В процессорах появился привилегированный (Супервизор в OS/360) и пользовательский (Задача в OS/360) режимы работы, мощная система прерываний, защита памяти, специальные регистры для быстрого переключения программ, средства поддержки виртуальной памяти и др.

В начале 70-х годов появились первые сетевые ОС, которые позволили не только рассредоточить пользователей, как в системах телеобработки данных, но и организовать распределенное хранение и обработку данных между компьютерами, соединенных электрическими связями. Известен проект ARPANET MO США. В 1974 году IBM объявила о создании собственной сетевой архитектуры SNA для своих мэйнфреймов, обеспечивающей взаимодействие типа "терминал - терминал", "терминал - компьютер", "компьютер - компьютер". В Европе активно разрабатывалась технология построения сетей с коммутацией пакетов на основе протоколов Х.25.

К середине 70-х годов наряду с мэйнфреймами широкое распространение получили мини-компьютеры (PDP-11, Nova, HP). Архитектура мини-компьютеров была значительно проще, многие функции мультипрограммных ОС мэйнфреймов были усечены. Операционные системы мини-ЭВМ стали делать специализированными (RSX-11M - разделение времени, RT-11 - OC реального времени) и не всегда многопользовательскими.

Важной вехой в истории мини-компьютеров и вообще в истории операционных систем явилось создание ОС UNIX. Написал эту систему Кен Томпсон (Ken Thompson), один из специалистов по компьютерам в BELL Labs, работавший над проектом MULTICS . Собственно его UNIX – это усеченная однопользовательская версия системы MULTICS. Первоначальное название этой системы – UNICS (UNiplexed Information and Computing Service) – примитивная информационная и компьютерная служба. Так в шутку была названа эта система, поскольку MULTICS (MULTiplexed Information and Computing Service) – мультиплексная информационная и компьютерная служба. С середины 70-х годов началось массовое использование ОС UNIX, написанной на 90% на языке С. Широкое распространение С-компиляторов сделало UNIX уникальной переносимой OC, а поскольку она поставлялась вместе с исходными кодами, то она стала первой открытой операционной системой. Гибкость, элегантность, мощные функциональные возможности и открытость позволили ей занять прочные позиции во всех классах компьютеров – от персональных до супер-ЭВМ.

Доступность мини-компьютеров послужила стимулом для создания локальных сетей. В простейших ЛВС компьютеры соединялись через последовательные порты. Первое сетевое приложение для ОС UNIX – программа UUCP (Unix to Unix Copy Program) появилась в 1976 году.

Дальнейшее развитие сетевых систем со стеком протоколов TCP/IP. В 1983 году он был принят MO США в качестве стандарта и использован в сети ARPANET. В этом же году ARPANET разделилась на MILNET (для военного ведомства США) и новую ARPANET, которую стали называть Internet.

Все восьмидесятые годы характерны появлением все более совершенных версий UNIX: Sun OS, HP-UX, Irix, AIX и др. Для решения проблемы их совместимости были приняты стандарты POSIX и XPG, определяющие интерфейсы этих систем для приложений.

Еще одним знаменательным событием для истории операционных систем было появление в начале 80-х годов персональных компьютеров. Они послужили мощным толчком для распределения локальных сетей, в результате поддержка сетевых функций стала для ОС ПК необходимым условием. Однако и дружественный интерфейс, и сетевые функции появились у ОС ПК не сразу [37].

Наиболее популярной версией ОС раннего этапа развития персональных компьютеров была MS-DOS компании Microsoft – однопрограммная, однопользовательская ОС с интерфейсом командной строки. Многие функции, обеспечивающие удобство работу пользователю, в этой ОС предоставлялись дополнительными программами - оболочкой Norton Commander, PC Tools и др. Наибольшее влияние на развитие программного обеспечения ПК оказала операционная среда Windows, первая версия которой появилась в 1985 году. Сетевые функции также реализовались с помощью сетевых оболочек и появились в MS-DOS версии 3.1. В это же время появились сетевые продукты Microsoft - MS-NET, а позже - LAN Manager, Windows for Workgroup, а затем и Windows NT.

Другим путем пошла компания Novell, ее продуктом NetWare является операционная система со встроенными сетевыми функциями. ОС NetWare, распространялась как операционная система для центрального сервера локальной сети и за счет специализации функций файл-сервера обеспечивала высокую скорость удаленного доступа к файлам и повышенную безопасность данных. Однако эта ОС имела специфический программный интерфейс (API), что затрудняло разработку приложений.

В 1987 году появилась первая многозадачная ОС для ПК – OS/2, разработанная Microsoft совместно с IBM. Эта была хорошо продуманная система с виртуальной памятью, графическим интерфейсом и позволяла выполнять DOS-приложения. Для этой ОС получили распространение сетевые оболочки LAN Manager(Microsoft) и LAN Server (IBM). Эти оболочки уступали по производительности файловому серверу NetWare и потребляли больше аппаратных ресурсов, но имели важные достоинства. Они позволяли выполнять на сервере любые программы, разработанные для OS/2, MS-DOS и Windows. Кроме того, можно было использовать компьютер, на котором они работали, в качестве рабочей станции. Однако неудачная рыночная судьба OS/2 не позволила системам LAN-Manager и LAN-Server захватить заметную долю рынка, но принципы работы этих сетевых систем во многом нашли свое воплощение в ОС 90-х годов - MS Windows NT.

В 80-е годы были приняты основные стандарты на коммуникационные технологии для локальных сетей: в 1980 г. – Ethernet, в 1985 г. – Token King, в конце 80-х FDDI (Fiber Distributed Data Interface) – распределенный интерфейс передачи данных по волоконно-оптическим каналам, двойное кольцо с маркером. Это позволило обеспечить совместимость сетевых ОС на нижних уровнях, а также стандартизировать операционные системы с драйверами сетевых адаптеров.

Для ПК применялись не только специально разработанные для них ОС (MS-Dos, NetWare, OS/2), но и адаптировались уже существующие ОС, в частности UNIX. Наиболее известной системой этого типа была версия UNIX компании Santa Cruz Operation (SCO UNIX).

В 90-е годы практически все операционные системы, занимающие заметное место на рынке, стали сетевыми. Сетевые функции встраиваются в ядро ОС, являясь ее неотъемлемой частью. В ОС используются средства мультиплексирования нескольких стеков протоколов, за счет которого компьютеры могут поддерживать одновременную работу с разнородными серверами и клиентами. Появились специализированные ОС, например сетевая ОС IOS компании Cisco System, работающая в маршрутизаторах.

Во второй половине 90-х годов все производители ОС усилили поддержку средств работы с интерфейсами. Кроме стека протоколов TCP/IP в комплект поставки начали включать утилиты, реализующие популярные сервисы Интернета: telnet, ftp, DNS, Web и др.

Особое внимание в последнем десятилетии и в настоящее время уделялось корпоративным сетевым операционным системам. Это одна из наиболее важных задач в обозримом будущем. Корпоративные ОС должны быть хорошо и устойчиво работать в крупных сетях, которые характерны для крупных организаций (предприятий, банков и т.п.), имеющих отделения во многих городах и, возможно, в разных странах. Корпоративная ОС должна беспроблемно взаимодействовать с ОС разного типа и работать на различных аппаратных платформах. Сейчас определилась лидеры в классе корпоративных ОС - это MS Windows 2000/2003, UNIX и Linux-системы, а также Novell NetWare 6.5.

**3.4. Назначение состав и функции ОС**

В настоящее время существует большое количество различных типов операционных систем, отличающихся областями применения, аппаратными платформами, способами реализации и др. Назначение операционных систем можно разделить на четыре основные составляющие [20, 37. 39.].

1. Организация (обеспечение) удобного интерфейса между приложениями и пользователями с одной стороны, и аппаратурной компьютера с другой стороны. Вместо реальной аппаратуры компьютера ОС представляет пользователю расширенную виртуальную машину, с которой удобнее работать и которую легче программировать. Вот список основных сервисов, предоставляемых типичными операционными системами.

* 1. *Разработка программ*: ОС представляет программисту разнообразные инструменты разработки приложений: редакторы, отладчики и т.п. Если необязательно знать, как функционируют различные электронные и электромеханические узлы и устройства компьютера. Часто пользователь может не узнать даже системы команд процессора, поскольку он может обойтись мощными высокоуровневыми функциями, которые представляет ОС.
  2. *Исполнение программ.* Для запуска программы нужно выполнить ряд действий: загрузить в основную память программу и данные, инициализировать устройства ввода-вывода и файлы, подготовить другие ресурсы. ОС выполняет всю эту рутинную работу вместо пользователя.
  3. *Доступ к устройствам ввода-вывода*. Для управления каждым устройством используется свой набор команд. ОС предоставляет пользователю единообразный интерфейс, который скрывает все эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения и записи. Если бы программист работал непосредственно с аппаратурой компьютера, то для организации, например, чтение блока данных с диска ему пришлось бы использовать более десятка команд с указанием множества параметров. После завершения обмена, программист должен был бы предусмотреть еще более сложный анализ результата выполненной операции.
  4. *Контролируемый доступ к файлам*. При работе с файлами управление со стороны ОС предполагает не только глубокий учет природы устройства ввода-вывода, но и знание структур данных, записанных в файлах. Многопользовательские ОС, кроме того, обеспечивают механизм защиты при обращении к файлам.
  5. *Системный доступ*. ОС управляет доступом к совместно используемой или общедоступной вычислительной системе в целом, а также к отдельным системным ресурсам. Она обеспечивает защиту ресурсов и данных от несанкционированного использования и разрешает конфликтные ситуации.
  6. *Обнаружение ошибок и их обработка*. При работе компьютерной системы могут происходить разнообразные сбои за счет внутренних и внешних ошибок в аппаратном обеспечении, различного рода программных ошибок (переполнение, попытка обращения к ячейке памяти, доступ к которой запрещен и др.). В каждом случае ОС выполняет действия, минимизирующие влияние ошибки на работу приложения (от простого сообщения об ошибке до аварийной остановки программы).
  7. *Учет использования ресурсов.*  Хорошая ОС имеет средства учета использования различных ресурсов и отображения параметров производительности вычислительной системы. Эта информация важна для настройки (оптимизации) вычислительной системы с целью повышения ее производительности.

В результате реальная машина, способная выполнить только небольшой набор элементарных действий (машинных команд), с помощью операционной системы превращается в виртуальную машину, выполняющую широкий набор гораздо более мощных функций. Виртуальная машина тоже управляется командами, но уже команды более высокого уровня. Например, такие как удалить файл с определенным именем, запустить на выполнение прикладную программу, повысить приоритет задачи, вывести текст файла на печать и т.д. Таким образом, назначение ОС состоит в предоставлении пользователю (программисту) некоторой расширенной виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющий реальный компьютер, систему или сеть.

2. Организация эффективного использования ресурсов компьютера. ОС не только представляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является своеобразным диспетчером ресурсов компьютера. К числу основных ресурсов современных вычислительных систем относятся процессоры, основная память, таймеры, наборы данных, диски, накопители на МЛ, принтеры, сетевые устройства, и др. ресурсы определяются операционной системой между выполняемыми программами. В отличие от программы, которая является статическим объектом, выполняемая программа – это динамический объект, такой объект называется процессом и является базовым понятием современных ОС.

Управление ресурсами вычислительной системы с целью наиболее эффективного их использования является вторым назначением операционной системы. *Критерий эффективности,* в соответствии с которым ОС организует управление ресурсами компьютера, может быть различным. Например, в одних системах важен такой критерий, как пропускная способность вычислительной систем, в других - время ее реакции. Зачастую ОС должны удовлетворять нескольким, противоречащим друг другу критериям, что доставляет разработчикам серьезные трудности.

Управление ресурсами включает решение ряда общих, независящих от типа ресурса задач:

2.1*. Планирование ресурса* – определение, какому процессу, когда и в каком качестве (если ресурс может выделяться частями) следует выделить данный ресурс.

*2.2. Удовлетворение запросов на ресурсы* – выделение ресурса процессам.

2.3*. Отслеживание состояния и учет использования ресурса* - поддержание оперативной информации о занятости ресурса и распределенной его доли.

2.4*. Разрешение конфликтов между процессами,* претендующими на один и тот же ресурс.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых, в конечном счете, определяют облик ОС в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Таким образом, управление ресурсами составляют важное назначение ОС. В отличие от функций расширенной виртуальной машины большинство функций управления ресурсами выполняются операционной системой автоматически и прикладному программисту недоступны.

3. Облегчение процессов эксплуатации аппаратных и программных средств вычислительной системы. Ряд операционных систем имеют в своем составе наборы служебных программ, обеспечивающие резервное копирование, архивацию данных, проверку, очистку и дефрагментацию дисковых устройств и др.

Кроме того, современные ОС имеют достаточно большой набор средств и способов диагностики и восстановление работоспособности системы. Сюда относятся:

* диагностические программы для выявления ошибок в конфигурации ОС;
* средства восстановления последней работоспособной конфигурации;
* средства восстановления поврежденных и пропавших системных файлов и др.

Следует отметить еще одно назначение ОС.

4. Возможность развития. Современные ОС организуются таким образом, что допускают эффективную разработку, тестирования и внедрения новых системных функций, не прерывая процесса нормального функционирования вычислительной системы. Большинство операционных систем постоянно развиваются (нагляден пример Windows). Происходит это в силу следующих причин:

4.1*. Обновление и возникновение новых видов аппаратного обеспечения*. Например, ранние версии ОС UNIX и OS/2 не использовали механизмы страничной организации памяти (что это такое мы рассмотрим позже), потому, что они работали на машинах, не обеспеченных соответствующими аппаратными средствами.

4.2*. Новые сервисы*. Для удовлетворения пользователей или нужд системных администраторов ОС должны постоянно предоставлять новые возможности. Например, может потребоваться добавить новые инструменты для контроля ли оценки производительности, новые средства ввода-вывода данных, например, речевой ввод. Другой пример - поддержка новых приложений, использующих окна на экране дисплея.

4.3. *Исправления.* В каждой ОС есть ошибки. Время от времени они обнаруживаются и исправляются. Отсюда постоянные появления новых версий и редакций ОС. Необходимость регулярных изменений накладывается определенные требования на организацию операционных систем. Очевидно, что эти системы (как впрочем, и другие сложные программы системы) должны иметь модульную структуру с четко определенными межмодульными связями (интерфейсами). Важную роль играет хорошая и полная документированность системы.

Перейдем к рассмотрению состава компонентов и функций ОС. Современные операционные системы содержат сотни и тысячи модулей (например, W2000 содержит 29 млн. строк исходного кода на языке С). Функции ОС обычно группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам. Совокупности модулей, выполняющих такие группы функций, образуют подсистемы операционной системы.

Наиболее важными подсистемами управления ресурсами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

*Управление процессами.* Подсистема управления процессами непосредственно влияет на функционирование вычислительной системы. Для каждой выполняемой программы ОС организует один или более процессов. Каждый такой процесс представляется в ОС информационной структурой (таблицей, дескриптором, контекстом процессора), содержащей данные о потребностях процесса в ресурсах, а также о фактически выделенных ему ресурсов (области оперативной памяти, количестве процессорного времени, файлах, устройствах ввода-вывода и др.). Кроме того, в этой информационной структуре хранятся данные, характеризующие историю пребывания процесса в системе: текущее состояние (активное или заблокированное), приоритет, состояние регистров, программного счетчика и др.

В современных мультипрограммных ОС может существовать одновременно несколько процессов, порожденных по инициативе пользователей и их приложений, а также инициированных ОС для выполнения своих функций (системные процессы). Поскольку процессы могут одновременно претендовать на одни и те же ресурсы, подсистема управления процессами планирует очередность выполнения процессов, обеспечивает их необходимыми ресурсами, обеспечивает взаимодействие и синхронизацию процессов.

*Управление памятью.* Подсистема управления памятью производит распределение физической памяти между всеми существующими в системе процессами, загрузку и удаление программных кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса. Стратегия управления памятью складывается из стратегий выборки, размещения и замещения блока программы или данных в основной памяти. Соответственно используются различные алгоритмы, определяющие, когда загрузить очередной блок в памяти (по запросу или с упреждением), в какое место памяти его поместить, и какой блок программы или данных удалить из основной памяти, чтобы освободить место для размещения новых блоков.

Одним из наиболее популярных способов управления памятью в современных ОС является виртуальная память. Реализация механизма виртуальной памяти позволяет программисту считать, что в его распоряжении имеется однородная оперативная память, объем которой ограничивается только возможностями адресации, предоставляемые системой программирования.

Важная функция управления памятью - защита памяти. Нарушение в защите памяти связано с обращениями процессов к участкам памяти, выделенной другими процессами прикладных программ или программ самой ОС. Средства защиты памяти должны пресекать такие попытки доступа путем аварийного завершения программы нарушителя.

*Управление файлами.* Функции управления файлами сосредоточены в файловой системе ОС. Операционная система виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде файла - простой неструктурированной последовательности байтов, имеющих символьное имя. Для удобства работы с данными файлы группируются в каталоги, которые, в свою очередь, образуют группы - каталоги более высокого уровня. Файловая система преобразует символьные имена файлов, с которыми работает пользователь или программист, в физические адреса данных на дисках, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

*Управление внешними устройствами*. Функции управления внешними устройствами возлагается на подсистему управления внешними устройствами, называемую также подсистемой ввода-вывода. Она является интерфейсом между ядром компьютера и всеми подключенными к нему устройствами. Спектр этих устройств очень обширен (принтеры, сканеры, мониторы, модемы, манипуляторы, сетевые адаптеры, АЦП разного рода и др.) сотни моделей этих устройств отличаются набором и последовательностью команд, используемых для обмена информацией с процессором и другими деталями.

Программа, управляющая конкретной моделью внешнего устройства и учитывающая все его особенности, называется драйвером. Наличие большого количества подходящих драйверов во многом определяет успех ОС на рынке. Создание драйверов занимаются как разработчики ОС, так и компании, выпускающие внешние устройства. ОС должна поддерживать четко определенный интерфейс между драйверами и остальными частями ОС. Тогда разработчики компаний-производителей устройств ввода-вывода могут поставлять вместе со своими устройствами драйверы для конкретной операционной системы.

*Защита данных и администрирование*. Безопасность данных вычислительной системы обеспечивается средствами отказоустойчивости ОС, направленными на защиту от сбоев и отказов аппаратуры и ошибок программного обеспечения, а также средствами защиты от несанкционированного доступа. Для каждого пользователя системы обязательна процедура логического входа, в процессе которой ОС убеждается, что в систему входит пользователь, разрешенный административной службой. Администратор вычислительной системы определяет и ограничивает возможности пользователей в выполнении тех или иных действий, т.е. определяет их права по обращению и использованию ресурсов системы.

Важным средством защиты являются функции аудита ОС, заключающегося в фиксации всех событий, от которых зависит безопасность системы. Поддержки отказоустойчивости вычислительной системы реализуется на основе резервирования (дисковые RAID -массивы, резервные принтеры и другие устройства, иногда резервирование центральных процессоров, в ранних ОС - дуальные и дуплексные системы, системы с мажоритарным органом и др.). Вообще обеспечение отказоустойчивости системы – одна из важнейших обязанностей системного администратора, который для этого использует ряд специальных средств и инструментов [25].

*Интерфейс прикладного программирования*. Прикладные программисты используют в своих приложениях обращения к операционной системе, когда для выполнения тех или иных действий им требуется особый статус, которым обладает только ОС. Возможности операционной системы доступны программисту в виде набора функций, который называется интерфейсом прикладного программирования (Application Programming Interface, API). Приложения обращаются к функциям API с помощью системных вызовов. Способ, которым приложение получает услуги операционной системы, очень похож на вызов подпрограмм.

Способ реализации системных вызовов зависит от структурной организации ОС, особенностей аппаратной платформы и языка программирования.

В ОС UNIX системные вызовы почти идентичны библиотечным процедурам. Ситуация в Windows иная (более подробно это рассмотрим далее).

*Пользовательский интерфейс.* ОС обеспечивает удобный интерфейс не только для прикладных программ, но и для пользователя (программиста, администратора). В ранних ОС интерфейс сводился к языку управления заданиями и не требовал терминала. Команды языка управления заданиями набивались на перфокарты, а результаты выполнения задания выводились на печатающее устройство.

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифровыми и графическими. При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, развитость которой отражает функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с каталогами и файлами, получать информацию о состоянии ОС, администрировать систем команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого командного файла содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательности команд из командного файла, иногда называют командным интерпретатором (в MS-DOS командным процессором).

Вычислительные системы, управляемые из командной строки, например, UNIX-системы имеют командный интерпретатор, называемый оболочкой (Shell). Она собственно не входит в состав ОС, но пользуется многими функциями операционной системы. Когда какой либо пользователь входит в систему, запускается оболочка. Стандартным терминалом для нее является монитор с клавиатурой. Оболочка начинает работу с печати приглашения (prompt) – знака доллара (или иного знака), говорящего пользователю, что оболочка ожидает ввода команды (аналогично управляется MS-DOS). Если теперь пользователь напечатает какую либо команду, оболочка создает системный вызов, и ОС выполнит эту команду. После завершения оболочка опять печатает приглашение и пытается прочесть следующую входную строку.

Ввод команд может быть упрощен, если операционная система поддерживает графический пользовательский интерфейс. В этом случае пользователь выбирает на экране нужный пункт меню или графический символ (так это происходит, например, в ОС Windows).

**3.5. Архитектура операционной системы**

Под архитектурной операционной системы понимают структурную и функциональную организацию ОС на основе некоторой совокупности программных модулей. В состав ОС входят исполняемые и объектные модули стандартных для данной ОС форматов, программные модули специального формата (например, загрузчик ОС, драйверы ввода-вывода), конфигурационные файлы, файлы документации, модули справочной системы и т.д.

На архитектуру ранних операционных систем обращалось мало внимания, во-первых, ни у кого не было опыта в разработке больших программных систем, а во-вторых, проблема взаимозависимости и взаимодействия модулей недооценивались. В подобных монолитных ОС почти все процедуры могли вызывать одной другою. Такое отсутствие структуры было несовместимо с расширением операционных систем. Первая версия ОС OS/360 была создана коллективом из 5000 человек за 5 лет и содержала более 1 млн. строк кода. Разработанная несколько позже операционная система Mastics содержала к 1975 году уже 20 млн. строк. Стало ясно, что разработка таких систем должна вестись на основе модульного программирования.

Большинство современных ОС представляют собой хорошо структурированное модульные системы, способные к развитию, расширению и переносу на новые платформы. Какой-либо единой унифицированной архитектуры ОС не существует, но известны универсальные подходы к структурированию ОС. Принципиально важными универсальными подходами к разработке архитектуры ОС являются [4, 8, 18, 24, 28,37]:

* модульная организация;
* функциональная избыточность;
* функциональная избирательность;
* параметрическая универсальность;
* концепция многоуровневой иерархической вычислительной системы, по которой ОС представляется многослойной структурой;
* разделение модулей на 2 группы по функциям – ядро – модули, выполняющие основные функции ОС, и модули, выполняющие вспомогательные функции ОС;
* разделение модулей ОС на 2 группы по размещению в памяти вычислительной системы – резидентные, постоянно находящиеся в оперативной памяти, и транзитные, загружаемые в оперативную память только на время пополнения своих функций;
* реализация двух режимов работы вычислительной системы – привилегированного режима (или режима ядра – Kernel mode), или режима супервизора (supervisor mode), и пользовательского режима (user mode) или режима задача (task mode);
* ограничение функций ядра (а, следовательно, и количества модулей ядра) до минимального количества необходимых самых важных функций.

Первые ОС разрабатывались как монолитные системы без четко выраженной структуры (рис. 3.4).

Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их вместе в единый объектный файл с помощью компоновщика (примерами могут служить ранние версии ядра UNIX или Novell NetWare). Каждая процедура видит любую другую процедуру (в отличие от структуры, содержащей модули, в которой большая часть информации является локальной для модуля, и процедуры модуля можно вызвать только через специально определенные точки входа).

Однако даже такие монолитные системы могут быть немного структурированными. При обращении к системным вызовам, поддерживаемым ОС, параметры помешаются в строго определенные места, такие, как регистры или стек, а затем выполняется специальная команда прерывания, известная как вызов ядра или вызов супервизора. Эта команда переключает машину из режима пользователя в режим ядра, называемый также режимом супервизора, и передает управление ОС. Затем ОС проверяет параметры вызова для того, чтобы определить, какой системный вызов должен быть выполнен. После этого ОС индексирует таблицу, содержащую ссылки на процедуры, и вызывает соответствующую процедуру.

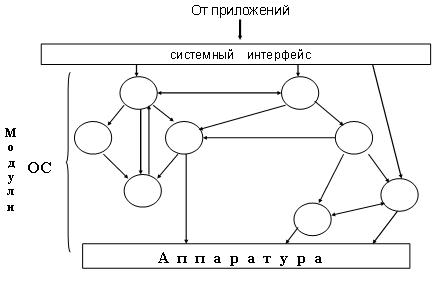


Рис. 1.4

Такая организация ОС предполагает следующую структуру [28]:

* главная программа, которая вызывает требуемые сервисные процедуры;
* набор сервисных процедур, реализующих системные вызовы;
* набор утилит, обслуживающих сервисные процедуры.

В этой модели для каждого системного вызова имеется одна сервисная процедура. Утилиты выполняют функции, которые нужны нескольким сервисным процедурам. Это деление процедур на три слоя показано на рис. 3.5.

Классической считается архитектура ОС, основанная на концепции иерархической многоуровневой машины, привилегированном ядре и пользовательском режиме работы транзитных модулей. Модули ядра выполняют базовые функции ОС: управление процессами, памятью, устройствами ввода-вывода и т.п. Ядро составляет сердцевину ОС, без которой она является полностью неработоспособной и не может выполнить ни одну из своих функций. В ядре решаются внутрисистемные задачи организации вычислительного процесса, недоступные для приложения.

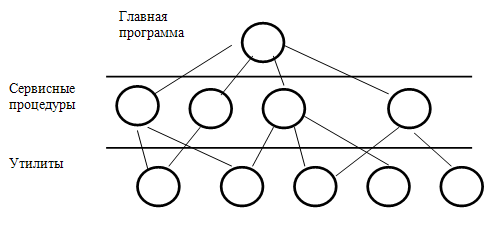


Рис.3.5

Особый класс функций ядра служит для поддержки приложений, создавая для них так называемою прикладную программную среду. Приложения могут обращаться к ядру запросами – системными вызовами – для выполнения тех или иных действий, например, открытие и чтение файла, получение системного времени, вывода информации на дисплей и т.д. Функции ядра, которые могут вызываться приложениями, образуют интерфейс прикладного программирования – API (Application Programming Interface).

Для обеспечения высокой скорости работы ОС модули ядра (по крайней мере, большая их часть) являются резидентными и работают в привилегированном режиме (Kernel mode). Этот режим обеспечивает, во-первых, безопасность работы самой ОС от вмешательства приложений, и, во-вторых, возможность работы модулей ядра с полным набором машинных инструкций, позволяющих собственно ядру выполнять управление ресурсами компьютера, в частности переключение процессора с задачи на задачу, управлением устройствами ввода-вывода, распределением и защитой памяти и др.

Остальные модули ОС выполняют не столь важные, как ядро, функции и являются транзитными. Например, это могут быть программы архивирования данных, дефрагментации диска, сжатие дисков, очистки дисков и т.п.

Вспомогательные модули обычно подразделяются на группы:

* утилиты – программы, выполняющие отдельные задачи управления и сопровождения вычислительной системы;
* системные обрабатывающие программы – текстовые и графические редакторы (Paint, Imaging в Windows 2000), компиляторы и др.;
* программы представления пользователю дополнительных услуг (специальный вариант пользовательского интерфейса, калькулятор, игры, средства мультимедиа Window 2000);
* библиотеки процедур различного назначения, упрощения разработку приложений, например, библиотека функций ввода-вывода, библиотека математических функций и т.п.

Эти модули ОС оформляются как обычные приложения, обращаются к функциям ядра посредством системных вызовов и выполняются в пользовательском режиме (user mode). В этом режиме запрещается выполнение некоторых команд, которые связаны с функциями ядра ОС (управление ресурсами, распределение и защита памяти и т.п.).

В концепции многоуровневой (многослойной) иерархической машины структура ОС также представляется рядом слоев. При такой организации каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс. На основе этих функций следующий верх по иерархии слой строит свои функции – более сложные и более мощные и т.д. Такая организация системы существенно упрощает ее разработку, т.к. позволяет сначала «сверху вниз» определить функции слоев и межслойные интерфейсы, а при детальной реализации, двигаясь в «снизу вверх» наращивать мощность функции слоев. Кроме того, модули каждого слоя можно изменять без необходимости изменений в других слоях (но, не меняя межслойных интерфейсов!).

Многослойная структура ядра ОС может быть представлена, например, вариантом, показанным на рис. 3.6.

В данной схеме выделены следующие слои:

1. *Средства аппаратной поддержки ОС.* Значительной часть функций ОС может выполняться аппаратными средствами [24]. Чисто программные ОС сейчас не существуют. Как правило, в современных системах всегда есть средство аппаратной поддержки ОС, которые прямо участвуют в организации вычислительных процесса. К ним относятся: система прерываний, средство поддержки привилегированного режима, средства поддержки виртуальной памяти, системный таймер, средство переключения контекстов процессов (информация о состоянии процесса в момент его приостановки), средство защиты памяти и др.

2. *Машинно-зависимые модули ОС.* Этот слой образует модули, в которых отражается специфика аппаратной платформы компьютера. Назначение этого слоя – «экранирование» выше лежащих слоев ОС от особенностей аппаратуры (например, Windows 2000 это слой HAL – Hardware Abstraction Layer – уровень аппаратных абстракций).

**Аппаратура**

Средства аппарат.

поддержки ОС

Машинно-зависимые

модули ядра ОС

Базовые механизмы

ядра

Менеджеры ресурсов

Файловая система, вирт. память

Файловая сис., вирт. память и др.

Интерфейс системных

вызовов API

Утилиты, системные программы

Приложения пользователей

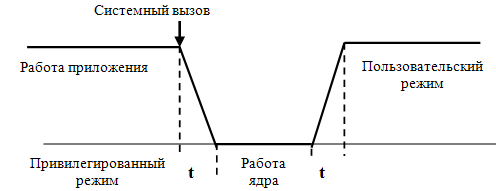
Рис. 3.6

3. *Базовые механизмы ядра.* Этот слой модулей выполняет наиболее примитивные операции ядра: программное переключение контекстов процессов, диспетчерскую прерываний, перемещение страниц между основной памятью и диском и т.п. Модули этого слоя не принимают решений о распределении ресурсов, а только обрабатывают решения, принятые модули вышележащих уровней. Поэтому их часто называют исполнительными механизмами для модулей верхних слоев ОС.

4. *Менеджеры ресурсов*. Модули этого слоя выполняют стратегические задачи по управлению ресурсами вычислительной системы. Это менеджеры (диспетчеры) процессов, ввода-вывода, оперативной памяти и файловой системы. Каждый менеджер ведет учет свободных и используемых ресурсов и планирует их распределение в соответствии запросами приложений.

5. *Интерфейс системных вызовов*. Это верхний слой ядра ОС, взаимодействующий с приложениями и системными утилитами, он образует прикладной программный интерфейс ОС. Функции API обслуживающие системные вызовы, предоставляют доступ к ресурсам системы в удобной компактной форме, без указания деталей их физического расположения.

Повышение устойчивости ОС обеспечивается переходом ядра в привилегированный режим. При этом происходит некоторое замедление выполнение системных вызовов. Системный вызов привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению – обратное переключение. За счет этого возникает дополнительная задержка в обработке системного вызова (рис. 3.7). Однако такое решение стало классическим и используется во многих ОС (UNIX, VAX, VMS, IBM OS/390, OS/2 и др.).



Время переключения режимов

Рис. 3.7

Многослойная классическая многоуровневая архитектура ОС не лишена своих проблем. Дело в том, что значительные изменения одного из уровней могут иметь трудно предвидимое влияние на смежные уровни. Кроме того, многочисленные взаимодействия между соседними уровнями усложняют обеспечение безопасности. Поэтому, как альтернатива классическому варианту архитектура ОС, часто используется микроядерная архитектура ОС.

Суть этой архитектуры состоит в следующем. В привилегированном режиме остаются работать только очень небольшая часть ОС, называемая микроядром. Микроядро защищено от остальных частей ОС и приложений. В его состав входят машинно-зависимые модули, а также модули, выполняющие базовые механизмы обычного ядра. Все остальные более высокоуровневые функции ядра оформляются как модули, работающие в пользовательском режиме. Так менеджеры ресурсов, являющиеся неотъемлемой частью обычного ядра, становятся «периферийными» модулями, работающими в пользовательском режиме. Таким образом, в архитектуре с микроядром традиционное расположение уровней по вертикали заменяется горизонтальным. Это можно представить, как показано на рис. 3.8.

Внешние по отношению к микроядру компоненты ОС реализуются как обслуживающие процессы. Между собой они взаимодействуют как равноправные партнеры с помощью обмена сообщениями, которые передаются через микроядро. Поскольку назначением этих компонентов ОС является обслуживание запросов приложений пользователей, утилит и системных обрабатывающих программ, менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС, т.е. модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей ОС.

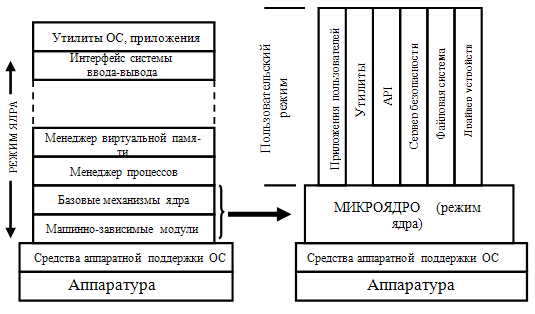


Рис. 3.8

Схематично механизм обращений к функциям ОС, оформленным в виде серверов, выглядит, как показано на рис. 3.9.

Приложение

А П П А Р А Т У Р А

МИКРОЯДРО

Сервер памяти

Файл-сервер

Принт-сервер

РЕЖИМ ПОЛЬЗОВАТЕЛЯ

РЕЖИМ ЯДРА

Запрос

Ответ

Запрос

Ответ

Сервер процессов

Сервер ввода-вывода

Рис. 3.9

Схема смены режимов при выполнении системного вызова в ОС с микроядерной архитектурой выглядит, как показано на рис. 3.10.

Приложение

Системный

вызов

МИКРОЯДРО

МИКРОЯДРО

СЕРВЕР ОС

Приложение

t

t

t

t

РЕЖИМ ЯДРА

РЕЖИМ ПОЛЬЗОВАТЕЛЯ

Рис. 3.10

Из рисунка ясно, что выполнение системного вызова сопровождается четырьмя переключениями режимов (4 t), в то время как в классической архитектуре – двумя. Следовательно, производительность ОС с микроядерной архитектурой при прочих равных условиях будит ниже, чем у ОС с классическим ядром.

В тоже время признаны следующие достоинства микроядерной архитектуры [36]:

* единообразные интерфейсы;
* простота расширяемости;
* высокая гибкость;
* возможность переносимости;
* высокая надежность;
* поддержка распределенных систем;
* поддержка объектно-ориентированных ОС.

По многим источникам вопрос масштабов потери производительности в микроядерных ОС является спорным. Многое зависит от размеров и функциональных возможностей микроядра. Избирательное увеличение функциональности микроядра приводит к снижению количества переключений между режимами системы, а также переключений адресных пространств процессов.

Может быть, это покажется парадоксальным, но есть и такой подход к микроядерной ОС, как уменьшение микроядра.

Для возможности представления о размерах микроядер операционных систем в ряде источников [37] приводятся такие данные:

* типичное микроядро первого поколения - 300 Кбайт кода, и 140 интерфейсов системных вызовов;
* микроядро ОС L4 (второе поколение) – 12 Кбайт кода и 7 интерфейсов системных вызовов.

В современных операционных системах различают следующие виды ядер.

1. Наноядро (НЯ) – крайне упрощённое и минимальное ядро, выполняет лишь одну задачу – обработку аппаратных прерываний, генерируемых устройствами компьютера. После обработки посылает информацию о результатах обработки вышележащему программному обеспечению. НЯ используются для виртуализации аппаратного обеспечения реальных компьютеров или для реализации механизма гипервизора.
2. Микроядро (МЯ) предоставляет только элементарные функции управления процессами и минимальный набор абстракций для работы с оборудованием. Бо́льшая часть работы осуществляется с помощью специальных пользовательских процессов, называемых сервисами. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Микроядерными являются ядра ОС Minix и GNU Hurd и ядро систем семейства
3. Экзоядро (ЭЯ) – предоставляет лишь набор сервисов для взаимодействия между приложениями, а также необходимый минимум функций, связанных с защитой: выделение и высвобождение ресурсов, контроль прав доступа, и т. д. ЭЯ не занимается предоставлением абстракций для физических ресурсов – эти функции выносятся в библиотеку пользовательского уровня (так называемую libOS). В отличие от микроядра ОС, базирующиеся на ЭЯ, обеспечивают большую эффективность за счет отсутствия необходимости в переключении между процессами при каждом обращении к оборудованию.
4. Монолитное ядро (МЯ) предоставляет широкий набор абстракций оборудования. Все части ядра работают в одном адресном пространстве. МЯ требуют перекомпиляции при изменении состава оборудования. Компоненты операционной системы являются не самостоятельными модулями, а составными частями одной программы. МЯ более производительно, чем микроядро, поскольку работает как один большой процесс. МЯ является большинство Unix-систем и Linux. Монолитность ядер усложняет отладку, понимание кода ядра, добавление новых функций и возможностей, удаление ненужного, унаследованного от предыдущих версий, кода. «Разбухание» кода монолитных ядер также повышает требования к объёму оперативной памяти.
5. Модульное ядро (Мод. Я) – современная, усовершенствованная модификация архитектуры МЯ. В отличие от «классических» МЯ, модульные ядра не требуют полной перекомпиляции ядра при изменении состава аппаратного обеспечения компьютера. Вместо этого они предоставляют тот или иной механизм подгрузки модулей, поддерживающих то или иное аппаратное обеспечение (например, драйверов). Подгрузка модулей может быть как динамической, так и статической (при перезагрузке ОС после переконфигурирования системы). Мод. Я удобнее для разработки, чем традиционные монолитные ядра. Они предоставляют программный интерфейс (API) для связывания модулей с ядром, для обеспечения динамической подгрузки и выгрузки модулей. Не все части ядра могут быть сделаны модулями. Некоторые части ядра всегда обязаны присутствовать в оперативной памяти и должны быть жёстко «вшиты» в ядро.
6. Гибридное ядро (ГЯ) – модифицированные микроядра, позволяющие для ускорения работы запускать «несущественные» части в пространстве ядра. Имеют «гибридные» достоинства и недостатки. примером смешанного подхода может служить возможность запуска операционной системы с монолитным ядром под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляется монолитным ядром. Данный подход сформировался в результате попыток использовать преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженный код монолитного ядра.
7. Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT. Хотя Windows NT часто называют микроядерной операционной системой, это не совсем так. Микроядро NT слишком велико (более 1 Мбайт), чтобы носить приставку «микро». Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем переда-чи сообщений, как и положено в микроядерных операционных систе-мах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром.

**1.5. Классификация операционных систем**

Все многообразие существующих (и ныне не использующихся) ОС можно классифицировать по множеству различных признаков. Остановимся на основных классификационных признаках:

1. По назначению ОС делятся на *универсальные и специализированные*. Специализированные ОС, как правило, работают с фиксированным набором программ (функциональных задач). Применение таких систем обусловлено невозможностью использования универсальной ОС по соображениям эффективности, надежности, защищенности и т.п., а также вследствие специфики решаемых задач [28].

Универсальные ОС рассчитаны на решение любых задач пользователей, но, как правило, форма эксплуатации вычислительной системы может предъявлять особые требования к ОС, т.е. к элементам ее специализации.

2. По способу загрузки можно выделить *загружаемые ОС* (большинство) и *системы, постоянно находящиеся в памяти* вычислительной системы. Последние, как правило, специализированные и используются для управления работой специализированных устройств (например, в БЦВМ баллистической ракеты или спутника, научных приборах, автоматических устройствах различного назначения и др.).

3. По особенностям *алгоритмов управления ресурсами*. Главным ресурсным системы является процессор, поэтому дадим классификацию по алгоритмам управления процессором, хотя можно, конечно, классифицировать ОС по алгоритмам управления памятью, устройствами ввода-вывода и.т.д.

3.1 Поддержка многозадачности (многопрограммности). По числу одновременно выполняемых задач ОС делятся на 2 класса: однопрограммные (однозадачные), например, MS-DOS, MSX, и многопрограммные (многозадачные), например, ОС ЕС ЭВМ, OS/360, OS/2, UNIX, Windows разных версий.

Однопрограммные ОС предоставляет пользователю виртуальную машину, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Они также имеют средства управления файлами, периферийными устройствами и средства общения с пользователем. Многозадачные ОС, кроме того, управляют разделением совместно используемых ресурсов (процессор, память, файлы и т.д.) это позволяет значительно повысить эффективной вычислительной системы.

3.2 Поддержка многопользовательского режима. По числу одновременно работающих пользователей ОС делятся: на однопользовательские (MS-DOS, Windows 3х, ранние версии OS/2) и многопользовательские (UNIX, Windows NT/2000/2003/XP/Vista).

Главное отличие многопользовательских систем от однопользовательских – наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что может быть однопользовательская мультипрограммная система.

3.3 Виды многопрограммной работы. Специфику ОС во многом определяет способ распределения времени между несколькими одновременно существующими в системе процессами (или потоками). По этому признаку можно выделить 2 группы алгоритмов: не вытесняющая многопрограммность (Windows3.x, NetWare) и вытесняющая многопрограммность (Windows 2000/2003/XP, OS/2, Unix).

В первом случае активный процесс выполняется до тех пор, пока он сам не отдает управление операционной системе. Во втором случае решение о переключении процессов применяет операционная система. Возможен и такой режим многопрограммности, когда ОС разделяет процессорное время между отдельными ветвями (потоками, волокнами) одного процесса.

3.4 Многопроцессорная обработка. Важное свойство ОС – отсутствие или наличие средств поддержки многопроцессорной обработки. По этому признаку можно выделить ОС: без поддержки мультипроцессирования (Windows 3.x, Windows 95) и с поддержкой мультипроцессирования (Solaris, OS/2, UNIX, Windows NT/2000/2003/XP).

Многопроцессорные ОС классифицируются по способу организации вычислительного процесса на асимметричные ОС (выполняются на одном процессоре, распределяя прикладные задачи по остальным процессорам) и симметричные ОС (децентрализованная система).

4. По *области использования и форме эксплуатации*. Обычно здесь выделяют три типа в соответствии с использованными при их разработке критериями эффективности:

* системы пакетной обработки (OS/360, OC EC);
* системы разделения времени (UNIX, VMS);
* системы реального времени (QNX, RT/11).

Первые предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Критерий создания таких ОС - максимальная пропуская способность при хорошей загрузке всех ресурсов компьютера. В таких системах пользователь отстранен от компьютера.

Системы разделения времени обеспечивают удобство и эффективность работы пользователя, который имеет терминал и может вести диалог со своей программной.

Системы реального времени предназначены для управления техническими объектами (станок, спутник, технологический процесс, например доменный и т.п.), где существует предельное время на выполнение программ, управляющих объектом.

5. *По аппаратной платформе* (типу вычислительной техники), для которой предназначаются операционные системы, их делят а следующие группы:

5.1 Операционные системы для смарт-карт. Некоторые из них могут управлять только одной операцией, например, электронным платежом. Некоторые смарт-карты являются JAVA-ориентированным и содержат интерпретатор виртуальной машины JAVA. Апплеты JAVA загружаются на карту и выполняются JVM –интерпретатором. Некоторые из таких карт могут одновременно управлять несколькими апплетами JAVA, что приводит к многозадачности и необходимости планирования.

5.2 Встроенные операционные системы. Управляют карманными компьютерами (Palm OS, Windows CE – Consumer Electronics – бытовая техника), мобильными телефонами, телевизорами, микроволновыми печами и т.п.

5.3 Операционные системы для персональных компьютеров, например, Windows 9.x, Windows ХР, Linux, Mac OSX и др.

5.4 Операционные системы мини-ЭВМ, например, RT-11 для PDP-11 – OC реального времени, RSX-11 M для PDP-11 – ОС разделение времени, UNIX для PDP-7.

5.5 Операционные системы мэйнфреймов (больших машин), например, OS/390, происшедшая от OS/360 (IBM). Обычно ОС мэйнфреймов предполагает одновременно три вида обслуживания: пакетную обработку, обработку транзакций, (например, работа с БД, бронирование авиабилетов, процесс работы в банках) и разделение времени.

5.6 Серверные операционные системы, например, UNIX, Windows 2000, Linux. Область применения – ЛВС, региональные сети, Intranet, Internet.

5.7 Кластеры операционные системы. Кластер – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системной, например, Windows 2000 Cluster Server, Windows 2008 Server, Sun Cluster (базовая ОС – Solaris).

**1.6. Эффективность и требования, предъявляемые к ОС**

К операционным системам современных компьютеров предъявляется ряд требований. Главным требованием является выполнение основных функций эффективного управления ресурсами и обеспечения удобного интерфейса для пользователя и прикладных программ. Современная ОС должна поддерживать мультипрограммную обработку, виртуальную память, свопинг, развитый интерфейс пользователя (многооконный графический, аудио, меню ориентированный и т.д.), высокую степень защиты, удобство работы, а также выполнять многие другие необходимые функции и услуги. Кроме этих требований функциональной полноты, к ОС предъявляется ряд важных эксплуатационных требований.

1. *Эффективность*. Под эффективностью вообще любой технической (да и не только технической) системы понимается степень соответствия системы своему назначению, которая оценивается некоторым множеством показателей эффективности [24].

Поскольку ОС представляет собой сложную программную систему, она использует для собственных нужд значительную часть ресурсов компьютера. Часто эффективность ОС оценивают ее производительностью (пропускной способностью) – количеством задач пользователей, выполняемых за некоторый промежуток времени, временем реакции на запрос пользователя и др.

На все эти показатели эффективности ОС влияет много различных факторов, среди которых основными являются архитектура ОС, многообразие ее функций, качество программного кода, аппаратная платформа (компьютер) и др.

2. *Надежность и отказоустойчивость*. Операционная система должна быть, по меньшей мере, так же надежна, как компьютер, на котором она работает. Система должна быть защищена как от внутренних, так и от внешних, сбоев и отказов. В случаи ошибки в программе или аппаратуре система должна обнаружить ошибку и попытаться исправить положение или, по крайней мере, постараться свести к минимуму ущерб, нанесенный этой ошибкой пользователям.

Надежность и отказоустойчивость ОС, прежде всего, определяются архитектурными решениями, положенными в ее основу, а также отлаженностью программного кода (основные отказы и сбои ОС в основном обусловлены программными ошибками в ее модулях). Кроме того, важно чтобы компьютер имел резервные дисковые массивы, источники бесперебойного питания и др., а также программную поддержку этих средств.

3. *Безопасность (защищенность).* Каждый пользователь не хочет, чтобы другие пользователи ему мешали. ОС должна защищать пользователей и от воздействия чужих ошибок, и от попыток злонамеренного вмешательства (несанкционированного доступа). С этой целью в ОС как минимум должны быть средства аутентификации – определения легальности пользователей, авторизации – предоставление легальным пользователям установленных им прав доступа к ресурсам и аудита – фиксации всех потенциально опасных для системы событий.

Свойство безопасности особенно важно для сетевых ОС. В таких ОС к задаче контроля доступа добавляется задача защиты данных передаваемых по сети.

4. *Предсказуемость.* Требования, которые пользователь, может предъявить к системе, в большинстве случаев непредсказуемы. В то же время пользователь предпочитает, чтобы обслуживание не очень сильно менялось в течение предположительного времени. В частности, запуская свою программу в системе, пользователь должен иметь основанное на опыте работы с этой программной приблизительное представление, когда ему ожидать выдачи результатов.

5. *Расширяемость*. В отличие от аппаратных средств компьютера полезная жизнь операционных систем измеряется десятками лет. Примером может служить ОС UNIX, да и MS-DOS. Операционные системы изменяются со временем эволюционной, как правило, за счет приобретения новых свойств, например, поддержки новых типов внешних устройств или новых сетевых технологий. Если программный код модулей ОС написан таким образом, что дополнения и изменения могут вноситься без нарушения целостности системы, то такую ОС называют расширяемой. Операционная система может быть расширяемой, если при ее создании руководствовались принципами модульности, функциональной избыточности, функциональной избирательности и параметрической универсальности.

6. *Переносимость*. В идеале код ОС должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которые различаются не только типом процессора, но и способом организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа. Переносимые ОС имеют несколько вариантов реализации для разных платформ, такое свойство ОС называется также многоплатформенностью. Достигается это свойство за счет того, что основная часть ОС пишется на языке высокого уровня (например С, C++ и др.) и может быть легко перенесена на другой компьютер (машинно-независимая часть), а некоторая меньшая часть ОС (программы ядра) является машинно-зависимой и разрабатывается на машинном языке другого компьютера.

7. *Совместимость*. Существует несколько «долгоживущих» популярных ОС (разновидности UNIX, MS-DOS, Windows3.x, Windows NT, OS/2), для которых наработана широкая номенклатура приложений. Для пользователя, переходящего с одной ОС на другую, очень привлекательна возможность, выполнить свои приложения в новой операционной системе. Если ОС имеет средства для выполнения прикладных программ, написанных для других операционных систем, то она совместима с этими системами. Следует различать совместимость на уровне двоичных кодов и совместимость на уровне исходных текстов. Кроме того, понятие совместимости включает также поддержку пользовательских интерфейсов других ОС.

8. *Удобство*. Средства ОС должны быть простыми и гибкими, а логика ее работы ясна пользователю. Современные ОС ориентированы на обеспечение пользователю максимально возможного удобства при работе с ними. Необходимым условием этого стало наличие у ОС графического пользовательского интерфейса и всевозможных мастеров – программ, автоматизирующих активизацию функций ОС, подключение периферийных устройств, установку, настройку и эксплуатацию самой ОС.

9. *Масштабируемость*. Если ОС позволяет управлять компьютером с различным числом процессов, обеспечивая линейное (или почти такое) возрастание производительности при увеличении числа процессоров, то такая ОС является масштабируемой. В масштабируемой ОС реализуется симметричная многопроцессорная обработка. С масштабируемостью связано понятие кластеризации – объединение в систему двух (и более) многопроцессорных компьютеров. Правда, кластеризация направлена не столько на масштабируемость, сколько на обеспечение высокой готовности системы.

Следует заметить, что в зависимости от области применения конкретной операционной системы может изменяться и состав предъявляемых к ней требований

Производители могут предлагать свои ОС в различных, различающихся ценой и производительностью конфигурациях. Например, Microsoft продает [40]:

* Windows 2003 Server (до 4-х процессоров) – для малого и среднего бизнеса;
* Windows 2003 Advanced Server (до 8 процессоров, 2-узловой кластер) – для средних и крупных предприятий;
* Windows 2003 DataCenter Server (16-32 процессора, 4-узловой кластер) – для особо крупных предприятий.

**1.7. Совместимость и множественные прикладные среды**

В то время как многие архитектурные особенно ОС непосредственно касаются только системных программистов, концепция множественных прикладных (операционных) средств непосредственно связано с нуждами конечных пользователей – возможностью операционной системы выполнять приложения, написанные для других операционных систем. Такое свойство операционной системы называется совместимостью.

Совместимость приложений может быть на двоичном уровне и на уровне исходных текстов [28]. Приложения обычно хранятся в ОС в виде исполняемых файлов, содержащих двоичные образы кодов и данных. Двоичная совместимость достигается в том случае, если можно взять исполняемую программу и запустить ее на выполнение в среде другой ОС.

Совместимость на уровне исходных текстов требует наличие соответствующего компилятора в составе программного обеспечения компьютера, на котором предполагается выполнить данное приложение, а также совместимости на уровне библиотек и системных вызовов. При этом необходимо перекомпиляция исходных текстов приложения в новый исполняемый модуль.

Совместимость на уровне исходных текстов важна в основном для разработчиков приложений, в распоряжении которых эти исходные тексты имеются. Но для конечных пользователей практическое значение имеет только двоичная совместимость, так как только в этом случаи они могут использовать один и тот же продукт в различных операционных системах и на различных машинах.

Вид возможной совместимости зависит от многих факторов. Самый главный из них – архитектура процессора. Если процессор использует тот же набор команд (возможно с добавлениями, как в случае IBM PC – стандартный набор + мультимедиа + графика + потоковые) и тот же диапазон адресов, то двоичная совместимость может быть достигнута достаточно просто. Для этого необходимо соблюдение следующих условий:

* API, который использует приложение, должны поддерживаться данной О С;
* внутренняя структура исполняемого фала приложения должна соответствовать структуре исполняемых фалов данной ОС.

Если процессоры имеют разную архитектуру, то кроме перечисленных условий, необходимо организовать эмуляцию двоичного кода. Например, широко используется эмуляция команд процессора Intel на процессоре Motorola 680x0 компьютера Macintosh. Программный эмулятор в этом случаи последовательно выбирает двоичную инструкцию процессора Intel и выполняет эквивалентную подпрограмму, написанную в инструкциях процессора Motorola. Так как у процессора Motorola нет в точности таких же регистров, флагов, внутреннего АЛУ др., как в процессорах Intel, он должен также имитировать (эмулировать) все эти элементы с использованием своих регистров или памяти.

Это простая, но очень медленная работа, поскольку одна команда Intel выполняется значительно быстрее, чем эмулирующая ее последовательность команд процессора Motorola. Выходом в таких случаях является использование так называемых прикладных программных сред или операционных сред. Одной из составляющих такой среды является набор функций интерфейса прикладного программирования API, которая ОС предоставляет своим приложениям. Для сокращения времени на выполнение чужих программ прикладные среды имитируют обращение к библиотечным функциям.

Эффективность этого подхода связана с тем, что большинство сегодняшних программ работают под управлением GUI (графических интерфейсов пользователя) типа Windows, MAC, или UNIX Motif, при этом приложения тратят 60-80% времени на выполнение функций GUI и других библиотечных вызовов ОС. Именно это свойство приложений позволяет прикладным средам компенсировать большие затраты времени, потраченные на покомандное эмулирование программ. Тщательно спроектированная программная прикладная среда имеет в своем составе библиотеки, имитирующие библиотеки GUI, но написанная на «родном коде». Таким образом, достигается существенное ускорение выполнения программ с API другой операционной системы. Иначе такой подход называют трансляцией для того, чтобы отличить его от более медленного процесса эмулирования по одной команде за раз.

Например, для Windows программы, работающей на Macintosh, при интерпретации команд процессора Intel производительность может быть очень низкой. Но когда производится вызов функции GUI, открытие окна и др. модуль ОС, реализующий прикладную среду Windows, может перехватить этот вызов и перенаправить его на перекомпилированную для процессора Motorola 680x0 подпрограмму открытия окна. В результате на таких участках кода скорость работы программы может достичь (а, возможно, и превзойти) скорость работы на своём родном процессоре.

Чтобы программа, написанная для одной ОС, могла быть выполнена в рамках другой ОС, недостаточно лишь обеспечивать совместимость API. Концепции, положенные в основу разных ОС, могут входить в противоречия друг с другом. Например, в одной ОС приложению может быть разрешено управлять устройствами ввода-вывода, в другой – эти действия являются прерогативой ОС.

Каждая ОС имеет свои собственные механизмы защиты ресурсов, свои алгоритмы обработки ошибок и исключительных ситуаций, особую структуру процессора и схему управления памятью, свою семантику доступа к файлам и графически пользовательский интерфейс. Для обеспечения совместимости необходимо организовать бесконфликтное сосуществование в рамках одной ОС нескольких способов управления ресурсами компьютера.

Существуют различные варианты построения множественных прикладных сред, отличающиеся как особенностями архитектурных решений, так и функциональными возможностями, обеспечивающими разную степень переносимости приложений. Один из наиболее очевидных вариантов реализации множественных преклонных сред основывается на стандартной многоуровневой структуре ОС.

На рис. 3.11 ОС OS1 поддерживает кроме своих «родных» приложений, приложения операционных систем OS2 и OS3. Для этого в её составе имеются специальные приложения, прикладные программные среды – которые транслируют интерфейсы «чужих» операционных систем API OS2 и API OS3 в интерфейс своей «родной» ОС – API OS1. Так, например, в случае, если бы в качестве OS2 выступала ОС UNIX, а в качестве OS1 - OS/2, для выполнения системного вызова создания процесса fork () в UNIX – приложении программная среда должна обращаться к ядру операционной системы OS/2 с системным вызовом DOS ExecPgm ().

Обычное приложение OS1

Приложение

OS2

API OS2

Приложение

OS3

API OS3

API OS1

Менеджеры ресурсов

Базовые механизмы

Машинно-независимые задачи

Пользовательский режим

Привилегированный режим

Прикладная среда OS2

Прикладная среда OS3

Рис. 3.11

К сожалению, поведение почти всех функций, составляющих API одной ОС, как правило, существенно отличается от поведения соответствующих функций другой ОС. Например, чтобы функция создания процесса в OS/2 Dos ExecPgm ( ) полностью соответствовала функции создания процесса fork ( ) в UNIX – подобных системах, её нужно было бы изменить, чтобы она поддерживала возможность копирования адресного пространства родительского процесса в пространство процесса-потомка [28].

Еще один способ построения множественных прикладных сред основан на микроядерном подходе. При этом очень важно отметить базовое общее для всех прикладных сред, механизмы операционной системы от специфических для каждой из прикладных сред высокоуровневых функций, решающих стратегические задачи. В соответствии с микро ядерной архитектурой все функции ОС реализуются микроядром и серверами пользовательского режима. Важно, что прикладная среда оформляется в виде отдельного сервера пользовательского режима и не включает базовых механизмов.

Приложения, используя API , обращаются с системными вызовами к соответствующей прикладной среде через микроядро. Прикладная среда обрабатывает запрос, выполняет его (возможно обращаясь для этого за помощью к базовым функция микроядра) и отсылает приложению результат. В ходе выполнения запроса прикладной среде приходиться, в свою очередь, обращаться к базовым механизмам ОС, реализуемым микроядром и другими серверами ОС.

Такому подходу к конструированию множественных прикладных сред присуще все достоинства и недостатки микро ядерной архитектуры, в частности:

* очень просто можно добавлять и исключать прикладные среды, что является следствием хорошей расширяемости микро ядерных ОС;
* при отказе одной из прикладных сред остальные сохраняют работоспособность, что способствует надежности и стабильности системы в целом;
* низкая производительность микроядерных ОС сказывается на скорости работы прикладных средств, а значит, и на скорости работы приложений.

В итоге следует отметить, что создание в рамках одной ОС нескольких прикладных средств для выполнения приложений различных ОС представляет собой путь, который позволяет иметь единственную версию программы и переносить ее между различными операционными системами. Множественные прикладные среды обеспечивают совместимость на двоичном уровне данной ОС с приложениями, написанные для других ОС.

**1.8. Виртуальные машины как современный подход к реализации**

**множественных прикладных сред**

Понятие «монитор виртуальных машин» (МВМ) возникло в конце 60-х как про­граммный уровень абстракции, разделявший аппаратную платформу на несколько вирту­альных машин. Каждая из этих виртуальных машин (ВМ) была настолько похожа на базовую физическую машину, что существующее программное обеспечение могло вы­полняться на ней в неизменном виде. В то время вычислительные задачи общего харак­тера решались на дорогих мэйнфреймах (типа IBM/360), и пользователи высоко оценили способность МВМ распределять дефицитные ресурсы среди нескольких приложений.

В 80-90-е годы существенно снизилась стоимость компьютерного оборудования и появились эффективные многозадач­ные ОС, что уменьшило ценность МВМ в глазах пользователей. Мэйнфреймы уступили место мини-компьютерам, а затем ПК, и нужда в МВМ отпала. В результате из компьютерной архитектуры попросту исчезли аппаратные средства для их эффективной реализации. К концу 80-х в науке и на производстве МВМ воспринимались не иначе как исторический курьез [3, 13, 15, 34. 44].

Сегодня МВМ — снова в центре внимания. Корпорации Intel, AMD, Sun Microsystems и IBM создают стратегии виртуализации, в научных лабораториях и университетах для решения проблем мобильности, обеспечения безопасности и управляемости развиваются подходы, основанные на виртуальных машинах. Что же произошло между отставкой МВМ и их возрождением?

В 90-е годы исследователи из Стэндфордского университета начали изучать возможность применения ВМ для преодоления ограничений оборудования и операционных систем. Проблемы возникли у компьютеров с массовой параллельной обработкой (Massively Parallel Processing, MPP), которые плохо поддавались программированию и не могли выполнять имеющиеся ОС. Исследователи обнаружили, что с помощью виртуальных машин можно сделать эту неудобную архитектуру достаточно похожей на существующие платформы, чтобы использовать преимущества готовых ОС. Из этого проекта вышли люди и идеи, ставшие золотым фондом компании VMware (*www.vmware.com*), первого поставщика МВМ для компьютеров массового применения.

Как ни странно, развитие современных ОС и снижение стоимости оборудования привели к появлению проблем, которые исследователи надеялись решить с помощью МВМ. Дешевизна оборудования способствовала быстрому распространению компьютеров, но они часто бывали недогруженными, требовали дополнительных площадей и усилий по обслуживанию. А следствиями роста функциональных возможностей ОС стали их неустойчивость и уязвимость.

Чтобы уменьшить влияние системных аварий и защититься от взломов, системные администраторы вновь обратились к однозадачной вычислительной модели (с одним приложением на одной машине). Это привело к дополнительным расходам, вызванным повышенными требованиями к оборудованию. Перенос приложений с разных физических машин на ВМ и консолидация этих ВМ на немногих физических платформах позволили повысить эффективность использования оборудования, снизить затраты на управление и производственные площади. Таким образом, способность МВМ к мультиплексированию аппаратных средств — на этот раз во имя консолидации серверов и организации коммунальных вычислений — снова возродила их к жизни.

В настоящее время МВМ стал не столько средством организации многозадачности, каким он был когда-то задуман, сколько решением проблем обеспечения безопасности, мобильности и надежности. Во многих отношениях МВМ дает создателям операционных систем возможность развития функциональности, невозможной в нынешних сложных ОС. Такие функции, как миграция и защита, намного удобнее реализовать на уровне МВМ, поддерживающих обратную совместимость при развертывании инновационных решений в области операционных систем при сохранении предыдущих достижений.

Виртуализация – развивающаяся технология. В общих словах, виртуализация позволяет отделить ПО от нижележащей аппаратной инфраструктуры. Фактически она разрывает связь между определенным набором программ и конкретным компьютером. Монитор виртуальных машин отделяет программное обеспечение от оборудования и формирует промежуточный уровень между ПО, выполняемым виртуальными машинами, и аппаратными средствами. Этот уровень позволяет МВМ полностью контролировать использование аппаратных ресурсов *гостевыми операционными системами* (GuestOS), которые выполняются на ВМ.

МВМ создает унифицированное представление базовых аппаратных средств, благодаря чему физические машины различных поставщиков с разными подсистемами ввода-вывода выглядят одинаково и ВМ выполняются на любом доступном оборудовании. Не заботясь об отдельных машинах с их тесными взаимосвязями между аппаратными средствами и программным обеспечением, администраторы могут рассматривать оборудование просто как пул ресурсов для оказания любых услуг по требованию.

Благодаря полной инкапсуляции состояния ПО на ВМ монитор МВМ может отобразить ВМ на любые доступные аппаратные ресурсы и даже перенести с одной физической машины на другую. Задача балансировки нагрузки в группе машин становится тривиальной, и появляются надежные способы борьбы с отказами оборудования и наращивания системы. Если нужно отключить отказавший компьютер или ввести в строй новый, МВМ способен соответствующим образом перераспределить виртуальные машины. Виртуальную машину легко тиражировать, что позволяет администраторам по мере необходимости оперативно предоставлять новые услуги.

Инкапсуляция также означает, что администратор может в любой момент приостановить или возобновить работу ВМ, а также сохранить текущее состояние виртуальной машины либо вернуть ее к предыдущему состоянию. Располагая возможностью универсальной отмены, удается легко справиться с авариями и ошибками конфигурации. Инкапсуляция является основой обобщенной модели мобильности, поскольку приостановленную ВМ можно копировать по сети, сохранять и транспортировать на сменных носителях.

МВМ играет роль посредника во всех взаимодействиях между ВМ и базовым оборудованием, поддерживая выполнение множества виртуальных машин на единой аппаратной платформе и обеспечивая их надежную изоляцию. МВМ позволяет собрать группу ВМ с низкими потребностями в ресурсах на отдельном компьютере, снизив затраты на аппаратные средства и потребность в производственных площадях.

Полная изоляция также важна для надежности и обеспечения безопасности. Приложения, которые раньше выполнялись на одной машине, теперь можно распределить по разным ВМ. Если одно из них в результате ошибки вызовет аварию ОС, другие приложения будут от нее изолированы и продолжат работу. Если же одному из приложений угрожает внешнее нападение, атака будет локализована в пределах «скомпрометированной» ВМ. Таким образом, МВМ — это инструмент реструктуризации системы для повышения ее устойчивости и безопасности, не требующий дополнительных площадей и усилий по администрированию, которые необходимы при выполнении приложений на отдельных физических машинах.

МВМ должен связать аппаратный интерфейс с ВМ, сохранив полный контроль над базовой машиной и процедурами взаимодействия с ее аппаратными средствами. Для достижения этой цели существуют разные методы, основанные на определенных технических компромиссах. При поиске таких компромиссов принимаются во внимание основные требования к МВМ — совместимость, производительность и простота. Совместимость важна потому, что главное достоинство МВМ — способность выполнять унаследованные приложения. Производительность определяет величину накладных расходов на виртуализацию –программы на ВМ должны выполняться с той же скоростью, что и на реальной машине. Простота необходима, поскольку отказ МВМ приведет к отказу всех ВМ, выполняющихся на компьютере. В частности, для надежной изоляции требуется, чтобы МВМ был свободен от ошибок, которые злоумышленники могут использовать для разрушения системы.

Вместо того чтобы заниматься сложной переработкой кода гостевой операционной системы, можно внести некоторые изменения в основную операционную систему, изменив некоторые наиболее “мешающие” части ядра. Подобный подход называется паравиртуализацией [3, 27]. Ясно, что в этом случае адаптировать ядро ОС может только автор, и например Microsoft не проявляет желания адаптировать популярное ядро Windows 2000 к реалиям конкретных виртуальных машин.

При паравиртуализации разработчик МВМ переопределяет интерфейс виртуальной машины, заменяя непригодное для виртуализации подмножество исходной системы команд более удобными и эффективными эквивалентами. Заметим, что хотя ОС нужно портировать для выполнения на таких ВМ, большинство обычных приложений могут выполняться в неизменном виде.

Самый большой недостаток паравиртуализации – несовместимость. Любая операционная система, предназначенная для выполнения под управлением паравиртуализованного монитора МВМ, должна быть портирована в эту архитектуру, для чего нужно договариваться о сотрудничестве с поставщиками ОС. Кроме того, нельзя использовать унаследованные операционные системы, а существующие машины не удается легко заменить виртуальными.

Чтобы добиться высокой производительности и совместимости при виртуализации архитектуры x86, компания VMware разработала новый метод виртуализации, который объединяет традиционное прямое выполнение с быстрой трансляцией двоичного кода «на лету». В большинстве современных ОС режимы работы процессора при выполнении обычных прикладных программ легко поддаются виртуализации, а, следовательно, их можно виртуализировать посредством прямого выполнения. Непригодные для виртуализации привилегированные режимы может выполнять транслятор двоичного кода, исправляя «неудобные» команды x86. В результате получается высокопроизводительная виртуальная машина, которая полностью соответствует оборудованию и поддерживает полную совместимость ПО.

Преобразованный код очень похож на результаты паравиртуализации. Обычные команды выполняются в неизменном виде, а команды, нуждающиеся в специальной обработке (такие, как POPF и команды чтения регистров сегмента кода), транслятор заменяет последовательностями команд, которые подобны требующимся для выполнения на паравиртуализованной виртуальной машине. Однако есть важное различие: вместо того, чтобы изменять исходный код операционной системы или приложений, транслятор двоичного кода изменяет код при его выполнении в первый раз.

Хотя трансляция двоичного кода требует некоторых дополнительных расходов, при нормальных рабочих нагрузках они незначительны. Транслятор обрабатывает лишь часть кода, и скорость выполнения программ становится сопоставимой со скоростью прямого выполнения –как только заполнится кэш-память трассировки.

Трансляция двоичного кода также помогает оптимизировать прямое выполнение. Например, если при прямом выполнении привилегированного кода часто происходит перехват команд, это может привести к существенным дополнительным расходам, поскольку при каждом перехвате управление передается от виртуальной машины к монитору и обратно. Трансляция кода может устранить многие из таких перехватов, что приведет к снижению накладных расходов на виртуализацию. Это особенно верно для центральных процессоров с длинными конвейерами команд, в частности для современного семейства x86, в котором перехват связан с высокими дополнительными расходами.

Наибольшую проблему при реализации монитора виртуальных машин создает виртуализация процессора. Архитектура центрального процессора поддается виртуализации, если она поддерживает прямое выполнение команд ВМ на реальной машине. Однако надеяться на чистую эмуляцию средствами центрального процессора команд ВМ не приходится, в связи с неприемлемыми потерями производительности. Поэтому все виртуальные компьютеры идут другим путем – не эмулируя в прямом смысле слова несколько виртуальных машин, а запуская на одном компьютере несколько операционных систем, позволяя МВМ полностью контролировать ЦП.

Следовательно, ключевое требование к архитектуре — наличие механизма перехвата, позволяющего МВМ непосредственно использовать центральный процессор для выполнения ВМ. Располагая таким механизмом, МВМ посредством прямого выполнения может создать для программного обеспечения ВМ иллюзию нормальной физической машины.

Традиционный метод виртуализации памяти состоит в том, что МВМ поддерживает «теневую» копию структуры данных для управления памятью виртуальной машины. Эта структура, называемая «теневой таблицей страниц», позволяет МВМ точно знать, какие страницы физической памяти доступны виртуальной машине.

МВМ следит за изменениями, которые ОС на виртуальной машине вносит в свою таблицу страниц, и соответствующим образом изменяет строки теневой таблицы страниц, указывающей фактическое местоположение страниц в физической памяти. При выполнении кода ВМ аппаратные средства используют теневую таблицу страниц для трансляции адресов памяти, и МВМ всегда может контролировать память, которую использует каждая виртуальная машина.

Подобно подсистемам виртуальной памяти в традиционных ОС, МВМ может подкачивать страницы памяти ВМ с диска, так что выделенная ей память может превышать объем физической памяти. Поскольку ВМ способна расходовать машинную память практически без ограничений, для работы ВМ требуется меньше аппаратных ресурсов. МВМ может динамически контролировать, сколько памяти получает каждая виртуальная машина в соответствии со своими нуждами.

Подсистема виртуальной памяти МВМ постоянно контролирует, сколько памяти выделено ВМ, и должна периодически забирать часть этой памяти, выгружая некоторые страницы ВМ на диск. Однако гостевая ОС на ВМ (GuestOS) «лучше знает», какие страницы следует выгружать. Например, она способна определить, что создавший страницу процесс завершил работу и страница больше не нужна. МВМ, который работает на аппаратном уровне, «не знает» об этом и может продолжать подкачку страницы, расточительно расходуя ресурсы.

В системе с несколькими виртуальными машинами, использующими одно и то же устройство ввода/вывода, МВМ будет нуждаться в эффективном механизме маршрутизации прерываний, сигнализирующих о завершении операции на устройстве, чтобы прерывание было направлено на нужную ВМ. Устройства ввода-вывода должны будут в тесном взаимодействии с МВМ поддерживать изоляцию оборудования от ПО, создавать условия для переноса виртуальных машин и сохранения их состояний в виде контрольных точек. Такие устройства минимизируют накладные расходы на виртуализацию, позволив эффективно использовать виртуальные машины при решении задач с предельной интенсивностью ввода-вывода. Помимо выигрыша в производительности, существенно повысятся безопасность и надежность — благодаря удалению из МВМ сложного кода драйверов устройств.

1.9. Эффекты виртуализации

Экспертиза современных продуктов и недавние исследования раскрывают некоторые интересные возможности развития МВМ и требования, которые они предъявят к технологиям виртуализации.

Администраторы центра данных могут с единой консоли быстро вводить в действие ВМ и управлять тысячами виртуальных машин, выполняющихся на сотнях физических серверов. Вместо того чтобы конфигурировать отдельные компьютеры, администраторы будут создавать по имеющимся шаблонам новые экземпляры виртуальных серверов, и отображать их на физические ресурсы в соответствии с политиками администрирования. Уйдет в прошлое взгляд на компьютер как на средство предоставления конкретных услуг. Администраторы будут рассматривать компьютеры просто как часть пула универсальных аппаратных ресурсов (примером тому может служить виртуальный центр VMware VirtualCenter).

Отображение виртуальных машин на аппаратные ресурсы очень динамично. Возможности миграции работающих ВМ (подобные тем, которые обеспечивает технология VMotion компании Vmware) позволяют ВМ быстро перемещаться между физическими машинами в соответствии с потребностями центра данных. МВМ сможет справляться с такими традиционными проблемами, как отказ оборудования, за счет простого перемещения ВМ с отказавшего компьютера на исправный. Возможность перемещения работающих ВМ облегчит решение аппаратных проблем, таких как планирование профилактического обслуживания, окончание срока действия лизингового договора и модернизация оборудования: администраторы станут устранять эти проблемы без перерывов в работе.

Еще недавно нормой являлась ручная миграция, но сейчас уже распространены инфраструктуры виртуальных машин, которые автоматически выполняет балансировку нагрузки, прогнозируют отказы аппаратных средств и соответствующим образом перемещают ВМ, создают их и уничтожают в соответствии со спросом на конкретные услуги.

Решение проблем на уровне МВМ положительно сказывается на всех программах, выполняющихся на ВМ, независимо от их возраста (унаследованная или новейшая) и поставщиков. Независимость от ОС избавляет от необходимости покупать и обслуживать избыточную инфраструктуру. Например, из нескольких версий ПО службы поддержки или резервного копирования останется лишь одна – та, которая работает на уровне МВМ.

Виртуальные машины сильно изменили отношение к компьютерам. Уже сейчас простые пользователи умеют легко создавать, копировать и совместно использовать ВМ. Модели их применения значительно отличаются от привычных, сложившихся в условиях вычислительной среды с ограниченной доступностью аппаратных средств. А разработчики ПО могут применять такие продукты, как VMware Workstation, чтобы легко установить компьютерную сеть для тестирования или создать собственный набор испытательных машин для каждой цели.

Повышенная мобильность ВМ значительно изменила способы их применения. Такие проекты, как Collective и Internet Suspend/Resume , демонстрируют возможность перемещения всей вычислительной среды пользователя по локальной и территориально-распределенной сети. Доступность высокоемких недорогих сменных носителей, таких как жесткие диски USB, означает, что потребитель может захватить свою вычислительную среду с собой, куда бы он ни направлялся.

Динамический характер компьютерной среды на базе ВМ требует и более динамичной топологии сети. Виртуальные коммутаторы, виртуальные брандмауэры и оверлейные сети становятся неотъемлемой частью будущего, в котором логическая вычислительная среда отделится от своего физического местоположения.

Виртуализация обеспечивает высокий уровень работоспособности и безопасности благодаря нескольким ключевым возможностям.

#### *Локализация неисправностей* – большинство отказов приложений происходят из-за ошибок ПО. Виртуализация обеспечивает логическое разделение виртуальных разделов, поэтому программный сбой в одном разделе никак не влияет на работу приложения в другом разделе. Логическое разделение также позволяет защищаться от внешних атак, что повышает безопасность консолидированных сред.

#### *Гибкая обработка отказов* – виртуальные разделы можно настроить так, чтобы обеспечить автоматическую обработку отказов для одного или нескольких приложений. Благодаря средствам обеспечения высокой степени работоспособности, заложенным сейчас в платформы на базе процессоров Intel® Itanium® 2 и Intel® Xeon™ MP, требуемый уровень услуг часто можно обеспечить, предусмотрев аварийный раздел на той же платформе, где работает основное приложение. Если требуется еще более высокий уровень работоспособности, аварийный раздел можно разместить на отдельной платформе.

#### *Разные уровни безопасности* – для каждой виртуальной машины можно установить разные настройки безопасности. Это позволит IT-организациям обеспечить высокий уровень контроля за конечными пользователями, а также гибкое распределение административных привилегий.

#### МВМ имеют мощный потенциал для реструктуризации существующих программных систем в целях повышения уровня защиты, а также облегчают развитие новых подходов к построению безопасных систем. Сегодняшние ОС не обеспечивают надежной изоляции, оставляя машину почти беззащитной. Перемещение механизмов защиты за пределы ВМ (чтобы они выполнялись параллельно с ОС, но были изолированы от нее) позволяет сохранить их функциональные возможности и повысить устойчивость к нападениям.

#### Размещение средств безопасности за пределами ВМ – привлекательный способ изоляции сети. Доступ к сети предоставляется ВМ после проверки, гарантирующей, что она, с одной стороны, не представляет угрозы, а с другой — неуязвима для нападения. Управление доступом к сети на уровне ВМ превращает виртуальную машину в мощный инструмент борьбы с распространением злонамеренного кода.

#### Мониторы МВМ особенно интересны в плане управления многочисленными группами программ с различными уровнями безопасности. Благодаря отделению ПО от оборудования ВМ обеспечивают максимальную гибкость при поиске компромисса между производительностью, обратной совместимостью и степенью защиты. Изоляция программного комплекса в целом упрощает его защиту. В сегодняшних ОС почти невозможно судить о безопасности отдельного приложения, поскольку процессы плохо изолированы от друг друга. Таким образом, безопасность приложения зависит от безопасности всех остальных приложений на машине.

#### Гибкость управления ресурсами, которую обеспечивают МВМ, может сделать системы более стойкими к нападениям. Возможность быстро тиражировать ВМ и динамически адаптироваться к большим рабочим нагрузкам станет основой мощного инструмента, позволяющего справиться с нарастающими перегрузками из-за внезапного наплыва посетителей на Web-сайте или атаки типа «отказ в обслуживании».

#### Модель распространения программных продуктов на основе ВМ потребует от поставщиков ПО корректировки лицензионных соглашений. Лицензии на эксплуатацию на конкретном процессоре или физической машине не приживутся в новых условиях, в отличие от лицензий на число пользователей или неограниченных корпоративных лицензий. Пользователи и системные администраторы будут отдавать предпочтение операционным средам, которые легко и без особых затрат распространяются в виде виртуальных машин.

#### Возрождение МВМ существенно изменило представления разработчиков программных и аппаратных средств о структурировании сложных компьютерных систем и управлении ими. Кроме того, МВМ обеспечивают обратную совместимость при развертывании инновационных решений в области операционных систем, которые позволяют решать современные задачи, сохраняя предыдущие достижения. Эта их способность станет ключевой при решении грядущих компьютерных проблем.

#### Компании все чаще отказываются от стратегии развития, основанной на закупке отдельных машин и увязке их в сложные программные системы с обилием межкомпонентных связей. Мониторы МВМ дают этим неустойчивым, плохо управляемым системам новую свободу. В ближайшие годы ВМ выйдут далеко за рамки простого предоставления ресурсов и перешагнут пороги машинных залов, чтобы стать фундаментальными строительными блоками для обеспечения мобильности, безопасности и практичности настольных ПК. Несомненно, МВМ станут важной частью вечно меняющегося компьютерного мира.

Виртуализация предоставляет также преимущества для сред разработки и тестирования ПО. Различные этапы цикла создания ПО, включая получение рабочей версии, можно выполнять в разных виртуальных разделах одной и той же платформы. Это поможет повысить степень полезного использования аппаратного обеспечения и упростить управление жизненным циклом. Во многих случаях IT-организации получат возможность тестировать новые и модернизированные решения на имеющихся рабочих платформах, не прерывая производственный процесс. Это не только упрощает миграцию, но также позволяет сократить расходы, устранив необходимость дублирования вычислительной среды.

Освобождая разработчиков и пользователей от ресурсных ограничений и недостатков интерфейса, виртуальные машины снижают уязвимость системы, повышают мобильность программного обеспечения и эксплуатационную гибкость аппаратной платформы.

Компьютерные системы существуют и продолжают развиваться благодаря тому, что разработаны по законам иерархии и имеют хорошо определенные интерфейсы, отделяющие друг от друга уровни абстракции. Использование таких интерфейсов облегчает независимую разработку аппаратных и программных подсистем силами разных групп специалистов. Абстракции скрывают детали реализации нижнего уровня, уменьшая сложность процесса проектирования.

Подсистемы и компоненты, разработанные по спецификациям разных интерфейсов, не способны взаимодействовать друг с другом. Например, приложения, распространяемые в двоичных кодах, привязаны к определенной ISA и зависят от конкретного интерфейса к операционной системе. Несовместимость интерфейсов может стать сдерживающим фактором, особенно в мире компьютерных сетей, в котором свободное перемещение программ столь же необходимо, как и перемещение данных.

Виртуализация позволяет обойти эту несовместимость. Виртуализация системы или компонента (например, процессора, памяти или устройства ввода/вывода) на конкретном уровне абстракции отображает его интерфейс и видимые ресурсы на интерфейс и ресурсы реальной системы. Следовательно, реальная система выступает в роли другой, виртуальной системы или даже нескольких виртуальных систем.

В отличие от абстракции, виртуализация не всегда нацелена на упрощение или сокрытие деталей. Например, при отображении виртуальных дисков на реальный программные средства виртуализации используют абстракцию файла как промежуточный шаг. Операция записи на виртуальный диск преобразуется в операцию записи в файл (и, следовательно, в операцию записи на реальный диск). Отметим, что в данном случае никакого абстрагирования не происходит – уровень детализации интерфейса виртуального диска (адресация секторов и дорожек) ничем не отличается от уровня детализации реального диска.

#### Глава 4. Инструментальное ПО

#### 4.1. Языки программирования

Инструментальное программное обеспечение (ПО) или системы программирования предназначены для автоматизации разработки новых программ на языке программирования. *Язык программирования* можно определить как формальную [знаковую систему](http://ru.wikipedia.org/wiki/%D0%97%D0%BD%D0%B0%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), предназначенную для записи программ, задающих [алгоритм](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) в форме, понятной для исполнителя (например, [компьютера](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80)). Язык программирования определяет набор [лексических](http://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D0%BA%D0%B0), [синтаксических](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [семантических](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) правил, используемых при составлении [компьютерной программы](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0). Он позволяет [программисту](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) точно определить то, на какие события будет реагировать компьютер, как будут храниться и передаваться [данные](http://ru.wikipedia.org/wiki/%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5_(%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0)), а также какие именно действия следует выполнять над этими данными при различных обстоятельствах.

Со времени создания первых программируемых машин человечество придумало уже более восьми с половиной тысяч языков программирования [1 http://ru.wikipedia.org/wiki/]. Каждый год их число пополняется новыми языками. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие становятся известны миллионам людей. Профессиональные программисты иногда применяют в своей работе более десятка разнообразных языков программирования. Современные прикладные и системные программы, в том числе операционные системы и системы программирования, в основном разрабатываются на алгоритмических языках или языках высокого уровня, которые обеспечивают удобство и высокую производительность работы программиста.

История развития языков программирования начинается с *машинных языков*. Программы для первых компьютеров разрабатывались в машинных кодах, а основным носителем информации были перфокарты и перфоленты. Программисты обязаны были знать архитектуру машины досконально. Программы были достаточно простыми, что обусловливалось, во-первых, весьма ограниченными возможностями этих машин, и, во-вторых, большой сложностью разработки и, главное, отладки программ непосредственно на машинном языке.

Вместе с тем такой способ разработки обеспечивал программисту просто неограниченные возможности работы с компьютером. Становилось возможным использование таких хитроумных алгоритмов и способов организации программ, которые не используют (а некоторые и не знают) многие современные программисты. Например, могла применяться такая возможность, как самомодифицирующийся код. Знание двоичного представления команд позволяло иногда не хранить некоторые данные отдельно, а встраивать их в код как команды. И это далеко не полный список приемов, владение хотя бы одним из которых сейчас сразу же продвигает программиста до уровня экстра-класса [М. Плискин Эволюция языков программирования http://schools.keldysh.ru/sch444/MUSEUM/LANR/evol.htm#1.#1].

Первым значительным шагом был переход к языку *ассемблера* (assembly language, или assembler). Не очень заметный, казалось бы, шаг – переход к *символическому кодированию* машинных команд – имел на самом деле огромное значение. Программисту не надо было больше вникать в хитроумные способы кодирования команд на аппаратном уровне. Более того, зачастую одинаковые по сути команды кодировались различным образом в зависимости от своих параметров.

Известный пример из мира современных компьютеров – кодирование инструкции mov в процессорах Intel. Существует несколько совершенно по-разному кодируемых вариантов команды. Выбор того или иного варианта зависит от операндов, хотя суть выполняемой операции неизменна: поместить содержимое (или значение) второго операнда в первый. Появилась также возможность использования макросов и меток, что также упрощало создание, модификацию и отладку программ. Появилось даже некое подобие переносимости — существовала возможность разработки целого семейства машин со сходной системой команд и некоего общего ассемблера для них, при этом не было нужды обеспечивать двоичную совместимость.

Вместе с тем, переход к новому языку таил в себе и некоторые отрицательные (на первый взгляд) стороны. Становилось почти невозможным использование всяческих хитроумных приемов, подобных упомянутым выше. Кроме того, впервые в истории развития программирования появились два представления программы: в исходных текстах и в откомпилированном виде. Сначала, пока ассемблеры только транслировали мнемонические коды в машинные, одно легко переводилось в другое и обратно, но затем, по мере появления таких возможностей, как метки и макросы, *дизассемблирование* (перевод из машинного кода в ассемблер) становилось все более и более трудным делом.

К концу ассемблерной эры возможность автоматической трансляции в обе стороны была утеряна окончательно. В связи с этим было разработано большой количество специальных программ-дизассемблеров, осуществляющих обратное преобразования, однако в большинстве случаев они с трудом могут разделить код и данные. Кроме того, вся логическая информация (имена переменных, меток и т.п.) теряется безвозвратно. В случае же задачи о декомпиляции языков высокого уровня примеры удовлетворительного решения проблемы и вовсе единичны.

В 1954 году в корпорации IBM группой разработчиков во главе с Джоном Бэкусом (John Backus) был создан язык программирования *Fortran*. Значение этого события трудно переоценить. Это первый язык программирования высокого уровня. Впервые программист мог по-настоящему абстрагироваться от особенностей машинной архитектуры. Ключевой идеей, отличающей новый язык от ассемблера, была концепция подпрограмм.

Напомним, что это современные компьютеры поддерживают подпрограммы на аппаратном уровне, предоставляя соответствующие команды и структуры данных (стек) прямо на уровне ассемблера, в 1954 же году это было совершенно не так. Поэтому компиляция Fortran’а была процессом отнюдь не тривиальным. Кроме того, синтаксическая структура языка была достаточно сложна для машинной обработки в первую очередь из-за того, что пробелы как синтаксические единицы вообще не использовались. Это порождало массу возможностей для скрытых ошибок.

Язык Фортран использовался (и используется сейчас) для научных вычислений. Он страдает от отсутствия многих привычных языковых конструкций и атрибутов, компилятор практически никак не проверяет синтаксически правильную программу с точки зрения семантической корректности (соответствие типов и др.). В нем нет поддержки современных способов структурирования кода и данных. Это осознавали и сами разработчики. По признанию самого Бэкуса, перед ними стояла задача скорее разработки компилятора, чем языка. Понимание самостоятельного значения языков программирования пришло позже.

Появление Фортрана было встречено еще большей критикой, чем внедрение ассемблера. Программистов пугало снижение эффективности программ за счет использования промежуточного звена в виде компилятора. И эти опасения имели под собой основания: действительно, хороший программист, скорее всего, при решении какой-либо небольшой задачи вручную напишет код, работающий быстрее, чем код, полученный как результат компиляции. Через некоторое время пришло понимание того, что реализация больших проектов невозможна без применения языков высокого уровня. Мощность вычислительных машин росла, и с тем падением эффективности, которое раньше считалось угрожающим, стало возможным смириться. Преимущества же языков высокого уровня стали настолько очевидными, что побудили разработчиков к созданию новых языков, все более и более совершенных.

Вторым в истории высокоуровневым языком программирования стал *Lisp*. Он использовался и по сей день используется в основном для разрешения сложных задач. Датой рождения Лиспа был 1958 год, известность к нему пришла чуть позже. В 1960 в журнале Communications of the ACM вышла статья Джона Маккарти (автора Лиспа) с подробным описанием нового языка. Он стал отцом не только Лиспа, но и основоположником всего функционального программирования. Язык Lisp — язык для обработки списков. Получил достаточно широкое распространение в системах искусственного интеллекта. Имеет несколько потомков: *Planner* (1967), *Scheme* (1975), *Common Lisp* (1984). Многие его черты были унаследованы современными языками функционального программирования.

В 1960 году в США был создан язык программирования *Cobol*. Он был рассчитан специально для создания коммерческих приложений. На Коболе написаны тысячи прикладных коммерческих систем. Отличительной особенностью языка является возможность эффективной работы с большими массивами данных, что характерно именно для коммерческих приложений. Популярность Кобола столь высока, что даже сейчас, при всех его недостатках (по структуре и замыслу Кобол во многом напоминает Фортран) появляются новые его диалекты и реализации. Так недавно появилась реализация Кобола, совместимая с Microsoft .NET, что потребовало, вероятно, внесения в язык некоторых черт объектно-ориентированного языка.

В 1960 году командой во главе с Петером Науром (Peter Naur) был создан язык программирования *Algol*. Этот язык дал начало целому семейству алголоподобных языков (важнейший представитель – *Pascal*). В 1968 году появилась новая версия языка - Algol 68. Она не нашла столь широкого практического применения, как первая версия, но была весьма популярна в кругах теоретиков. Язык был достаточно интересен, так как обладал многими уникальными на так момент характеристиками.

К середине 60-х годов прошлого века в США резко возросла потребность в обучении программированию не только специалистов в области вычислительной техники, но и широкого круга пользователей. Это было связано с резким увеличением количества компьютеров в бизнесе. Два профессора Дартмутского колледжа – Томас Курт и Джон Кемени для обучения студентов программированию создали язык *Бейсик* (BASIC). Свое название язык получил по первым буквам английских слов “Beginner’s All-purpose Symbolic Instruction Code” – универсальный код символических инструкций для начинающих.

Есть и другой перевод – базовый, основной, что хорошо соответствовало сложившемуся положению дел в программировании для бизнеса. Язык предназначался для обучения [программированию](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и получил широкое распространение в виде различных диалектов, прежде всего, как язык для [домашних микрокомпьютеров](http://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BC%D0%B0%D1%88%D0%BD%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80). Впоследствии большая часть критики этого языка строилась на том, что после Basic'а нормально программировать человек не может, и исправить это уже не удастся. Как бы то ни было, в 1963 язык был создан и получил имя Dartmouth BASIC.

Настоящую популярность этот язык получил в 1975 году. Тогда Microsoft (в то время только два человека – Билл Гейтс и Пол Аллен) написали интерпретатор бейсика для компьютеров Altair 8800, названный Altair BASIC. Язык стремительно разветвился на множество диалектов. Например, Apple II базировался на одной из его версий, а для операционной системы CP/M был написан BASIC-80. Заметим, что второе (или даже третье) дыхание развитию Basic дал опять же Microsoft. Произошло это в начале 90-х годов прошлого столетия, когда был выпущен Visual Basic, уже совсем не похожий на своего предка.

В 1964 году же корпорация IBM создала язык *PL/1*, который был призван заменить Cobol и Fortran в большинстве приложений. Язык обладал исключительным богатством синтаксических конструкций. В нем впервые появилась обработка исключительных ситуаций и поддержка параллелизма. Надо заметить, что синтаксическая структура языка была крайне сложной. Пробелы уже использовались как синтаксические разделители, но ключевые слова не были зарезервированы. В силу таких особенностей разработка компилятора для PL/1 была исключительно сложным делом. Язык так и не стал популярен вне мира IBM. Однако широко использовался в бывшем Советском Союзе и странах социалистического содружества. Причина этого заключается в производстве этими странами ряда программно совместимых моделей компьютеров ЕС ЭВМ, которые практически были скопированы с компьютеров IBM/360.

Создание каждого из вышеупомянутых языков (за исключением, может быть, Algol’а) было вызвано некоторыми практическими требованиями. Эти языки послужили фундаментом для более поздних разработок. Все они представляют одну и ту же парадигму программирования. Следующие языки пошли существенно дальше в своем развитии, в сторону более глубокого абстрагирования.

В 1970 году Никлаус Вирт создал язык программирования *Pascal*. Язык замечателен тем, что это первый широко распространенный язык для структурного программирования (первым, был Алгол, но он не получил столь широкого распространения). Впервые оператор безусловного перехода перестал играть основополагающую роль при управлении порядком выполнения операторов. В этом языке также внедрена строгая проверка типов, что позволило выявлять многие ошибки на этапе компиляции.

Отрицательной чертой языка было отсутствие в нем средств для разбиения программы на модули. Вирт осознавал это и разработал язык *Modula-2* (1978), в котором идея модуля стала одной из ключевых концепций языка. В 1988 году появился язык *Modula-3*, в котором были добавлены объектно-ориентированные черты. Логическим продолжением Pascal и Modula являются язык *Oberon* и *Oberon-2*. Они характеризуются движением в сторону объектно- и компонентно-ориентированности. В этом плане интересно рассмотреть С-подобные языки.

В 1972 году Керниганом и Ритчи был создан язык программирования *C*. Он создавался как язык для разработки операционной системы UNIX. Язык С часто называют «переносимым ассемблером», имея в виду то, что он позволяет работать с данными практически так же эффективно, как на ассемблере, предоставляя при этом структурированные управляющие конструкции и абстракции высокого уровня (структуры и массивы). Именно с этим связана его огромная популярность и поныне. И именно это является его ахиллесовой пятой. Компилятор C очень слабо контролирует типы, поэтому очень легко написать внешне совершенно правильную, но логически ошибочную программу.

В 1986 году Бьярн Страуструп создал первую версию языка *C++*, добавив в язык C объектно-ориентированные черты, взятые из Simula (см. ниже), и исправив некоторые ошибки и неудачные решения языка. C++ продолжает совершенствоваться и в настоящее время, так в 1998 году вышла новая (третья) версия стандарта, содержащая в себе некоторые довольно существенные изменения. Язык стал основой для разработки современных больших и сложных проектов. У него имеются, однако же, и слабые стороны, вытекающие из требований эффективности.

В 1995 году в корпорации Sun Microsystems Кеном Арнольдом и Джеймсом Гослингом был создан язык *Java.* Он наследовал синтаксис C и C++ и был избавлен от некоторых неприятных черт последнего. Отличительной особенностью языка является компиляция в код некоей абстрактной машины, для которой затем пишется эмулятор (Java Virtual Machine) для реальных систем. Кроме того, в Java нет указателей и множественного наследования, что сильно повышает надежность программирования.

В 1998–2001 годах в корпорации Microsoft группой инженеров под руководством [Андерса Хейлсберга](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D0%B9%D0%BB%D1%81%D0%B1%D0%B5%D1%80%D0%B3,_%D0%90%D0%BD%D0%B4%D0%B5%D1%80%D1%81) в компании был создан язык C#. Он в достаточной степени схож с Java (и задумывался как альтернатива последнему), но имеет и отличительные особенности. Язык C# ориентирован, в основном, на разработку многокомпонентных Интернет-приложений. Это основной язык разработки приложений для платформы [Microsoft .NET](http://ru.wikipedia.org/wiki/.NET_Framework). [Компилятор](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) с C# входит в стандартную установку самой .NET, поэтому программы на нём можно создавать и компилировать даже без инструментальных средств, вроде [Visual Studio](http://ru.wikipedia.org/wiki/Microsoft_Visual_Studio).

В 1983 году под эгидой Министерства Обороны США был создан язык *Ada*. Язык замечателен тем, что очень много ошибок может быть выявлено на этапе компиляции. Кроме того, поддерживаются многие аспекты программирования, которые часто отдаются на откуп операционной системе (параллелизм, обработка исключений). В 1995 году был принят стандарт языка Ada 95, который развивает предыдущую версию, добавляя в нее объектную ориентированность и исправляя некоторые неточности. Оба этих языка не получили широкого распространения вне военных и прочих крупномасштабных проектов (авиация, железнодорожные перевозки). Основной причиной является сложность освоения языка и достаточно громоздкий синтаксис (значительно более громоздкий, чем Pascal).

Все вышеперечисленные языки являются языками общего назначения в том смысле, что они не ориентированы и не оптимизированы под использование каких-либо специфических структур данных или на применение в каких-либо специфических областях. Было разработано большое количество языков, ориентированных на достаточно специфические применения. Ниже приведен краткий обзор таких языков.

В 1957 году была предпринята попытка создания языка для описания математической обработки данных. Язык был назван *APL* (Application Programming Language). Его отличительной особенностью было использование математических символов (что затрудняло применение на текстовых терминалах; появление графических интерфейсов сняло эту проблему) и очень мощный синтаксис, который позволял производить множество нетривиальных операций прямо над сложными объектами, не прибегая к разбиению их на компоненты. Широкому применению помешало, как уже отмечалось, использование нестандартных символов как элементов синтаксиса.

В 1962 году появился язык *Snobol* (а в 1974 – его преемник *Icon*), предназначенный для обработки строк. Синтаксис Icon напоминает С и Pascal одновременно. Отличие заключается в наличии мощных встроенных функций работы со строками и связанная с этими функциями особая семантика. Современным аналогом Icon и Snobol является *Perl –*язык обработки строк и текстов, в который добавлены некоторые объектно-ориентированные возможности. Считается очень практичным языком, однако ему недостает элегантности.

В 1969 году был создан язык *SETL* – язык для описания операций над множествами. Основной структурой данных в языке является множество, а операции аналогичны математическим операциям над множествами. Язык полезен при написании программ, имеющих дело со сложными абстрактными объектами.

В последнее время в связи развитием Интернет-технологий, широким распространением высокопроизводительных компьютеров и рядом других факторов получили распространение так называемые *скриптовые языки*. Эта языки первоначально ориентировались на использование в качестве внутренних управляющих языков во всякого рода сложных системах. Многие из них, однако же, вышли за пределы сферы своего изначального применения и используются ныне в совсем иных областях. Характерными особенностями данных языков являются, во-первых, их интерпретируемость (компиляция либо невозможна, либо нежелательна), во-вторых, простота синтаксиса, а в-третьих, легкая расширяемость. Таким образом, они идеально подходят для использования в часто изменяемых программах, очень небольших программах или в случаях, когда для выполнения операторов языка затрачивается время, несопоставимое со временем их разбора. Было создано достаточно большое количество таких языков, перечислим лишь основные и наиболее часто используемые.

Язык *JavaScript* был создан в компании Netscape Communications в качестве языка для описания сложного поведения веб-страниц. Первоначально язык назывался *LiveScript*, причиной смены названия послужили маркетинговые соображения. Интерпретируется браузером во время отображения веб-страницы. По синтаксису похож на Java и (отдаленно) на C/C++. Язык имеет возможность использовать встроенную в браузер объектную функциональность, однако подлинно объектно-ориентированным языком не является.

Другой скриптовый язык *VBScript* был создан в корпорации Microsoft во многом в качестве альтернативы JavaScript. Имеет подобную область применения. Синтаксически похож на язык Visual Basic (является усеченной версией последнего). Так же, как и JacaScript, исполняется браузером при отображении веб-страниц и имеет ту же степень объектной ориентированности.

Язык Perl, нашедший применение для динамической генерации веб-страниц на веб-серверах, создавался в помощь системному администратору операционной системы Unix для обработки различного рода текстов и выделения нужной информации. Развился до мощного средства работы с текстами. Является интерпретируемым языком и реализован практически на всех существующих платформах. Интерпретируемый объектно-ориентированный язык программирования *Python* по структуре и области применения близок к Perl, однако менее распространен и более строг и логичен. Имеются реализации для большинства существующих платформ.

Интересно рассмотрение группы ранних объектно-ориентированных языков. Объектно-ориентированный подход, пришедший на смену структурному, впервые появился отнюдь не в C++, как полагают некоторые. Существует целая череда чистых объектно-ориентированных языков, без сведений о которых наш обзор был бы неполным. Первым объектно-ориентрованным языком был язык *Simula* (1967). Этот язык был предназначен для моделирования различных объектов и процессов, и объектно-ориентированные черты появились в нем именно для описания свойств модельных объектов.

Популярность объектно-ориентированному программированию принес язык *Smalltalk*, созданный в 1972 году. Язык предназначался для проектирования сложных графических интерфейсов и был первым по-настоящему объектно-ориентированным языком. В нем классы и объекты – это единственные конструкции программирования. Недостатком Smalltalk являются большие требования к памяти и низкая производительность полученных программ. Это связано с не очень удачной реализацией объектно-ориентированных особенностей. Популярность языков C++ и Ada 95 связана именно с тем, что объектная ориентированность реализована без существенного снижения производительности.

Существует еще язык с очень хорошей реализацией объектной ориентированности, не являющийся надстройкой ни над каким другим языком. Это язык *Eiffel* (1986). Являясь чистым языком объектно-ориентированного программирования, он, кроме того, повышает надежность программы путем использования «контрольных утверждений».

Большинство компьютерных архитектур и языков программирования ориентированы на последовательное выполнение операторов программы. В настоящее время существуют программно-аппаратные комплексы, позволяющие организовать параллельное выполнение различных частей одного и того же вычислительного процесса. Для программирования таких систем необходима специальная поддержка со стороны средств программирования, в частности, языков программирования. Некоторые языки общего назначения содержат в себе элементы поддержки параллелизма, однако программирование истинно параллельных систем требует подчас специальных приемов [М. Плискин Эволюция языков программирования http://schools.keldysh.ru/sch444/MUSEUM/LANR/evol.htm#1.#1]. .

Язык *Оccam* был создан в 1982 году и предназначен для программирования транспьютеров – многопроцессорных систем распределенной обработки данных. Он описывает взаимодействие параллельных процессов в виде каналов – способов передачи информации от одного процесса к другому. Отметим особенность синтаксиса языка Occam – в нем последовательный и параллельный порядки выполнение операторов равноправны, и их необходимо явно указывать ключевыми словами PAR и SEQ.

В 1985 году была предложена модель параллельных вычислений *Linda*. Основной ее задачей является организация взаимодействия между параллельно выполняющимися процессами. Это достигается за счет использования глобальной кортежной области (tuple space). Процесс может поместить туда кортеж с данными (то есть совокупность нескольких, возможно разнородных, данных), а другой процесс может ожидать появления в кортежной области некоторого кортежа и, после его появления, прочитать кортеж с возможным последующим его удалением.

Заметим, что процесс может, например, поместить кортеж в область и завершиться, а другой процесс может через некоторое время воспользоваться этим кортежем. Таким образом, обеспечивается возможность асинхронного взаимодействия. Очевидно, что при помощи такой модели можно эмулировать и синхронное взаимодействие. Linda – это модель параллельных вычислений, она может быть добавлена в любой язык программирования. Существуют достаточно эффективные реализации Linda, обходящие проблему существования глобальной кортежной области с потенциально неограниченным объемом памяти.

Все языки, о которых шла речь ранее, имеют одно общее свойство: они императивны. Это означает, что программы на них, в конечном итоге, представляют собой пошаговое описание решения той или иной задачи. Можно попытаться описывать лишь постановку проблемы, а решать задачу поручить компилятору. Существует два основных подхода, развивающие эту идею: функциональное и логическое программирование.

Основная идея, лежащая в основе функционального программирования, - это представление программы в виде математических функций (т.е. функций, значение которых определяется лишь их аргументами, а не контекстом выполнения). Оператор присваивания в таких языках не используется (или, как минимум, его использование не поощряется). Императивные возможности, как правило, имеются, но их применение обставлено серьезными ограничениями. Существуют языки с ленивой и с энергичной семантикой. Различие заключается, грубо говоря, в том, что в языках с энергичной семантикой вычисления производятся в том же месте, где они описаны, а в случае ленивой семантики вычисление производится только тогда, когда оно действительно необходимо. Первые языки имеют более эффективную реализацию, в то время как вторые – лучшую семантику.

Из языков с энергичной семантикой упомянем *ML* и два его современных диалекта – *Standard ML* (SML) и *CaML*. Последний имеет объектно-ориентированного потомка – Objective CaML (O’CaML). Среди языков с ленивой семантикой наиболее распространены два: *Haskell* и его более простой диалект *Clean*. Интересен язык функционального программирования *F#*. Он является языком мультипарадигменного программирования. На нем можно функциональный, императивный и объектно-ориентированный код. Это позволяет быть более прагматичным, вместо того чтобы пытаться загнать любую задачу, стоящую перед разработчиком, в прокрустово ложе классов и интерфейсов. Язык F# включен в стандартный набор *Visual Studio 2010*, хотя присутствует и сейчас, в виде плагина для VS2008.

F# использует вывод типов, что приводит к более лаконичным программам. Программист тратит массу времени, добавляя сигнатуры типов, точки с запятыми, фигурные скобки в своем любимом языке программирования. Вся эта информация предназначена большей частью для упрощения создания компилятора. Почему бы не писать тот код, который хочет программист, и предоставить компилятору вычислить остальное? Лаконичность языка F# его главная сильная сторона. F# – это .NET-язык программирования.

Он компилируется в IL в том же рабочем цикле, что и C# с VB, и будет работать с уже существующим .NET-кодом. У него есть интерактивная среда программирования, называемая F# Interactive Window. Программировать на нем увлекательно. Этот язык рушит многие барьеры, связанные с программированием и позволяет сконцентрироваться на написании кода, который нужен разработчику.

Важно отметить, что F# поддерживает почти все возможности, которые есть у C#. Поэтому его можно использовать, не опасаясь принципа «все или ничего». Не нужно выбрасывать существующий код и переводить все на F#. Вообще, предполагается, что код на F# будет главным образом использоваться как библиотеки классов, интегрированные в большой программный продукт.

Программы на языках логического программирования выражены как формулы математической логики, а компилятор пытается получить следствия из них. Родоначальником большинства языков логического программирования является язык *Prolog* (1971). У него есть ряд потомков – *Parlog* (1983, ориентирован на параллельные вычисления), *Delta Prolog* и др.

Технология программирования во многом определяется языком программирования, на котором пишутся программы. В языке могут быть заложены средства, влияющие на технологичность и архитектуру разрабатываемой системы (например, объектно-ориентированность, модульность и т.п.). О распространенности языков можно судить по рейтингу, ежемесячно составляемой фирмой TIOBE [ 

*Л.Н. Чернышов* СРЕДЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ: ИСТОРИЯ И ПЕРСПЕКТИВЫ ].

В таблице приводится первая десятка языков с наибольшими рейтингами на март 2009 и указанием тенденций в их местах по сравнению с тем же месяцем прошлого года. Рейтинг составляется по определенной методике, основанной в основном на упоминаниях в Интернете. Как видно, ведущие места занимают C-подобный языки. Язык Java прочно удерживает первую позицию в течение последних лет. Интересно, что эта десятка практически постоянна (см. рейтинги 2005 года), но доля первых трех языков снижается (с 54 до 45 % за 4 года). Это, по-видимому, связано с развитием интернет-технологий с ее скриптовыми языками Python, Perl, JavaScript и др.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Позиция | Изменения по сравнению с прошлым годом | Язык программирования | Рейтинг Март 2009 | Рейтинг октябрь 2005 |
| 1 | 0 | [Java](http://www.tiobe.com/content/paperinfo/tpci/Java.html) | 19.9% | 21.9% |
| 2 | 0 | [C](http://www.tiobe.com/content/paperinfo/tpci/C.html) | 15.9% | 18.8% |
| 3 | +2 | [C++](http://www.tiobe.com/content/paperinfo/tpci/C__.html) | 10.4% | 11.8% |
| 4 | 0 | [PHP](http://www.tiobe.com/content/paperinfo/tpci/PHP.html) | 9.5% | 9.7% |
| 5 | -2 | [(Visual) Basic](http://www.tiobe.com/content/paperinfo/tpci/(Visual)_Basic.html) | 8.3% | 6.9% |
| 6 | +1 | [Python](http://www.tiobe.com/content/paperinfo/tpci/Python.html) | 5.2% | 2.9% |
| 7 | +1 | [C#](http://www.tiobe.com/content/paperinfo/tpci/C_.html) | 4.3% | 3.5% |
| 8 | +2 | [JavaScript](http://www.tiobe.com/content/paperinfo/tpci/JavaScript.html) | 3.6% | 2.0% |
| 9 | -3 | [Perl](http://www.tiobe.com/content/paperinfo/tpci/Perl.html) | 3.4% | 7.4% |
| 10 | -1 | [Delphi](http://www.tiobe.com/content/paperinfo/tpci/Delphi.html) | 2.7% | 1.5% |

В заключение раздела можно выделить некоторую общую тенденцию в развитии языков программирования. Языки развиваются в сторону все большей и большей абстракции. И это сопровождается падением эффективности. Вопрос: а стоит ли этого абстракция? Ответ: стоит, так как повышение уровня абстракции влечет за собой повышение уровня надежности программирования. С низкой эффективностью можно бороться путем создания более быстрых компьютеров. Если требования к памяти слишком высоки, можно увеличить ее объем. Это требует времени и средств, но это решаемо. А вот с ошибками в программах можно бороться только одним способом: их надо исправлять. Еще лучше не совершать. А еще лучше максимально затруднить их совершение. И именно на это направлены все исследования в области языков программирования.

**4.2. Основы компиляции**

Программы, написанные на языках программирования высокого уровня перед выполнением на ЭВМ должны транслироваться в эквивалентные программы, написанные на машинном коде. *Транслятор* - это программа, которая переводит программу на исходном (входном) языке в эквивалентную ей программу на результирующем (выходном) языке. Если исходный язык является языком высокого уровня, например таким, как Паскаль, С++, и если объектный язык - автокод, то такой транслятор называется компилятором.

Достоинство компилятора заключается в том, что программа компилируется один раз и при каждом выполнении не требуется дополнительных преобразований. Соответственно, не требуется наличие компилятора на целевой машине, для которой компилируется программа. Недостаток: отдельный этап компиляции замедляет написание и отладку и затрудняет исполнение небольших, несложных или разовых программ. В том случае, если исходный язык является [языком ассемблера](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0) (низкоуровневым языком, близким к машинному языку), то компилятор такого языка называется ассемблером.

Другой метод реализации программ, написанных на языке высокого уровня – интерпретация [Р. У. Себеста. Основные концепции языков программирования/ Пер. с англ. – М.: [Вильямс](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)), 2001]. Интерпретатор программно моделирует машину, цикл выборки-исполнения которой работает с командами на языках высокого уровня, а не с машинными командами. Такое программное моделирование создаёт [виртуальную машину](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0), реализующую язык. Этот подход называется чистой интерпретацией. Чистая интерпретация применяется, как правило, для языков с простой структурой (например, [АПЛ](http://ru.wikipedia.org/wiki/%D0%90%D0%9F%D0%9B_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) или [Лисп](http://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF)). [Интерпретаторы командной строки](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%BE%D0%B9_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8) обрабатывают команды в [скриптах](http://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D1%80%D0%B8%D0%BF%D1%82) в [UNIX](http://ru.wikipedia.org/wiki/UNIX) или в [пакетных файлах](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) (.bat) в [MS-DOS](http://ru.wikipedia.org/wiki/MS-DOS) также, как правило, в режиме чистой интерпретации.

Достоинство чистого интерпретатора: отсутствие промежуточных действий для трансляции упрощает реализацию интерпретатора и делает его удобнее в использовании, в том числе в диалоговом режиме. Недостаток – интерпретатор должен быть в наличии на целевой машине, где должна исполняться программа. А свойство чистого интерпретатора, что ошибки в интерпретируемой программе обнаруживаются только при попытке выполнения команды (или строки) с ошибкой, можно признать как недостатком, так и достоинством.

Существуют компромиссные между компиляцией и чистой интерпретацией варианты реализации языков программирования, когда интерпретатор перед исполнением программы транслирует её на промежуточный язык (например, в [байт-код](http://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) или [p-код](http://ru.wikipedia.org/wiki/P-%D0%BA%D0%BE%D0%B4)), более удобный для интерпретации (т.е. речь идёт об интерпретаторе со встроенным транслятором). Такой метод называется смешанной реализацией. Примером смешанной реализации языка может служить [Perl](http://ru.wikipedia.org/wiki/Perl). Этот подход сочетает как достоинства компилятора и интерпретатора (бо́льшая скорость исполнения и удобство использования), так и недостатки (для трансляции и хранения программы на промежуточном языке требуются дополнительные ресурсы; для исполнения программы на целевой машине должен быть представлен интерпретатор). Так же, как и в случае компилятора, смешанная реализация требует, чтобы перед исполнением исходный код не содержал ошибок (лексических, синтаксических и семантических).

Для компиляции компилятор должен выполнить анализ исходной программы, а затем синтез объектной программы. Сначала исходная программа разлагается на ее составные части; затем из них строятся части эквивалентной объектной программы. Для этого на этапе анализа компилятор строит несколько таблиц (рис. 1), которые используются затем как при анализе, так и при синтезе.

Рис. 1

При анализе программы из описаний, заголовков процедур, заголовков циклов и т.д. извлекается информация и сохраняется для последующего использования. Эта информация обнаруживается в отдельных точках программы и организуется так, чтобы к ней можно было обратиться из любой части компилятора. Например, при каждом использовании идентификатора необходимо знать, как был описан этот идентификатор и как он использовался в других местах программы. Что конкретно следует хранить, зависит, конечно, от исходного языка, объектного языка и сложности компилятора. Но в каждом компиляторе в той или иной форме используется таблица символов (иногда ее называют списком идентификаторов или таблицей имен). Это таблица идентификаторов, встречающихся в исходной программе, вместе с их атрибутами. К атрибутам относятся тип идентификатора, его адрес в объектной программе и любая другая информация о нем, которая может понадобиться при генерации объектной программы.

Лексический анализатор (сканер) - самая простая часть компилятора. Сканер просматривает литеры исходной программы слева направо и строит символы программы – целые числа, идентификаторы, служебные слова и т. д. (Символы передаются затем на обработку фактическому анализатору.) На этой стадии может быть исключен комментарий. Сканер также может заносить идентификаторы в таблицу символов и выполнять другую простую работу, которая фактически не требует анализа исходной программы. Он может выполнить большую часть работы по макрогенерации в тех случаях, когда требуется только текстовая подстановка.

Обычно сканер передает символы анализатору во внутренней форме. Каждый разделитель (служебное слово, знак операции или знак пунктуации) будет представлен целым числом. Идентификаторы или константы можно представить парой чисел. Первое число, отличное от любого целого числа, использующегося для представления разделителя, характеризует сам "идентификатор" или "константу"; второе число является адресом или индексом идентификатора или константы в некоторой таблице. Это позволяет в остальных частях компилятора работать эффективно, оперируя с символами фиксированной длины, а не с цепочками литер переменной длины.

Синтаксический и семантический анализаторы выполняют сложную работу по расчленению исходной программы на составные части, формированию ее внутреннего представления и занесению информации в таблицу символов и другие таблицы. При этом также выполняется полный синтаксический и семантический контроль программы. Синтаксис – способ соединения слов (и их форм) в словосочетания и предложения. Семантика – это значения единиц языка. Обычный анализатор представляет собой синтаксически управляемую программу.

В действительности стремятся отделить синтаксис от семантики настолько, насколько это возможно. Когда синтаксический анализатор (распознаватель) узнает конструкцию исходного языка, он вызывает соответствующую семантическую процедуру, или семантическую программу, которая контролирует данную конструкцию с точки зрения семантики и затем запоминает информацию о ней в таблице символов или во внутреннем представлении программы. Например, когда распознается описание переменных, семантическая программа проверяет идентификаторы, указанные в этом описании, чтобы убедиться в том, что они не были описаны дважды, и заносит их вместе с атрибутами в таблицу символов.

Внутреннее представление исходной программы в значительной степени зависит от его дальнейшего использования. Это может быть дерево, отражающее синтаксис исходной программы. Это может быть исходная программа, в так называемой польской записи, список тетрад и т.д.

Перед генерацией команд обычно необходимо некоторым образом обработать и изменить внутреннюю программу. Кроме того, должна быть распределена память под переменные готовой программы. Одним, из важных моментов на этом этапе является оптимизация программы с целью уменьшения времени ее работы. По существу, на этом этапе происходит перевод внутреннего представления исходной программы на автокод или на машинный язык. Это наиболее хлопотная и кропотливая часть компилятора, хотя и наиболее понятная. В интерпретаторе эта часть компилятора заменяется программой, которая фактически выполняет (или интерпретирует) внутреннее представление исходной программы. Причем само внутреннее представление в этом случае мало чем отличается от того, которое получается при компиляции.

Естественно, возникает вопрос: в чем заключаются главные трудности реализации компилятора? Сканер весьма прост и хорошо изучен. Синтаксические анализаторы, если речь идет о простых формальных языках, также довольно хорошо изучены. В действительности эту часть можно в значительной степени автоматизировать. С тех пор, как синтаксис был формализован, большинство исследований по созданию компиляторов касалось именно синтаксиса, а не семантики. Наиболее трудными и запутанными частями компилятора являются семантический анализ, программы подготовки генерации и программы генерации команд. Эти три части взаимозависимы, должны в значительной степени разрабатываться совместно, и могут коренным образом измениться при переходе с одного объектного языка на другой или с одной машины на другую.

Более детальное представление о процессе компиляции можно получить в специальной литературе [ ].

**4.3. Понятие системы программирования**

Всякий компилятор является составной частью системного программного обеспечения. Основное назначение компиляторов – служить для разработки новых прикладных и системных программ с помощью языков высокого уровня. Любая программа, как системная, так и прикладная, проходит этапы жизненного цикла, начиная от проектирования и вплоть до внедрения и сопровождения. А компиляторы это средства, служащие для создания программного обеспечения на этапах кодирования, тестирования и отладки.  
 Однако сам по себе компилятор не решает полностью всех задач, связанных с разработкой новой программы. Средств только лишь компилятора недостаточно для того, чтобы обеспечить прохождение программой указанных этапов жизненного цикла. Поэтому компиляторы это программное обеспечение, которое функционирует в тесном взаимодействии с другими техническими средствами, применяемыми на данных этапах.

Основные технические средства, используемые в комплексе с компиляторами, включают в себя следующие программные модули:

* текстовые редакторы, служащие для создания текстов исходных программ;
* компоновщики, позволяющие объединять несколько объектных модулей, порождаемых компилятором, в единое целое;
* библиотеки прикладных программ, содержащие в себе наиболее часто используемые функции и подпрограммы в виде готовых объектных модулей;
* загрузчики, обеспечивающие подготовку готовой программы к выполнению;
* отладчики, выполняющие программу в заданном режиме с целью поиска, обнаружения и локализации ошибок;
* другие программные средства, служащие для разработки программ и их компонентов.

Все эти средства разработки функционируют не отдельно, каждое само по себе, а в тесном взаимодействии друг с другом. Данные, полученные в одном модуле, поступают на вход другого и наоборот. В современных средствах разработки интеграция модулей столь высока, что пользователь часто и не представляет, что он работает с несколькими программными средствами – для него вся система разработки представляет собой единое целое. Весь этот комплекс программно-технических средств составляет новое понятие, которое называется системой программирования.

Системы программирования в современном мире доминируют на рынке средств разработки. Практически все фирмы-разработчики компиляторов поставляют свои продукты в составе соответствующей системы программирования в комплексе всех прочих технических средств. Отдельные компиляторы являются редкостью и, как правило, служат только узкоспециализированным целям.

Тенденция такова, что все развитие систем программирования идет в направлении неуклонного повышения их дружественности и сервисных возможностей. Это связано с тем, что на рынке в первую очередь лидируют те системы программирования, которые позволяют существенно снизить трудозатраты, необходимые для создания программного обеспечения на этапах жизненного цикла, связанных с кодированием, тестированием и отладкой программ. Показатель снижения трудозатрат в настоящее время считается более существенным, чем показатели, определяющие эффективность результирующей программы, построенной с помощью системы программирования.

В качестве основных тенденций в развитии современных систем программирования следует указать внедрение в них средств разработки на основе так называемых языков четвертого поколения 4GL (four generation languages), а также поддержку систем быстрой разработки программного обеспечения RAD (rapid application development).

Языки четвертого поколения 4GL представляют собой широкий набор средств, ориентированных на проектирование и разработку программного обеспечения. Они строятся на основе оперирования не синтаксическими структурами языка и описаниями элементов, а представляющими их графическими образами. На таком уровне проектировать и разрабатывать прикладное программное обеспечение может пользователь, не являющийся квалифицированным программистом, зато имеющий представление о предметной области, на работу в которой ориентирована прикладная программа.

Описание программы, построенное на основе языков 4GL, транслируется затем в исходный текст и файл описания ресурсов интерфейса, представляющие собой обычный тексты на соответствующем входном языке высокого уровня. С этим текстом уже может работать профессиональный программист-разработчик, он может корректировать и дополнять его необходимыми функциями. Такой подход позволяет разделить работу проектировщика, ответственного за общую концепцию всего проекта создаваемой системы, дизайнера, отвечающего за внешний вид интерфейса пользователя, и профессионального программиста, отвечающего непосредственно за создание исходного кода создаваемого программного обеспечения.

В целом языки четвертого поколения решают уже более широкий класс задач, чем традиционные системы программирования. Они составляют часть средств автоматизированного проектирования и разработки программного обеспечения, поддерживающих все этапы жизненного цикла CASE-систем. Их рассмотрение выходит за рамки данного учебного пособия.

**4.4. Возникновение систем программирования**

Первоначально компиляторы представляли собой обособленные программные модули, решающие исключительно задачу перевода исходного текста программы на входном языке в язык машинных кодов. Компиляторы разрабатывались вне связи с другими техническими средствами, с которыми им приходилось взаимодействовать. В задачу разработчика программы входило обеспечить взаимосвязь всех используемых технических средств:

* подготовить тексты исходной программы на входном языке компилятора;
* подать входные данные в виде текста исходной программы на вход компилятора;
* получить от компилятора результаты его работы в виде набора объектных файлов;
* подать весь набор полученных объектных файлов вместе с необходимыми библиотеками подпрограмм на вход компоновщику;
* получить от компоновщика единый файл программы (исполняемый файл) и подготовить его к выполнению с помощью загрузчика;
* поставить программу на выполнение, при необходимости использовать отладчик для проверки правильности выполнения программы.

Все эти действия выполнялись с помощью последовательности команд, инициировавших запуск соответствующих программных модулей с передачей им всех необходимых параметров. Параметры передавались каждому модулю в командной строке и представляли собой набор имен файлов и настроек, реализованных в виде специальных ключей. Пользователи могли выполнять эти команды последовательно вручную, а с развитием средств командных процессоров ОС они стали объединять их в командные файлы.

Со временем разработчики компиляторов постарались облегчить труд пользователей, предоставив им все необходимое множество программных модулей в составе одной поставки компилятора. Теперь компиляторы поставлялись уже вкупе со всеми необходимыми сопровождающими техническими средствами. Кроме того, были унифицированы форматы объектных файлов и файлов библиотек подпрограмм. Теперь разработчики, имея компилятор от одного производителя, могли в принципе пользоваться библиотеками и объектными файлами, полученными от другого производителя компиляторов или от другой команды разработчиков.

Поскольку процессу компиляции всегда соответствует типичная последовательность команд, то разрабатывать для этой цели командный файл в операционной системе неэффективно и неудобно. Для написания подобных командных файлов пользователям компиляторов был предложен специальный командный язык, который обрабатывался интерпретатором команд компиляции – программой make (от английского глагола – строить, делать). Командный язык язык Makefile позволял в достаточно гибкой и удобной форме описать весь процесс создания программы от порождения исходных текстов до подготовки ее к выполнению [источник 55, 66 из Молчанова]. Makefile позволял в достаточно гибкой и удобной форме описать весь процесс создания программы от порождения исходных текстов до подготовки ее к выполнению.

Появление Makefile можно считать первым шагом по созданию систем программирования. Язык Makefile стал стандартным средством, единым для компиляторов всех разработчиков. Это было удобное, но достаточно сложное техническое средство, требующее от разработчика высокой степени подготовки и профессиональных знаний, поскольку сам командный язык Makefile был по сложности сравним с простым языком программирования.

Такая структура средств разработки существовала достаточно долгое время, а в некоторых случаях она используется и по сей день (особенно при создании системных программ). Ее широкое распространение было связано с тем, что сама по себе вся эта структура средств разработки была очень удобной при пакетном выполнении программ на компьютере, что способствовало ее повсеместному применению в эпоху mainframe с операционными системами типа Unix.

**4.5. Интегрированные среды разработки**

Командная строка – эффективное, но не всегда удобное средство управления компиляцией, хотя с этим могут и поспорить программисты, работающие под операционной системой Unix. Дело в том. Что фактически для каждого сложного проекта разработчикам приходится еще одну дополнительную программу, описывающую на языке Makefile, как следует собирать (компилировать) этот проект. А программирование всегда есть потенциальный источник ошибок. Поэтому развитие систем программирования на этом не завершилось.

Следующим шагом в развитии средств разработки стало появление так называемой интегрированной среды разработки. Интегрированная среда объединила в себе возможности текстовых редакторов исходных текстов программ и командный язык компиляции. Пользователь (разработчик исходной программы) теперь не должен был выполнять всю последовательность действий от порождения исходного кода до его выполнения, от него также не требовалось описывать этот процесс с помощью системы команд в Makefile. Теперь ему было достаточно только указать в удобной интерфейсной форме состав необходимых для создания программы исходных модулей и библиотек. Ключи, необходимые компилятору и другим техническим средствам, также задавались в виде интерфейсных форм настройки.

После этого интегрированная среда разработки сама автоматически готовила всю необходимую последовательность команд Makefile, выполняла их, получала результат и сообщала о возникших ошибках при их наличии. Причем сам текст исходных модулей пользователь мог изменить здесь же, не прерывая работу с интегрированной средой, чтобы потом при необходимости просто повторить весь процесс компиляции. Кроме удобства работы, объединение текстовых редакторов, компиляторов и компоновщиков в единую среду дало системам программирования целый ряд достоинств.

Создание интегрированных сред разработки стало возможным благодаря бурному развитию персональных компьютеров и появлению развитых средств интерфейса пользователя (сначала текстовых, а потом и графических). Их появление на рынке определило дальнейшие развитие такого рода технических средств. Пожалуй, первой удачной средой такого рода можно признать интегрированную среду программирования Turbo Pascal на основе языка Pascal производства фирмы Borland [ ]. Ее широкая популярность определила тот факт, что со временем все разработчики компиляторов обратились к созданию интегрированных средств разработки для своих продуктов.

Развитие интегрированных сред несколько снизило требования к профессиональным навыкам разработчиков исходных программ. Теперь в простейшем случае от разработчика требовалось только знание исходного языка (его синтаксиса и семантики). При создании прикладной программы ее разработчик мог в простейшем случае даже не разбираться в архитектуре целевой вычислительной системы.

Дальнейшее развитие средств разработки также тесно связано с повсеместным распространением развитых средств графического интерфейса пользователя. Такой интерфейс стал неотъемлемой составной частью многих современных ОС и так называемых графических оболочек. Со временем он стал стандартом де-факто практически во всех современных прикладных программах. Это не могло не сказаться на требованиях, предъявляемых к средствам разработки программного обеспечения. В их состав были сначала включены соответствующие библиотеки, обеспечивающие поддержку развитого графического интерфейса пользователя и взаимодействие с функциями API (Application Program Interface, прикладной программный интерфейс) операционных систем [ ]. А затем для работы с ними потребовались дополнительные средства, обеспечивающие разработку внешнего вида интерфейсных модулей. Такая работа была уже более характерна для дизайнера, чем для программиста.

Для описания графических элементов программ потребовались соответствующие языки. На их основе сложилось понятие ресурсов (resources) прикладных программ. Ресурсами прикладной программы будем называть множество данных, обеспечивающих внешний вид интерфейса пользователя этой программы, и не связанных напрямую с логикой выполнения программы. Характерными примерами ресурсов являются: тексты сообщений, выдаваемых программой; цветовая гамма элементов интерфейса; надписи на таких элементах, как кнопки и заголовки окон и т.п.

Для формирования структуры ресурсов в свою очередь потребовались редакторы ресурсов, а затем и компиляторы ресурсов, обрабатывающие результат их работы. Ресурсы, полученные с выхода компиляторов ресурсов, стали обрабатываться компоновщиками и загрузчиками. Весь этот комплекс программно-технических средств составляет новое понятие, называемое системой программирования. Используется и другое наименование системы программирования – среда разработки программного обеспечения – совокупность программных средств, используемая [программистами](http://ru.wikipedia.org/wiki/Программист) для разработки [программного обеспечения](http://ru.wikipedia.org/wiki/Программное_обеспечение).

Простая среда разработки включает в себя [редактор](http://ru.wikipedia.org/wiki/Текстовый_редактор) текста, [компилятор](http://ru.wikipedia.org/wiki/Компилятор) и/или [интерпретатор](http://ru.wikipedia.org/wiki/Интерпретатор), средства автоматизации сборки и [отладчик](http://ru.wikipedia.org/wiki/Отладчик). Когда эти компоненты собраны в единый программный комплекс, говорят об интегрированной среде разработки (Integrated development environment- IDE). Такая среда представлена одной программой, не выходя из которой можно производить весь цикл разработки. В состав комплекса кроме перечисленных выше компонент могут входить средства управления проектами, [система управления версиями](http://ru.wikipedia.org/wiki/Система_управления_версиями), разнообразные инструменты для упрощения разработки [интерфейса пользователя](http://ru.wikipedia.org/wiki/Графический_интерфейс_пользователя), стандартные заготовки («мастера»), упрощающие разработку стандартных задач, и др.

Обычно среда разработки включает в себя [текстовый редактор](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%80%D0%B5%D0%B4%D0%B0%D0%BA%D1%82%D0%BE%D1%80), [компилятор](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) и/или [интерпретатор](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80), средства автоматизации сборки и [отладчик](http://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BB%D0%B0%D0%B4%D1%87%D0%B8%D0%BA). Иногда также содержит средства для интеграции с [системами управления версиями](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F%D0%BC%D0%B8) и разнообразные инструменты для упрощения конструирования [графического интерфейса пользователя](http://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8F). Многие современные среды разработки также включают браузер классов, инспектор объектов и диаграмму [иерархии классов](http://ru.wikipedia.org/wiki/%D0%98%D0%B5%D1%80%D0%B0%D1%80%D1%85%D0%B8%D1%8F_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%BE%D0%B2) для использования при [объектно-ориентированной разработке](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) ПО. Хотя, и существуют среды разработки, предназначенные для нескольких [языков программирования](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) такие, как [Eclipse](http://ru.wikipedia.org/wiki/Eclipse_(%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8)), [NetBeans](http://ru.wikipedia.org/wiki/NetBeans_IDE), [Embarcadero RAD Studio](http://ru.wikipedia.org/wiki/Embarcadero_RAD_Studio) или [Microsoft Visual Studio](http://ru.wikipedia.org/wiki/Microsoft_Visual_Studio), обычно среды разработки предназначается для одного определённого языка программирования - как, например, [Visual Basic](http://ru.wikipedia.org/wiki/Visual_Basic), [Delphi](http://ru.wikipedia.org/wiki/Delphi_(%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8)), [Dev-C++](http://ru.wikipedia.org/wiki/Dev-C%2B%2B).

Иногда достаточно использовать только одну интегрированную среду разработки, но для больших проектов в среду разработки включаются разнородные продукты разных фирм, разных версий. Пример такого набора: файловый менеджер, набор вспомогательных утилит и пакетных файлов, С++Builder – как IDE, PLSQL Developer – для работы с СУБД Oracle, Cristal Reports – для создания отчетов , StarTeam – для ведения версий и поддержки коллективной работы.

Операционная среда разработки MS Visual Studio на сегодняшний день является предпочтительным выбором многих разработчиков, работающих на платформе Windows. В процессе своего развития MS Visual Studio имела несколько версий:

1. Visual Studio 97 – первая выпущенная версия Visual Studio. В ней впервые были собраны вместе различные средства разработки ПО. Система была выпущена в двух версиях Professional и Enterprise. Она включала в себя Visual Basic 5.0, Visual C++ 5.0, Visual J++ 1.1, Visual FoxPro 5.0, впервые появилась среда разработки ASP – Visual InterDev. Visual Studio 97 была первой попыткой [Microsoft](http://ru.wikipedia.org/wiki/Microsoft) создать единую среду для разработки на разных языках программирования: Visual C++, [Visual J++](http://ru.wikipedia.org/wiki/Visual_J%2B%2B), Visual InterDev, и MSDN использовали одну среду, называемую Developer Studio. Visual Basic и Visual FoxPro использовали отдельные среды для разработки.

2. Visual Studio 6.0  вышла в июне [1998](http://ru.wikipedia.org/wiki/1998). Это последняя версия Visual Studio, работающая на платформе [Win9x](http://ru.wikipedia.org/w/index.php?title=Win9x&action=edit&redlink=1). По-прежнему популярна среди программистов, использующих [Visual Basic](http://ru.wikipedia.org/wiki/Visual_Basic). Данная версия являлась основной средой разработки приложений под Windows от Microsoft, до появления платформы [.NET](http://ru.wikipedia.org/wiki/.NET_Framework).

3. Visual Studio .NET (кодовое имя Rainier; внутренняя версия 7.0) выпущена в феврале [2002 года](http://ru.wikipedia.org/wiki/2002_%D0%B3%D0%BE%D0%B4) (включает [.NET Framework](http://ru.wikipedia.org/wiki/.NET_Framework) 1.0). Service Pack 1 для Visual Studio .NET (2002) выпущен в марте [2005](http://ru.wikipedia.org/wiki/2005).

4. Visual Studio .NET 2003 (кодовое имя Everett; внутренняя версия 7.1) появилась в апреле [2003 года](http://ru.wikipedia.org/wiki/2003_%D0%B3%D0%BE%D0%B4) (включает [.NET Framework](http://ru.wikipedia.org/wiki/.NET_Framework) 1.1). Service Pack 1 для Visual Studio .NET 2003 выпущен 13 сентября [2006](http://ru.wikipedia.org/wiki/2006).

5. Visual Studio 2005 (кодовое имя Whidbey; внутренняя версия 8.0) выпущена в конце октября [2005 года](http://ru.wikipedia.org/wiki/2005_%D0%B3%D0%BE%D0%B4), последняя официально работающая на [Windows 2000](http://ru.wikipedia.org/wiki/Windows_2000), (включает [.NET Framework](http://ru.wikipedia.org/wiki/.NET_Framework) 2.0). В начале ноября [2005](http://ru.wikipedia.org/wiki/2005) также вышла серия продуктов в редакции Express: Visual C++ 2005 Express, Visual Basic 2005 Express, Visual C# 2005 Express и др.[[1]](http://ru.wikipedia.org/wiki/Microsoft_Visual_Studio#cite_note-0) 19 апреля 2006 редакция Express стала бесплатной. Service Pack 1 для VS2005 [[2]](http://ru.wikipedia.org/wiki/Microsoft_Visual_Studio#cite_note-1) и всех Express-редакций[[3]](http://ru.wikipedia.org/wiki/Microsoft_Visual_Studio#cite_note-2) выпущен 14 декабря 2006 года. Дополнительный патч для SP1, решающий проблему совместимости с Windows Vista выпущен 6 марта [2007](http://ru.wikipedia.org/wiki/2007).

6. Visual Studio 2008 (кодовое имя Orcas) выпущена 19 ноября [2007](http://ru.wikipedia.org/wiki/2007), одновременно с [.NET Framework](http://ru.wikipedia.org/wiki/.NET_Framework) 3.5. Нацелена на создание приложений для ОС [Windows Vista](http://ru.wikipedia.org/wiki/Windows_Vista) (но поддерживает и XP), [Office 2007](http://ru.wikipedia.org/wiki/Office_2007) и веб-приложений. Включает в себя [LINQ](http://ru.wikipedia.org/wiki/LINQ), новые версии языков [C#](http://ru.wikipedia.org/wiki/CSharp) и [Visual Basic](http://ru.wikipedia.org/wiki/Visual_Basic_.NET). В студию не вошёл [Visual J#](http://ru.wikipedia.org/wiki/Visual_J_Sharp). С [28 октября](http://ru.wikipedia.org/wiki/28_%D0%BE%D0%BA%D1%82%D1%8F%D0%B1%D1%80%D1%8F) [2008 года](http://ru.wikipedia.org/wiki/2008_%D0%B3%D0%BE%D0%B4) впервые доступна версия на русском языке.

7. Visual Studio 2010 (кодовое имя Hawaii, для Ultimate — Rosario) выпущена [12 апреля](http://ru.wikipedia.org/wiki/12_%D0%B0%D0%BF%D1%80%D0%B5%D0%BB%D1%8F) [2010 года](http://ru.wikipedia.org/wiki/2010_%D0%B3%D0%BE%D0%B4) вместе с [.NET Framework](http://ru.wikipedia.org/wiki/.NET_Framework) 4.0. Visual Studio включает поддержку языков [C#](http://ru.wikipedia.org/wiki/C_Sharp) 4.0 и [Visual Basic .NET](http://ru.wikipedia.org/wiki/Visual_Basic_.NET) 10.0, а также языка [F#](http://ru.wikipedia.org/wiki/F_Sharp), отсутствовавшего в предыдущих версиях.

Среда Visual Studio 2010 позволяет эффективно создавать сложные приложения в течение короткого периода времени. Модель данной среды существенно богаче ранних версий и использует такие понятия как решение (solution), проект, пространство имен (namespace) и сборка (assembly). Понятие проекта присутствует во многих средах, например, в среде Delphi. Файл проекта содержит перечисление исходных файлов и прочих ресурсов, из которых система будет строить приложение. В решение среды Visual Studio входят несколько проектов, которые могут быть зависимыми или независимыми друг от друга. Выделяется стартовый проект. Понятие сборки исходит из общеязыковой исполнительной среды CLR (Common Language Runtime). Среда CLR является наиболее революционным изобретением, с появлением которого процесс написания и выполнения приложений становиться принципиально другим [4].

Компилятор преобразует файлы с исходными кодами в коды на промежуточном языке MSIL (Microsoft Intermediate Language). Вместе с метаданными эти коды записываются PE-файлы (Portal Executable), имеющие расширение exe или dll в зависимости от типа проекта. Также может быть получен модуль с расширением netmodule, который не содержит метаданных.

Всего имеется 12 типов проектов. При загрузке PE-файлы «на лету» транслируются в команды реального процессора.

Каркас Framework.NET, обеспечивающий выполнение программ, не входит в Visual Studio, а является настройкой над операционной системой. Это аналог виртуальной Java-машины.

Сборка является минимальной единицей для развертывания приложений. Каждый тип сборки характеризуется уникальным идентификатором, идентифицируется цифровой подписью автора и уникальным номером версии. Между сборками и пространствами имен существует следующее соотношение. Сборка может содержать несколько пространств имен. В то же время, пространство имен может занимать несколько сборок. Сборка может иметь в своем составе как один, так и несколько файлов, которые объединяются в так называемом манифесте или описании сборки.

На уровне языка C# пространства имен, аналогично пакетам в Java, служат для структурирования проекта. Пространство имен включает один или несколько классов. В одном исходном файле может определяться несколько пространств имен и в тоже время одно пространство имен может определяться в нескольких файлах. И даже класс может располагаться в нескольких файлах (partial classes) .

Для начинающих программистов такое обилие возможностей может вызвать немалые затруднения. О масштабности и сложности среды можно судить по следующей сравнительной таблице трех сред:

|  |  |  |
| --- | --- | --- |
| Операционная среда | Число каталогов | Число типов файлов |
| Блэкбокс | 85 | 9 |
| C++Builder | 367 | 27 |
| Visual Studio | 3450 | 188 |

Интересной и перспективной представляется операционная среда Eclipse, разработанная в фирме IBM [6]. Первоначальной целью проекта было создание корпоративного стандарта IDE для разработки программ на разных языках под различные платформы. Потом проект был переименован в Eclipse и выделен в открытый доступ. Лицензия позволяет бесплатно использовать код и среду разработки, и при этом создавать закрытые коммерческие продукты. Благодаря этому система получила широкое распространение и для многих организаций стала корпоративным стандартом для разработки приложений.

В обзоре [1] экосистема Eclipse относится к консолидированным технологиям, годом широкого распространения которой является 2007 год.

Система реализована на языке Java и изначально представляла собой полноценную интегрированную среду для языка Java. В дальнейшем стали поддерживаться и другие языки. Первые версии были неудобны в том, что целевой продукт вынуждено включал лишнюю функциональность. Начиная с третьей версии, была переработана архитектура всей системы с целью максимального разделения модулей и взаимосвязи между ними. При этом модули Eclipse, образованные из согласованных наборов классов, давали функциональность целых подсистем, таких как подсистемы помощи, обновления продукта, обучения, презентации, многоязыковой поддержки и множество других. Разрабатывая приложение теперь можно постепенно наращивать функциональность, подключая уже готовые бесплатные компоненты. В терминологии Eclipse эти компоненты называются «подключаемыми модулями» или «плагинами» (Plugins). Такая технология становится типичной для развитых операционных сред. Платформа на основе этой технологии получила название RCP (Reach Client Platform), а приложения, соответственно, RCP-приложениями.

Графический интерфейс Eclipse построен с использованием графической библиотеки SWT, которая опирается на графику используемой операционной системы. Предусмотрена возможность создания настраиваемых стилей, которые позволяют произвольно менять как внешний вид приложения, так и поведение составляющих его компонентов. Многооконный интерфейс MDI, реализованный в ОС Windows еще в версии 3.1, заменен на альтернативный интерфейс, основанный на закладках. Этот интерфейс похож на интерфейс MS Visual Studio, но имеет и отличия. Режимы работы могут объединяться в одну панель, либо размещаться каждый в отдельной панели. Имеется встроенный механизм чередования панелей или механизм «быстрых» панелей, когда они не занимают место на экране, а вызываются кнопками по необходимости.

Интерфейс обладает богатыми и гибкими возможностями, некоторые из которых являются оригинальными и не встречаются в других операционных средах.

Данные, которые используются при разработке приложения, размещаются в одном каталоге – рабочей области. Графическая среда разработки (или просто рабочая среда) представляет собой инфраструктуру для управления ресурсами рабочей области и навигации по ресурсам. Рабочая среда включает одно или несколько окон, которые в свою очередь могут включать страницы. Каждая страница может включать так называемые проекции, которые управляют взаиморасположением редакторов и панелей для решения определенных задач.

Такая, сложная на первый взгляд, организация интерфейса, оказывается удобной именно благодаря использованию проекций. Проекцию можно рассматривать как набор визуальных компонент для выполнения конкретно поставленных задач. Например, проекция "Java" содержит необходимый набор для разработки java-приложений, а проекция "CVS" предназначена для оперирования удаленным репозитарием проектов при коллективной разработке. Каждая проекция имеет свой набор меню и панелей инструментов, состав которых можно настраивать и сохранять. Начальный макет проекции определяется разработчиком, но его можно изменять, открывая и закрывая панели и встраивая их в различные места окна Рабочей среды.

Основные визуальные компоненты рабочей среды – панели и редакторы.

Панель применяется для навигации по структурам объектов, отображения их свойств и внутренней структуры. В окне рабочей среды обычно размещается только один экземпляр определенного типа панели. Панели могут иметь собственные меню и панели инструментов, положение и внешний вид которых определяются выбранной темой представления. Действия, представленные элементами меню и кнопками панели инструментов, относятся только к входящим в панель элементам.

Редакторы чаще всего применяются для просмотра и редактирования ресурсов: файлов, таблиц базы данных и т.п. Любой ресурс в навигаторе ресурсов можно связать с необходимым для решения текущей задачи редактором. В отличие от панелей, изменения ресурсов в редакторах, подчиняются правилу открыть-сохранить-закрыть. В окне рабочей среды может быть размещено экземпляров одного и того же типа редактора для различных ресурсов. Для редакторов может быть создан отдельный раздел меню и панелей быстрого запуска, которые автоматически добавляются в главное меню и панель быстрого запуска приложения при активации редактора.

Процессы разработки приложений, которые включают модули расширений, также имеют отличия от традиционных подходов. При запуске среды разработки создается статический экземпляр рабочей среды, в котором пишется программный код, производится отладка и выполняются другие задачи производственного процесса. Когда же приложение запускается на тестирование, то создается динамический экземпляр рабочей среды, который включаются выбранные в конфигурации запуска разрабатываемые модули, а так же модули целевой платформы. Целевая платформа – это сборка продукта Eclipse, предназначенная для развертывания разрабатываемых приложений.

За последнее время произошли существенные шаги по упрощению разработки собственных приложений. Кроме мастеров по созданию RCP-приложений, появился мастер конфигурирования и экспорта продукта. Это существенно упрощает подготовку продукта для развертывания клиентом. Доступен экспорт, как в целевой каталог, так и в архив, который можно передать клиенту. Дополнительных инсталляторов не требуется, достаточно просто распаковать архив и создать ярлык на запускаемый файл. Есть возможность создания сборки сразу для нескольких операционных платформ. При желании, можно произвести экспорт продукта в каталог, используя содержимое которого создать автоматический инсталлятор для упрощения жизни неподготовленным пользователям.

Рассматривая перспективы программных технологий [1], можно было выделить следующие технологические концепции и методологии, которые наверняка повлекут собой изменения в операционных средах: программные шаблоны (паттерны), компонентная разработка, программы как сервис, сервисно-ориентированные архитектуры.

**Глава 5. Жизненный цикл программных систем**

**5.1. Понятие жизненного цикла программных систем**

Понятие *жизненного цикла* программных систем (ЖЦ ПС) является одним из базовых в программной инженерии. ЖЦ ПС определяется как период времени, который начинается с момента принятия решения о необходимости создания ПС и закачивается в момент ее полного изъятия из эксплуатации. Основным нормативным документом, регламентирующим состав процессов ЖЦ ПС, является международный стандарт ISO/IEC 12207: 1995 “Information Technology – Software Life Cycle Process” (ISO – International Organization for Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике). Этот стандарт определяет структуру ЖЦ, содержащую процессы. Действия и задачи, которые должны быть выполнены во время создания ПС.

В данном стандарте ПС (или программный продукт) определяется как набор компьютерных программ, процедур и, возможно, связанной с ними документацией и данных. Процесс определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные (Г. Майерс называет это трансляцией данных [ ]). Каждый процесс характеризуется определенными задачами и методами их решения. В свою очередь, каждый процесс разделен на набор действий, а каждое действие – на набор задач. Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения (естественно, при сохранении связей по входным данным).

Следует отметить, что в бывшем Советском Союзе, а затем в России создание программного обеспечения (ПО) первоначально, в 70-е годы прошлого столетия регламентировалось стандартами ГОСТ ЕСПД (Единой системы программной документации – серии ГОСТ 19.ХХХ), которые были ориентированы на класс относительно простых программ небольшого объема, создаваемых отдельными программистами. В настоящее время эти стандарты устарели концептуально и по форме, их сроки действия закончились и использование нецелесообразно.

Процессы создания автоматизированных систем (АС), в состав которых входит и ПО, регламентированы стандартами ГОСТ 34.601-90 ”Информационная технология. Комплекс стандартов на автоматизированные системы. Стадии создания”, ГОСТ 34.602-89 “Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы” и ГОСТ 34.603-92 “Информационная технология. Виды испытаний автоматизированных систем”. Однако многие положения этих стандартов устарели, а другие отражены недостаточно, чтобы их можно было использовать для серьезных проектов создания ПС. Поэтому в отечественных разработках целесообразно использовать современные международные стандарты.

В соответствии со стандартом ISO/IEC 12207 все процессы ЖЦ ПО разделены на три группы (рис. 5.1).



Рис. 5.1

В группах определено пять основных процессов: *приобретение, поставка, разработка, эксплуатация и сопровождение.* Восемь вспомогательных процессов обеспечивают выполнение основных процессов, а именно документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем. Четыре организационных процесса обеспечивают управление, создание инфраструктуры, усовершенствование и обучение.

**5.2. Основные процессы ЖЦ ПС**

5.2.1. *Процесс приобретения* состоит из действий и задач заказчика, приобретающего ПС. Данный процесс охватывает следующие действия:

1. инициирование приобретения;
2. подготовку заявочных предложений;
3. подготовку и корректировку договора;
4. надзор за деятельностью поставщика;
5. приемку и завершение работ.

Инициирование приобретения включает следующие задачи:

1. определение заказчиком своих потребностей в приобретении, разработке или усовершенствовании системы, программных продуктов или услуг;
2. анализ требований, предъявляемых к системе;
3. принятие решения относительно приобретения, разработки или усовершенствования существующего ПО;
4. проверку наличия необходимой документации, гарантий, сертификатов, лицензий и поддержки в случае приобретения программного продукта;
5. подготовку и утверждение плана приобретения, включающего требования к системе, тип договора, ответственность сторон и т.д.

Заявочные предложения должны содержать:

1. требования, предъявляемые к системе;
2. перечень программных продуктов;
3. условия приобретения и соглашения;
4. технические ограничения (например, по среде функционирования системы).

Заявочные предложения направляются к выбранному поставщику или нескольким поставщикам в случае тендера. Поставщик – это организация, которая заключает договор с заказчиком на поставку системы, ПО или программной услуги на условиях, оговоренных в договоре.

Подготовка и корректировка договора включает следующие задачи:

1. определение заказчиком процедуры выбора поставщика, включающей критерии оценки предложений возможных поставщиков;
2. выбор конкретного поставщика на основе анализа предложений;
3. подготовку и заключение договора с поставщиком;
4. внесение изменение (при необходимости) в договор в процессе его выполнения.

Надзор за деятельностью поставщика осуществляется в соответствии с действиями, предусмотренными в процессах совместной оценки и аудита. В процессе приемки подготавливаются и выполняются необходимые тесты. Завершение работ по договору осуществляется в случае удовлетворения всех условий приемки.

5.2.2. *Процесс поставки* охватывает действия и задачи, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой. Данный процесс включает следующие действия:

1. инициирование поставки;
2. подготовку ответа на заявочные предложения;
3. подготовку договора;
4. планирование работ по договору;
5. выполнение и контроль договорных работ и их оценку;
6. поставку и завершение работ.

Инициирование поставки заключается в рассмотрении поставщиком заявочных предложений и принятии решения, соглашаться с выставленными требованиями и условиями или предложить свои (согласовать).

Планирование включает следующие задачи:

1. принятие решения поставщиком относительно выполнения работ своими силами или с привлечением субподрядчика;
2. разработку поставщиком плана управления проектом, содержащего организационную структуру проекта, разграничение ответственности, технические требования к среде разработки и ресурсам, управление субподрядчиками и др.

5.2.3. *Процесс разработки* предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с заданными требованиями. Сюда включается оформление проектной и эксплуатационной документации, подготовка материалов, необходимых для проверки работоспособности и качества программных продуктов, материалов, необходимых для организации обучения персонала и др.

Процесс разработки включает следующие действия:

1. подготовительную работу;
2. анализ требований, предъявляемых к системе;
3. проектирование архитектуры системы;
4. анализ требований, предъявляемых к программному обеспечению;
5. проектирование архитектуры программного обеспечения;
6. детальное проектирование программного обеспечения;
7. кодирование и тестирование программного обеспечения;
8. интеграцию программного обеспечения;
9. квалификационное тестирование программного обеспечения;
10. интеграцию системы;
11. квалификационное тестирование системы;
12. установку программного обеспечения;
13. приемку программного обеспечения;

*Подготовительная работа* начинается с выбора модели ЖЦ ПО, соответствующей масштабу, значимости и сложности проекта. Действия и задачи процесса разработки должны соответствовать выбранной модели. Разработчик должен выбирать, адаптировать к условиям проекта и использовать согласованные с заказчиком стандарты, методы и средства разработки, а также составить план выполнения работ.

*Анализ требований, предъявляемых к системе*, подразумевает определение ее функциональных возможностей, пользовательских требований, требований к надежности, безопасности, требований к внешним интерфейсам, производительности и т.д. Требования к системе оцениваются, исходя из критериев реализуемости и возможности проверки при тестировании.

*Проектирование архитектуры системы* заключается в определении компонентов ее оборудования (аппаратуры), программного обеспечения и операций, выполняемых эксплуатирующим систему персоналом. Архитектура системы должна соответствовать требованиям, предъявляемым к системе, а также принятым проектным стандартам и методам.

*Анализ требований к программному обеспечению* предполагает определении следующих характеристик для каждого компонента ПО:

1. функциональных возможностей, включая характеристики производительности и среды функционирования компонента;
2. внешних интерфейсов;
3. спецификаций надежности и безопасности;
4. эргономических требований;
5. требований к используемым данным;
6. требований к установке и приемке;
7. требований к пользовательской документации;
8. требований к эксплуатации и сопровождению.

Требования к программному обеспечению оцениваются, исходя из критериев соответствия требованиям, предъявляемым к системе в целом, реализуемости и возможности проверки при тестировании.

*Проектирование архитектуры ПО* включает следующие задачи для каждого компонента ПО:

1. трансформацию требований к ПО в архитектуру, определяющую на высоком уровне структуру ПО и состав его компонентов;
2. разработку и документирование программных интерфейсов ПО и баз данных (БД);
3. разработку предварительной версии пользовательской документации;
4. разработку и документирование предварительных требований к тестам и плана интеграции ПО.

*Детальное проектирование ПО* включает следующие задачи:

1. описание компонентов ПО и интерфейсов между ними на более низком уровне, достаточном для последующего кодирования и тестирования;
2. разработку документирование детального проекта базы данных;
3. обновление (при необходимости) пользовательской документации;
4. разработку и документирование требований к тестам и плана тестирования компонентов ПО;
5. обновление плана интеграции ПО.

*Кодирование и тестирование ПО* включает следующие задачи:

1. кодирование и документирование каждого компонента ПО и базы данных, а также подготовку совокупности тестовых процедур и данных для их тестирования;
2. тестирование каждого компонента ПО и БД на соответствие предъявляемым к ним требованиям с последующим документированием результатов тестирования;
3. обновление документации (при необходимости);
4. обновление плана интеграции ПО.

*Интеграция ПО* предусматривает сборку разработанных компонентов ПО в соответствии с планом интеграции и тестирования агрегированных компонентов. Для каждого из агрегированных компонентов разрабатываются наборы тестов и тестовые процедуры, предназначенные для проверки каждого из *квалификационных требований* при последующем квалификационном тестировании. Квалификационное требование – это набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт как соответствующий своим спецификациям и готовый к использованию в условиях эксплуатации.

*Квалификационное тестирование ПО* проводится разработчиком в присутствии заказчика(по возможности) для демонстрации того, что ПО удовлетворяет своим спецификациям и готово к использованию в заданных условиях эксплуатации. Такое тестирование выполняется для каждого компонента программного продукта по всем разделам требований при широком варьировании тестов. При этом также проверяется полнота технической и пользовательской документации и ее адекватность самим компонентам ПО.

*Интеграция системы* заключается в сборке всех ее компонентов, включая ПО и оборудование. После интеграции система, в свою очередь, квалификационному тестированию на соответствие предъявляемым к ней требованиям. При этом также производится оформление и проверка полного комплекта документации на систему.

*Установка ПО* осуществляется разработчиком в соответствии с планом в той среде и на том оборудовании, которые предусмотрены договором. В процессе установки проверяется работоспособность ПО и БД. Если устанавливаемое ПО заменяет существующую систему, разработчик должен обеспечить их параллельное функционирование в соответствии с договором.

*Приемка ПО* предусматривает оценку результатов квалификационного тестирования ПО и системы и документирование результатов оценки, которые производятся заказчиком с помощью разработчика. Разработчик выполняет окончательную передачу ПО заказчику в соответствии с договором, обеспечивая при этом необходимое обучение и поддержку.

5.2.4. *Процесс эксплуатации* охватывает действия и задачи организации-оператора, эксплуатирующего систему. Процесс эксплуатации включает следующие действия:

1. подготовительную работу, которая включает проведение оператором следующих задач:

* планирование действий и работ, выполняемых в процессе эксплуатации, и установку эксплуатационных стандартов;
* определение процедур локализации и разрешения проблем, возникающих в процессе эксплуатации.

1. эксплуатационное тестирование, осуществляемое для каждой очередной редакции программного продукта, после чего эта редакция передается в эксплуатацию;
2. собственно эксплуатацию системы, которая выполняется в предназначенной для этого среде в соответствии с пользовательской документацией;
3. поддержку пользователей – оказание помощи и консультаций при обнаружении ошибок в процессе эксплуатации ПО.

5.2.5. *Процесс сопровождения* представляет собой действия и задачи, выполняемые сопровождающей организацией, при изменениях (модификациях) программного продукта и соответствующей документации, вызванных возникшими проблемами или потребностями в модернизации или адаптации ПО.

Изменения, вносимые в соответствующее ПО, не должны нарушать его целостность. Процесс сопровождения включает его перенос в другую среду (миграцию) и заканчивается снятием ПО с экплуатации.

Процесс сопровождения охватывает следующие действия:

1. подготовительную работу(планирование действий и работ, определение процедур локализации и разрешения проблем, возникающих в процессе сопровождения);
2. анализ проблем и запросов на модификацию ПО (анализ сообщений о возникшей проблеме или запроса на модификацию, оценка масштаба, стоимости модификации, получаемого эффекта, оценка целесообразности модификации);
3. модификацию ПО (внесение изменений в компоненты программного продукта и документацию в соответствии с правилами процесса разработки);
4. проверку и приемку (в части целостности модифицируемой системы);
5. перенос ПО в другую среду (конвертирование программ и данных, параллельная эксплуатация ПО в старой и новой среде в течение некоторого периода времени);
6. снятие ПО с эксплуатации по решению заказчика при участии эксплуатирующей организации, службы сопровождения и пользователей. При этом программные продукты и документации подлежат архивированию в соответствии с договором.

**5.3. Вспомогательные процессы ЖЦ ПО**

5.3.1. *Процесс документирования*. Предусматривает формализованное описание информации, созданной в течение ЖЦ ПО. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких, как руководство, технические специалисты и пользователи системы.

Процесс документирования включает следующие действия:

1. подготовительную работу;
2. проектирование и разработку;
3. выпуск документации;
4. сопровождение.

5.3.2*. Процесс управления конфигурацией* включает административные и технические процедуры на всем протяжении ЖЦ ПО для определения состояния компонентов ПО, описания и подготовки отчетов о состоянии компонентов ПО и запросов на модификацию, обеспечения полноты, совместимости и корректности компонентов ПО, управления хранением и поставкой ПО.

Согласно стандарту IEEE-90 под конфигурацией ПО понимается совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПО. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации по управлению конфигурацией ПО отражены в стандарте ISO/IEC 15288 “Information Technology. Software Life Cycle Process. Configuration Management for Software”.

Процесс управления конфигурацией включает следующие действия:

1. подготовительную работу, заключающуюся в планировании управления конфигурацией;
2. идентификацию конфигурации, устанавливающую правила, с помощью которых однозначно идентифицируются компоненты ПО и их версии. При этом каждому компоненту однозначно соответствует комплект документации;
3. контроль конфигурации – действие, предназначенное для систематической оценки предлагаемых модификаций ПО и координированной их реализации с учетом эффективности каждой модификации и затрат на ее выполнение;
4. учет состояния конфигурации, представляющий собой регистрацию состояния компонентов ПО. Обеспечивает подготовку отчетов о реализованных и отвергнутых модификациях версий компонентов ПО. Совокупность отчетов дает однозначное отражение текущего состояния системы и ее компонентов, а также обеспечивает ведение истории модификаций;
5. оценку конфигурации, заключающуюся в определении функциональной полноты компонентов ПО, а также соответствия их физического состояния текущему техническому описанию;
6. управление выпуском и поставку, охватывающие изготовление эталонных копий программ и документации, их хранение и поставку пользователям в соответствии с порядком, принятом в организации.

5.3.3. *Процесс обеспечения качества* должен обеспечивать гарантии того, что ПО и процессы его ЖЦ соответствуют заданным требованиям и утвержденным планам. *Под качеством ПО понимается совокупность свойств, которая характеризует способность ПО удовлетворять заданным требованиям.* Для получения достоверных оценок о создаваемом ПО процесс обеспечения его качества должен происходить независимо от субъектов, непосредственно связанных с разработкой программного продукта. При этом могут использоваться результаты других вспомогательных процессов, таких, как *верификация, аттестация*, совместная оценка, аудит и разрешение проблем.

Процесс обеспечения качества включает следующие действия:

1. подготовительную работу (координацию с другими вспомогательными процессами и планирование самого процесса обеспечения качества ПО с учетом используемых стандартов, методов, процедур и средств);
2. обеспечение качества продукта, подразумевающего гарантированное полное соответствие ПО и его документации требования заказчика, предусмотренным в договоре;
3. обеспечение качества процесса, предполагающее гарантированное соответствие процессов ЖЦ ПО, методов разработки, среды разработки и квалификации персонала условиям договора, установленным стандартам и процедурам;
4. обеспечение прочих показателей качества ПО, осуществляемое в соответствии с условиями договора и стандартом качества ISO 9001.

5.3.4. *Процесс верификации* состоит в определении того факта, что ПО, являющееся результатом некоторой деятельности, полностью удовлетворяет требованиям или условиям, обусловленным предшествующими действиями. Для повышения эффективности всего процесса ЖЦ ПО верификация должна как можно раньше интегрироваться с использующими ее процессами (т.е. с поставкой, разработкой, эксплуатацией). Процесс верификации может включать анализ, оценку и тестирование.

Верификация может проводиться с различными степенями независимости (от самого исполнителя до специалистов другой организации, не зависящей от поставщика, разработчика и т.д.). В процессе верификации проверяются следующие условия:

1. непротиворечивость требований, предъявляемых к системе и степень учета потребностей пользователей;
2. возможность поставщика выполнить заданные требования;
3. соответствие выбранных процессов ЖЦ ПО условиям договора;
4. адекватность стандартов, процедур и среды разработки процессам ЖЦ ПО;
5. соответствие проектных спецификаций ПО заданным требованиям;
6. корректность описания в проектных спецификациях входных и выходных данных, последовательности событий, интерфейсов, логики и т.д.;
7. соответствие кода проектным спецификациям и требованиям;
8. тестируемость и корректность кода, его соответствие принятым стандартам кодирования;
9. корректность интеграции компонентов ПО в систему;
10. адекватность, полнота и непротиворечивость документации.

5.3.5. *Процесс аттестации* предназначен для определения полноты соответствия заданных требований и созданного ПО их конкретному функциональному назначению (тому, что требуется потребителю). Под аттестацией обычно понимается подтверждение и оценка достоверности проведенного тестирования программного продукта. Аттестация должна гарантировать полное соответствие ПО спецификациям, требованиям и документации, а также возможность безопасного и надежного применения ПО пользователем.

Аттестация, как и верификация, может осуществляться с различными степенями независимости (вплоть до организации, не зависящей от поставщика, разработчика, оператора или службы сопровождения).

5.3.6. *Процесс совместной оценки* предназначен для оценки состояния работ по проекту и программному продукту, создаваемому при выполнении этих работ. Он сосредоточен в основном на контроле планирования и управления ресурсами, персоналом, аппаратурой и инструментальными средствами проекта.

Оценка применяется как на уровне управления проектом, так и на уровне технической реализации проекта и проводится в течение всего срока действия договора. Данный процесс может выполняться двумя сторонами, участвующими в договоре, при этом одна сторона проверяет другую.

5.3.7. *Процесс аудита* представляет собой определение соответствия проекта и продукта требованиям, планам и условиям договора. Аудит может выполняться двум любыми сторонами, участвующими в договоре, когда одна сторона проверяет другую.

Аудит – это ревизия (проверка), проводимая компетентным органом (лицом) в целях обеспечения независимой оценки степени соответствия ПО или процессов установленным требованиям.

Аудит служит для установления соответствия реальных работ и отчетов требованиям, планам и контракту. Аудиторы не должны иметь прямой зависимости от разработчиков ПО. Они определяют состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность тестирования и др.

5.3.7. *Процесс разрешения проблем* предусматривает анализ и разрешение проблем (включая обнаруженные несоответствия), которые обнаружены в ходе разработки, эксплуатации или других процессов независимо от их происхождения или источника.

**5.4. Организационные процессы ЖЦ ПО**

5.4.1*. Процесс управления* состоит из действий и задач, которые могут выполняться любой стороной, управляющей своими процессами. Данная сторона (менеджер) отвечает за управление выпуском продукта, управление проектом и управление задачами соответствующих процессов, таких, как приобретение, поставка, разработка, эксплуатация, сопровождение и др.

Процесс управления включает следующие действия:

1. инициирование и определение области управления – менеджер должен убедиться, что необходимые для управления ресурсы (персонал, оборудование и технология) имеются в его распоряжении в достаточном количестве;
2. планирование, как действие, подразумевает выполнение следующих задач:

* составление графиков выполнения работ;
* оценку затрат;
* выделение требуемых ресурсов;
* распределение ответственности;
* оценку рисков, связанных с конкретными задачами;
* создание инфраструктуры управления.

5.4.2*. Процесс создания инфраструктуры* охватывает выбор и поддержку технологий, стандартов и инструментальных средств, используемых для разработки, эксплуатации или сопровождения ПО. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура, в свою очередь, является одним из объектов управления конфигурацией.

Процесс создания инфраструктуры включает следующие действия:

* подготовительную работу;
* создание инфраструктуры;
* сопровождение инфраструктуры.

5.4.3*. Процесс* усовершенствования предусматривает оценку, измерение, контроль и собственно усовершенствование процессов ЖЦ ПО. Этот процесс включает три основных действия:

* создание процесса;
* оценку процесса;
* усовершенствование процесса.

Усовершенствование процессов ЖЦ ПО направлено на повышение производительности труда всех участвующих в них специалистов за счет совершенствования используемой технологии, методов управления, выбора инструментальных средств и обучения персонала. Усовершенствование основано на анализе достоинств и недостатков каждого процесса. Такому анализу способствует накопление в организации исторической, технической, экономической и иной информации по реализованным проектам.

5.4.3*. Процесс обучения* включает первоначальное обучение и последующее постоянное повышение квалификации персонала и состоит из трех действий:

* подготовительной работы;
* работки учебных материалов;
* реализации планов обучения.

**5.5. Взаимосвязь между процессами ЖЦ ПО**

Процессы ЖЦ ПО, регламентируемые стандартом ISO/IEC 12207 могут использоваться различными организациями в конкретных проектах самым различным образом. Тем не менее стандарт предлагает некоторый базовый набор взаимодействий между процессами с различных точек зрения (либо в различных аспектах), который показан на рис. 2.

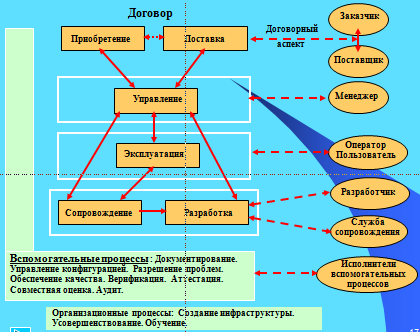


Рис. 2

Такими аспектами являются:

1. договорной аспект, в котором заказчик и поставщик вступают в договорные отношения и реализуют процессы приобретения и поставки;
2. аспект управления, включающий действия управления лицами, участвующими в ЖЦ ПО (поставщик, заказчик, разработчик, оператор и др.);
3. аспект эксплуатации, включающий действия оператора по предоставлению услуг пользователям системы;
4. инженерный аспект, содержащий действия разработчика или службы сопровождения по решению технических задач, связанных с разработкой или модификацией программных продуктов;
5. аспект поддержки, связанный с реализацией вспомогательных процессов, с помощью которых службы поддержки предоставляют необходимые услуги всем остальным участникам работ. В этом аспекте можно выделить аспект управления качеством ПО, включающий процессы обеспечения качества, верификацию, аттестацию, совместную оценку и аудит.

Организационные процессы выполняются на корпоративном уровне, или на уровне всей организации в целом, создавая базу для реализации и постоянного совершенствования процессов ЖЦ ПО.

1. По другим сведениям, первый компьютер был создан в Англии в 1943 году для расшифровки кодов немецких подводных лодок. [↑](#footnote-ref-1)