**Лекции по дисциплине «Межплатформенное программирование»**

**Кроссплатформенные языки программирования**

## *Понятие и классификация языков программирования*

Язык программирования — формальная знаковая система, предназначенная для записи компьютерных программ. Он определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполнит исполнитель (компьютер) под ее управлением.

Существует большое количество языков программирования. Приведём их классификацию.

По наиболее распространенной классификации все языки программирования, в соответствии с тем, в каких терминах необходимо описать задачу, делят на языки низкого и высокого уровня.

Если язык программирования близок к естественному языку, то он называется языком высокого уровня, если ближе к машинным командам, – языком низкого уровня.

Языки программирования также можно классифицировать на процедурные и непроцедурные.

В процедурных языках программа явно описывает действия, которые необходимо выполнить, а результат задается только способом получения его при помощи некоторой процедуры, которая представляет собой определенную последовательность действий.

Среди процедурных языков выделяют в свою очередь структурные и операционные языки. В структурных языках одним оператором записываются целые алгоритмические структуры: ветвления, циклы и т.д. В операционных языках для этого используются несколько операций. Широко распространены следующие структурные языки: Паскаль, Си, Ада, ПЛ/1. Среди операционных известны Фортран, Бейсик, Фокал.

Непроцедурное (декларативное) программирование появилось в начале 70-х годов 20 века. К непроцедурному программированию относятся функциональные и логические языки.

В функциональных языках программа описывает вычисление некоторой функции. Обычно эта функция задается как композиция других, более простых, те в свою очередь делятся на еще более простые задачи и т.д. Один из основных элементов функциональных языков — рекурсия. Оператора присваивания и циклов в классических функциональных языках нет.

В логических языках программа вообще не описывает действий. Она задает данные и соотношения между ними. После этого системе можно задавать вопросы. Машина перебирает известные и заданные в программе данные и находит ответ на вопрос. Порядок перебора не описывается в программе, а неявно задается самим языком. Классическим языком логического программирования считается Пролог. Программа на Прологе содержит, набор предикатов-утверждений, которые образуют проблемно-ориентированную базу данных и правила, имеющие вид условий.

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков.

В соответствии с данной методологией любая программа представляет собой структуру, построенную из трёх типов базовых конструкций: последовательное исполнение, ветвление, цикл. Повторяющиеся фрагменты могут оформляться в виде подпрограмм (процедур или функций).

Объектно-ориентированное программирование (в дальнейшем ООП) — методология разработки программного обеспечения, в которой основными концепциями являются понятия объектов и классов. На таких языках не описывают подробной последовательности действий для решения задачи, хотя они содержат элементы процедурного программирования.

1.2 Машинный код

Машинный код (платформенно-ориентированный код), машинный язык — [система команд](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4) (набор кодов операций) конкретной [вычислительной машины](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80), которая [интерпретируется](https://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) непосредственно [процессором](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) или [микропрограммами](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BA%D0%BE%D0%B4) этой вычислительной машины.

[Компьютерная программа](https://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), записанная на машинном языке, состоит из машинных инструкций, каждая из которых представлена в машинном коде в виде т. н. [опкода](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8) — [двоичного кода](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) отдельной операции из системы команд машины. Для удобства [программирования](https://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) вместо числовых опкодов, которые только и понимает процессор, обычно используют их условные буквенные [мнемоники](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%B5%D0%BC%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0). Набор таких мнемоник, вместе с некоторыми дополнительными возможностями (например, некоторыми макрокомандами, директивами), называется [языком ассемблера](https://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).

Каждая модель процессора имеет собственный набор команд, хотя во многих моделях эти наборы команд сильно перекрываются. Говорят, что процессор A совместим с процессором B, если процессор A полностью «понимает» машинный код процессора B. Если процессоры A и B имеют некоторое подмножество инструкций, по которым они взаимно совместимы, то говорят, что они одной «архитектуры» (имеют одинаковую [архитектуру набора команд](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BD%D0%B0%D0%B1%D0%BE%D1%80%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4)).

**Машинная инструкция.** Каждая машинная инструкция выполняет определённое действие, такое как операция с данными (например, [сложение](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) или копирование [машинного слова](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D0%BE%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%BE) в [регистре](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0) или в [памяти](https://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%D0%B0%D0%BC%D1%8F%D1%82%D1%8C)) или переход к другому участку кода (изменение порядка исполнения; при этом переход может быть [безусловным](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4%D0%B0) или [условным](https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4), зависящим от результатов предыдущих инструкций). Любая [исполнимая программа](https://ru.wikipedia.org/w/index.php?title=%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D0%BC%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C&action=edit&redlink=1) состоит из последовательности таких атомарных машинных операций.

Операции, записываемые в виде одной машинной инструкции, можно разделить на «простые» ([элементарные операции](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F)) и «сложные». Кроме того, большинство современных процессоров состоит из отдельных «исполнительных устройств» — вычислительных блоков, которые умеют исполнять лишь ограниченный набор простейших операций. При исполнении очередной инструкции специальный блок процессора — декодер — [транслирует](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80) (декодирует) её в последовательность элементарных операций, понимаемых конкретными исполнительными устройствами.

[Архитектура набора команд](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BD%D0%B0%D0%B1%D0%BE%D1%80%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4) процессора определяет, какие операции он способен выполнять, и какой машинной инструкции какие числовые [коды операций](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8) (опкоды) соответствуют. Опкоды бывают постоянной длины   
(у [RISC](https://ru.wikipedia.org/wiki/RISC)-, [MISC](https://ru.wikipedia.org/wiki/MISC)-архитектур) и диапазонной (у [CISC](https://ru.wikipedia.org/wiki/CISC)-архитектур; например: для архитектуры [x86](https://ru.wikipedia.org/wiki/X86) команда имеет длину от 8 до 120 битов).

Современные [суперскалярные процессоры](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%BF%D0%B5%D1%80%D1%81%D0%BA%D0%B0%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) способны выполнять несколько машинных инструкций за один [такт](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BA%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB).

**Машинный код как язык программирования**

Машинный код можно рассматривать как примитивный [язык программирования](https://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) или как самый низкий уровень представления [скомпилированных](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) или [ассемблированных](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80) компьютерных программ. Хотя вполне возможно создавать программы прямо в машинном коде, сейчас это делается редко в силу громоздкости кода и трудоёмкости ручного управления ресурсами процессора, за исключением ситуаций, когда требуется экстремальная [оптимизация](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Поэтому подавляющее большинство программ пишется на [языках более высокого уровня](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%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) и транслируется в машинный код [компиляторами](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80). Машинный код иногда называют нативным кодом (также собственным или родным кодом — от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) native code), когда говорят о платформенно-зависимых частях языка или библиотек.

Программы на [интерпретируемых языках](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D1%8B%D0%B9_%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) (таких как [Basic](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D0%B9%D1%81%D0%B8%D0%BA) или [Python](https://ru.wikipedia.org/wiki/Python)) не транслируются в машинный код; вместо этого они либо исполняются непосредственно [интерпретатором](https://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) языка, либо транслируются в псевдокод ([байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4)). Однако интерпретаторы этих языков (которые сами можно рассматривать как процессоры), как правило, представлены в машинном коде.

**Микрокод.** В некоторых [компьютерных архитектурах](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%B0) поддержка машинного кода реализуется ещё более низкоуровневым слоем программ, называемых [микропрограммами](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BA%D0%BE%D0%B4). Это позволяет обеспечить единый интерфейс машинного языка у всей линейки или семейства компьютеров, которые могут иметь значительные структурные отличия между собой, и облегчает перенос программ в машинном коде между разными моделями компьютеров. Примером такого подхода является семейство компьютеров [IBM System/360](https://ru.wikipedia.org/wiki/IBM_System/360) и их преемников: несмотря на разные [шины](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D0%BD%D0%B0_(%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80)) шириной от 8 до 64 бит и выше, тем не менее, у них общая архитектура на уровне машинного языка.

Использование слоя микрокода для реализации [эмулятора](https://ru.wikipedia.org/wiki/%D0%AD%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) позволяет компьютеру представлять архитектуру совершенно другого компьютера. В линейке System/360 это использовалось для переноса программ с более ранних машин IBM на новое семейство — например, эмулятор IBM 1401/1440/1460 на IBM S/360 model 40.

**Абсолютный и позиционно-независимый код.**

**Абсолютный код** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) absolute code) — программный код, пригодный для прямого выполнения процессором, то есть код, не требующий дополнительной обработки (например, [разрешения ссылок](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA) между различными частями кода или привязки к адресам в памяти, обычно выполняемой [загрузчиком программ](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC)). Примерами абсолютного кода являются исполнимые файлы в формате [.COM](https://ru.wikipedia.org/wiki/.COM) и загрузчик ОС, располагаемый в [MBR](https://ru.wikipedia.org/wiki/MBR). Часто абсолютный код понимается в более узком смысле как позиционно-зависимый код (то есть код, привязанный к определённым адресам памяти).

**Позиционно-независимый код** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) position-independent code) — программа, которая может быть размещена в любой области памяти, так как все ссылки на ячейки памяти в ней относительные (например, относительно [счётчика команд](https://ru.wikipedia.org/wiki/%D0%A1%D1%87%D1%91%D1%82%D1%87%D0%B8%D0%BA_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4)). Такую программу можно переместить в другую область памяти в любой момент, в отличие от перемещаемой программы, которая хотя и может быть загружена в любую область памяти, но после загрузки должна оставаться на том же месте.

Возможность создания позиционно-независимого кода зависит от архитектуры и системы команд целевой платформы. Например, если во всех инструкциях перехода в системе команд должны указываться абсолютные адреса, то код, требующий переходов, практически невозможно сделать позиционно-независимым. В архитектуре [x86](https://ru.wikipedia.org/wiki/X86) непосредственная адресация в инструкциях работы с данными представлена только абсолютными адресами, но поскольку адреса данных считаются относительно [сегментного регистра](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%B3%D0%BC%D0%B5%D0%BD%D1%82%D0%BD%D1%8B%D0%B9_%D1%80%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80), который можно поменять в любой момент, это позволяет создавать позиционно-независимый код со своими ячейками памяти для данных. Кроме того, некоторые ограничения набора команд могут сниматься с помощью [самомодифицирующегося кода](https://ru.wikipedia.org/wiki/%D0%A1%D0%B0%D0%BC%D0%BE%D0%BC%D0%BE%D0%B4%D0%B8%D1%84%D0%B8%D1%86%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B8%D0%B9%D1%81%D1%8F_%D0%BA%D0%BE%D0%B4) или нетривиальных последовательностей инструкций.

**1.3 Байт-код**

Байт-код (байткод; англ. bytecode, также иногда p-код, p-code от portable code) — стандартное промежуточное представление, в которое может быть переведена компьютерная программа автоматическими средствами. По сравнению с исходным кодом, удобным для создания и чтения человеком, байт-код — это компактное представление программы, уже прошедшей синтаксический и семантический анализ. В нём в явном виде закодированы типы, области видимости и другие конструкции. С технической точки зрения байт-код представляет собой машинно-независимый код низкого уровня, генерируемый транслятором из исходного кода.

Многие современные языки программирования, особенно интерпретируемые, используют байт-код для облегчения и ускорения работы интерпретатора. Трансляция в байт-код является методом, промежуточным по эффективности между прямой интерпретацией и компиляцией в машинный код.

По форме байт-код похож на машинный код, но предназначен для исполнения не реальным процессором, а виртуальной машиной. В качестве виртуальной машины обычно выступает интерпретатор соответствующего языка программирования (иногда дополненный JIT- или AOT-компилятором). Спецификации байт-кода и исполняющих его виртуальных машин могут сильно различаться для разных языков. Тем не менее, большинство инструкций байт-кода обычно эквивалентны одной или нескольким командам ассемблера.

Байт-код называется так, потому что длина каждого кода операции традиционно составляет один байт. Каждая инструкция обычно представляет собой однобайтовый код операции (от 0 до 255), за которым могут следовать различные параметры, например, номер регистра или адрес в памяти.

Программа на байт-коде обычно выполняется интерпретатором байт-кода. Преимущество байт-кода в большей эффективности и портируемости, то есть один и тот же байт-код может исполняться на разных платформах и архитектурах, для которых реализован интерпретатор. То же самое преимущество дают непосредственно интерпретируемые языки, однако, поскольку байт-код обычно менее абстрактен и более компактен, чем исходный код, эффективность интерпретации байт-кода обычно выше, чем чистая интерпретация исходного кода или интерпретация АСД. Кроме того, интерпретатор байт-кода зачастую проще интерпретатора исходного кода и его проще перенести (портировать) на другую аппаратную платформу.

В высокопроизводительных реализациях виртуальных машин может применяться комбинация интерпретатора и JIT-компилятора, который во время исполнения программы транслирует часто используемые фрагменты байт-кода в машинный код, применяя при этом различные оптимизации. Вместо JIT-компиляции может применяться AOT-компилятор, транслирующий байт-код в машинный код предварительно, до исполнения.

В то же время возможно создание процессоров, для которых данный байт-код является непосредственно машинным кодом (такие экспериментальные процессоры создавались, например, для языков Java и Форт).

**1.4 Широко распространённые платформы, использующие байт-код**

**1.4.1 Байт-код Java** (стековая виртуальная машина), исполняемый различными виртуальными машинами Java. Платформа была создана компанией Sun для языка Java, но стала использоваться и для других языков; существуют десятки высокопроизводительных реализаций JVM, использующих JIT-компиляторы.

Программы на Java обычно компилируются в class-файлы (англ.)русск., содержащие байт-код Java. Эти универсальные файлы передаются на различные целевые машины.

Существуют варианты трансляции Java в байт-код регистровых машин, например, в виртуальной машине Dalvik (с JIT-компиляцией) или при AOT-компиляции в ART.

**1.4.2 Платформа Microsoft .NET** использует стековый байт-код Intermediate Language (CIL, MSIL)[8], исполняемый с помощью Common Language Runtime (CLR), создана Microsoft для C# и некоторых других языков.

**1.4.3 Сценарный язык JavaScript** выполняется различными высокопроизводительными «движками», в основном, встроенными в веб-браузеры, часто с возможностью JIT-оптимизации. Многие интерпретаторы построены с применением байт-кода, однако программы на Javascript распространяются в виде исходных кодов.

**1.4.4 Сценарный язык ActionScript** транслируется в стековый байт-код, распространяется в составе swf- и pdf-файлов, и выполняется виртуальными машинами в Adobe Flash и Adobe Acrobat.

**1.4.5 Компилятор Clipper** создает исполняемый файл, в который включен байт-код, транслированный из исходного текста программы, и виртуальная машина, исполняющая этот байт-код.

В ранних реализациях Visual Basic (до версии 6) использовался высокоуровневый Microsoft p-code.

## **1.5 Платформы java и .net**

Для преобразования программы, написанной на одном из языков высокого уровня, в программу, состоящую из машинных команд, используется транслятор (англ. translator — переводчик).

Трансляторы реализуются в виде компиляторов или интерпретаторов. С точки зрения выполнения работы компилятор и интерпретатор существенно различаются.

Компилятор (англ. compiler — составитель, собиратель) читает всю программу целиком, делает ее перевод и создает законченный вариант программы на машинном языке или байт-коде, который затем и выполняется.

Интерпретатор (англ. interpreter — истолкователь, устный переводчик) переводит и выполняет программу строка за строкой.

Часть действий программа не может выполнять напрямую с оборудованием, а обращается к операционной системе (ОС) с соответствующими запросами. Набор классов, процедур, функций, структур и констант, предоставляемых ОС для использования в программах называется интерфейсом прикладного программирования (API — application programming interface).

По такому принципу работают многие языки программирования, например C и C++ (рис. 1).

Аппаратная платформа

ОС

API

Программа

Рис. 1.

Создание С знаменует собой начало современной эпохи программирования. Язык С был разработан Деннисом Ритчи в 1970-е годы для программирования на мини-ЭВМ DEC PDP-11 под управлением операционной системы Unix. Несмотря на то, что в ряде предшествовавших языков, в особенности Pascal, был достигнут значительный прогресс, именно С установил тот образец, которому до сих пор следуют в программировании. Язык С появился в результате революции в структурном программировании в 1960-е годы. До появления структурного программирования писать большие программы было трудно, поскольку логика программы постепенно вырождалась в запутанный клубок безусловных переходов, вызовов и возвратов, которые трудно отследить. В структурированных языках программирования этот недостаток устранялся путем ввода строго определенных управляющих операторов, подпрограмм с локальными переменными и других усовершенствований.

Язык C++ был разработан в 1979 году Бьярне Страуструпом, работавшим в компании Bell Laboratories. Язык С полностью входит в состав C++, а следовательно, С служит основанием, на котором зиждется C++. Большая часть дополнений, введенных Страуструпом, обеспечивала плавный переход к ООП. И вместо того, чтобы изучать совершенно новый язык, программирующему на С требовалось лишь освоить ряд новых свойств, чтобы воспользоваться преимуществами методики ООП.

Подход, показанный на рис. 1 обладает одним важным недостатком — плохая переносимость приложений, т.е. при необходимости использовать программу в другой ОС необходимо частично переписать её и заново скомпилировать.

Следующим важным шагом в развитии языков программирования стала разработка языка Java. Работа над языком Java, началась в 1991 году в компании Sun Microsystems. Java представляет собой структурированный, объектно-ориентированный язык с синтаксисом и конструктивными особенностями, унаследованными от C++. До появления на широкой арене Интернета большинство программ писалось, компилировалось и предназначалось для конкретного процессора и операционной системы. Как известно, программисты всегда стремились повторно использовать свой код, но, несмотря на это, легкой переносимости программ из одной среды в другую уделялось меньше внимания, чем более насущным задачам. Тем не менее с появлением Интернета, когда в глобальную сеть связывались разнотипные процессоры и операционные системы, застаревшая проблема переносимости программ вновь возникла с неожиданной остротой. Для решения проблемы переносимости потребовался новый язык, и им стал Java.

Аппаратная платформа

ОС Windows

API

Программа на Java

JRE для Windows

Аппаратная платформа

ОС Linux

API

JRE для Linux

Рис. 2.

Самым важным свойством (и причиной быстрого признания) Java является способность создавать межплатформенный, переносимый код. Переносимость программ на Java достигается благодаря трансляции исходного кода в промежуточный, называемый байт-кодом. Этот байт-код затем выполнялся виртуальной машиной Java (JVM) — основной частью исполняющей системы Java.

Java Runtime Environment (сокр. JRE) — минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки.

Таким образом, программа на Java могла выполняться в любой среде, для которой была доступна JVM. А поскольку JVM реализуется относительно просто, то она сразу же стала доступной для большого числа сред. Применением байт-кода Java коренным образом отличается от С и C++, где исходный код практически всегда компилируется в исполняемый машинный код, который, в свою очередь, привязан к конкретному процессору и операционной системе. Так, если требуется выполнить программу на С или C++ в другой системе, ее придется перекомпилировать в машинный код специально для данной вычислительной среды. Следовательно, для создания программы на С или C++, которая могла был выполняться в различных средах, потребовалось бы несколько разных исполняемых версий этой программы. Это оказалось бы не только непрактично, но и дорого. Изящным и рентабельным решением данной проблемы явилось применение в Java промежуточного кода. Именно это решение было в дальнейшем приспособлено для целей языка С#.

К недостаткам концепции виртуальной машины относят то, что исполнение байт-кода виртуальной машиной может снижать производительность программ и алгоритмов, реализованных на языке Java в 1.5 — 2 раза.

Аппаратная платформа

ОС Windows

API

Программа на C#

.net framework

Аппаратная платформа

ОС Linux

API

.net framework

Программа на VB.net

Рис. 3.

Язык С# непосредственно связан с С, C++ и Java. И это не случайно. Ведь это три самых широко распространенных и признанных во всем мире языка программирования. Кроме того, на момент создания С# практически все профессиональные программисты уже владели С, C++ или Java.

Родственные связи С# и Java более сложные. Как пояснялось выше, Java также происходит от С и C++ и обладает общим с ними синтаксисом и объектной моделью. Как и Java, C# предназначен для получения переносимого кода, но С# не происходит непосредственно от Java. Напротив, С# и Java — это близкие, но не кровные родственники, имеющие общих предков, но во многом отличающиеся друг от друга.

## **1.6 Среда 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.

Однако сам по себе компилятор не решает полностью всех задач, связанных с разработкой новой программы.

Система программирования (среда программирования, интегрированная среда разработки, англ. IDE — Integrated development environment) — комплекс программных средств, предназначенных для написания, тестирования и отладки программного обеспечения.

Современная система программирования это достаточ­но сложный комплекс различных программно-технических средств.

Текстовый редактор

Текстовый редактор в системе программирования — это программа, позволяющая создавать, изменять и обрабатывать исходные тексты программ на языках высокого уровня.

В итоге в современных системах программирования текстовый редактор стал важной составной частью, которая не только позволяет пользователю подготавливать исходные тексты программ, но и выполняет все интерфейсные и сервисные функции, предоставляемые пользователю системой программирования.

Транслятор

Отладчик

Отладчик — это программный модуль, который позволяет выполнить основные задачи, связанные с мониторингом процесса выполнения результирующей прикладной программы. Этот процесс называется отладкой и включает в себя следующие основные возможности:

* последовательное пошаговое выполнение результирующей программы на основе шагов по машинным командам или по операторам входного языка;
* выполнение результирующей программы до достижения ею одной из заданных точек останова (адресов останова);
* выполнение результирующей программы до наступления некоторых заданных условий, связанных с данными и адресами, обрабатываемыми этой программой;
* просмотр содержимого областей памяти, занятых командами или данными результирующей программы.

Отладчики в современных системах программирования представляют собой модули с развитым интерфейсом пользователя, работающие непосредственно с текстом и модулями исходной программы. Многие их функции интегрированы с функциями текстовых редакторов исходных текстов, входящих в состав систем программирования.

Библиотеки

Библиотека в программировании (от англ. library) — сборник подпрограмм или объектов, используемых для разработки программного обеспечения (ПО).

Библиотеки подпрограмм составляют существенную часть систем программирования. Наряду с дружественностью пользовательского интерфейса состав доступных библиотек подпрограмм во многом определяет возможности системы программирования и ее позиции на рынке средств разработки программного обеспечения.

С точки зрения системы программирования, библиотеки подпрограмм состоят из двух основных компонентов. Это собственно файл (или множество файлов) библиотеки, содержащий объектный код, и набор файлов описаний функций, подпрограмм, констант и переменных, составляющих библиотеку. Описания оформляются на соответствующем входном языке (например, для языка С или С++ это будет набор заголовочных файлов).

Объектный код библиотеки подключается компоновщиком к результирующей программе при создании исполняемого модуля. По структуре он обычно мало чем отличается от обычных объектных файлов, порождаемых компилятором. Чаще всего система программирования храпит объектный код входящих в ее состав библиотек в некотором упакованном виде.

Динамические библиотеки в отличие от традиционных (статических) библиотек подключаются к программе не в момент её компоновки, а непосредственно в ходе выполнения, как только программа затребовала ту или иную функцию, находящуюся в библиотеке. Преимущества таких библиотек очевидны — они не требуют включать в программу объектный код часто используемых функции, чем существенно сокращают объем кода. Различные программы, выполняемые в ОС, могут пользоваться кодом одной и той же библиотеки, содержащейся в данной ОС.

Широкий набор динамических библиотек поддерживается всеми современными ОС. Как правило, они содержат системные функции ОС и общедоступные функции программного интерфейса (API). Кроме того, многие независимые разработчики предоставляют для различных систем программирования свои динамические библиотеки как отдельные товары на рынке средств разработки прикладных программ.

Статическая библиотека — это коллекция объектных файлов, которые присоединяются к программе во время линковки программы. Таким образом статические библиотеки используются только при создании программы. Потом в работе самой программы они не принимают участие, в отличие от динамических библиотек.

Динамическая библиотека — это созданная специальным образом библиотека, которая присоединяется к результирующей программе в два этапа. Первый этап, это естественно этап компиляции. На этом этапе линковщик встраивает в программу описания требуемых функций и переменных, которые присутствуют в библиотеке. Сами объектные файлы из библиотеки не присоединяются к программе. Присоединение этих объектных файлов (кодов функций) осуществляет системный динамический загрузчик во время запуска программы. Загрузчик проверяет все библиотеки, прилинкованные к программе на наличие требуемых объектных файлов, затем загружает их в память и присоединяет их в копии запущенной программы, находящейся в памяти.

Сложный процесс загрузки динамических библиотек замедляет запуск программы, но у него есть существенный, даже можно сказать неоценимый плюс — если другая запускаемая программа линкована с этой же загруженной динамической библиотекой, то она использует туже копию библиотеки. Это означает, что требуется гораздо меньше памяти для запуска нескольких программ, сами загрузочные файлы меньше по размеру, что экономит место на дисках.

Компоновщик

Компоновщик (или редактор связей) предназначен для связывания между собой объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав системы программирования.

Объектный модуль (также — объектный файл, англ. object file) — файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки исходного кода компилятором. Объектный файл содержит в себе особым образом подготовленный код (часто называемый бинарным), который может быть объединён с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполнимого модуля, либо библиотеки.

Примеры сред программирования:

Microsoft Visual Studio

QT Creator

NetBeans

Xamarin Studio

Eclipse

Embarcadero RAD Studio

MonoDevelop

KDevelop

**1.7. Критерии качества программного обеспечения**

*Качество* (*quality*)ПО — это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей. Качество ПО является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.

Поэтому при описании качества ПО, прежде всего, должны быть фиксированы *критерии* отбора требуемых свойств ПО. В настоящее время *критериями качества ПО* (criteria of software quality) принято считать: функциональность, надежность, легкость применения, эффективность, сопровождаемость, мобильность.

*Функциональность* — это способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей.

*Надежность* (reliability) — это способность ПС безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под *отказом* в ПО понимают проявление в нем ошибки. Таким образом, надежное ПС не исключает наличия в нем ошибок — важно лишь, чтобы эти ошибки при практическом применении проявлялись достаточно редко. Убедиться, что ПО является надежным можно при его тестировании, а также при практическом применении. Таким образом, можно разрабатывать лишь надежные, а не правильные ПО. Степень надежности можно характеризовать вероятностью работы ПО без отказа в течение определенного периода времени. При оценке степени надежности ПС следует также учитывать последствия каждого отказа. Некоторые ошибки в ПС могут вызывать лишь некоторые неудобства при его применении, тогда как другие ошибки могут иметь катастрофические последствия. Поэтому для оценки надежности ПС иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

Основным методом обеспечения надежности программного средства является борьба со сложностью. Известны два общих метода борьбы со сложностью систем:

- обеспечения независимости компонентов системы;

- использование в системах иерархических структур.

Обеспечение независимости компонент означает разбиение системы на такие части, между которыми должны остаться по возможности меньше связей. Одним из воплощений этого метода является модульное программирование. Использование иерархических структур допускает связи только между компонентами, принадлежащими смежным уровням иерархии. Этот метод означает разбиение большой системы на подсистемы путем абстрагирования.

При реализации ПО сначала необходимо обеспечить требуемую функциональность и надежность ПС, а затем уже доводить остальные критерии качества до приемлемого уровня.

*Легкость применения* — это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПО и оценке полученных результатов, а также вызывать положительные эмоции пользователя. Легкость применения определяется составом и качеством пользовательской документации, а также некоторыми свойствами, реализуемыми программным путем (например, *пользовательский интерфейс)*.

*Пользовательский интерфейс* представляет средство взаимодействия пользователя с ПС. При разработке пользовательского интерфейса следует учитывать потребности, опыт и способности пользователя. Поэтому потенциальные пользователи должны быть вовлечены в процесс разработки такого интерфейса.

При разработке интерфейса необходимо соблюдать следующие принципы: базирование на терминах и понятиях, знакомых пользователю; единообразность; возможность пользователю получать справочную информацию как по запросу, так и генерируемую ПО.

В настоящее время широко распространены командные и графические пользовательские интерфейсы.

*Эффективность* — это отношение уровня услуг, предоставляемых ПО пользователю при заданных условиях, к объему используемых ресурсов.

На эффективность ПС влияет выбор способа представления данных и выбор алгоритмов, а также особенности их реализации (включая выбор языка программирования. При этом постоянно приходится разрешать противоречие между *временнόй эффективностью* и *эффективностью по памяти* (*ресурсам*).

Для отыскания критических модулей с точки зрения временнóй эффективности ПО потребуется получить распределение по модулям времени работы ПО путем соответствующих измерений во время выполнения ПО. Это может быть сделано с помощью динамического анализатора (специального программного инструмента), который может определить частоту обращения к каждому модулю в процессе применения ПО.

*Сопровождаемость* — это характеристика ПО, которая позволяет минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей. Обеспечение сопровождаемости ПО сводится к обеспечению изучаемости ПО и к обеспечению его модифицируемости.

*Мобильность* — это способность ПО быть перенесенным из одной среды (окружения) в другую, в частности, с одного компьютера на другой.

Критерии качества являются противоречивыми: хорошее обеспечение одного какого-либо критерия качества может существенно затруднить или сделать невозможным обеспечение некоторых других. Поэтому существенная часть процесса обеспечения качества состоит из поиска приемлемых компромиссов.

**1.8 Кросс-компилятор**

Кросс-компилятор (англ. cross compiler) — компилятор, производящий исполняемый код для платформы, отличной от той, на которой исполняется сам кросс-компилятор. Такой инструмент бывает полезен, когда нужно получить код для платформы, экземпляров которой нет в наличии, или в случаях, когда компиляция на целевой платформе невозможна или нецелесообразна (например, это касается мобильных систем или микроконтроллеров с минимальным объёмом памяти).

Пример кросс-компилятора — MinGW GCC с опцией -mno-cygwin. С этой опцией он может в среде Cygwin создавать код, использующий только «родные» библиотеки Windows.

GCC, набор компиляторов для различных языков программирования, может быть использован для кросс-компиляции.

**Операционные системы как составная часть платформы**

Операционная система (ОС, англ. operating system, OS) — комплекс взаимосвязанных программ, предназначенных для управления ресурсами компьютера и организации взаимодействия с пользователем.

В логической структуре типичной вычислительной системы операционная система занимает положение между устройствами с их микроархитектурой, машинным языком и, возможно, собственными (встроенными) микропрограммами (драйверами) — с одной стороны — и прикладными программами с другой.

Разработчикам программного обеспечения операционная система позволяет абстрагироваться от деталей реализации и функционирования устройств, предоставляя минимально необходимый набор функций (см.: интерфейс программирования приложений).

В большинстве вычислительных систем операционная система является основной, наиболее важной (а иногда и единственной) частью системного программного обеспечения. С 1990-х годов наиболее распространёнными операционными системами являются системы семейства Windows, Unix и UNIX-подобные системы.

История

Предшественником операционных систем следует считать служебные программы (загрузчики и мониторы), а также библиотеки часто используемых подпрограмм, начавшие разрабатываться с появлением универсальных компьютеров 1-го поколения (конец 1940-х годов). Служебные программы минимизировали физические манипуляции оператора с оборудованием, а библиотеки позволяли избежать многократного программирования одних и тех же действий (осуществления операций ввода-вывода, вычисления математических функций и т. п.).

В 1950—1960-х годах сформировались и были реализованы основные идеи, определяющие функциональность ОС: пакетный режим, разделение времени и многозадачность, разделение полномочий, реальный масштаб времени, файловые структуры и файловые системы.

Пакетный режим

Необходимость оптимального использования дорогостоящих вычислительных ресурсов привела к появлению концепции «пакетного режима» исполнения программ. Пакетный режим предполагает наличие очереди программ на исполнение, причём система может обеспечивать загрузку программы с внешних носителей данных в оперативную память, не дожидаясь завершения исполнения предыдущей программы, что позволяет избежать простоя процессора.

Разделение времени и многозадачность

Уже пакетный режим в своём развитом варианте требует разделения процессорного времени между выполнением нескольких программ.

Необходимость в разделении времени (многозадачности, мультипрограммировании) проявилась ещё сильнее при распространении в качестве устройств ввода-вывода телетайпов (а позднее, терминалов с электронно-лучевыми дисплеями) (1960-е годы). Поскольку скорость клавиатурного ввода (и даже чтения с экрана) данных оператором намного ниже, чем скорость обработки этих данных компьютером, использование компьютера в «монопольном» режиме (с одним оператором) могло привести к простою дорогостоящих вычислительных ресурсов.

Разделение времени позволило создать «многопользовательские» системы, в которых один (как правило) центральный процессор и блок оперативной памяти соединялся с многочисленными терминалами. При этом часть задач (таких как ввод или редактирование данных оператором) могла исполняться в режиме диалога, а другие задачи (такие как массивные вычисления) — в пакетном режиме.

Разделение полномочий

Распространение многопользовательских систем потребовало решения задачи разделения полномочий, позволяющей избежать возможности изменения исполняемой программы или данных одной программы в памяти компьютера другой программой (намеренно или по ошибке), а также изменения самой системы прикладной программой.

Реализация разделения полномочий в операционных системах была поддержана разработчиками процессоров, предложивших архитектуры с двумя режимами работы процессора — «реальным» (в котором исполняемой программе доступно всё адресное пространство компьютера) и «защищённым» (в котором доступность адресного пространства ограничена диапазоном, выделенным при запуске программы на исполнение).

Масштаб реального времени

Применение универсальных компьютеров для управления производственными процессами потребовало реализации «масштаба реального времени» («реального времени») — синхронизации исполнения программ с внешними физическими процессами.

Включение функции масштаба реального времени позволило создавать решения, одновременно обслуживающие производственные процессы и решающие другие задачи (в пакетном режиме и/или в режиме разделения времени).

Файловые системы и структуры

Постепенная замена носителей с последовательным доступом (перфолент, перфокарт и магнитных лент) накопителями произвольного доступа (на магнитных дисках).

Файловая система — способ хранения данных на внешних запоминающих устройствах.

Функции ОС

Основные функции:

Исполнение запросов программ (ввод и вывод данных, запуск и остановка других программ, выделение и освобождение дополнительной памяти и др.).

Загрузка программ в оперативную память и их выполнение.

Стандартизованный доступ к периферийным устройствам (устройства ввода-вывода).

Управление оперативной памятью (распределение между процессами, организация виртуальной памяти).

Управление доступом к данным на энергонезависимых носителях (таких как жёсткий диск, оптические диски и др.), организованным в той или иной файловой системе.

Обеспечение пользовательского интерфейса.

Сохранение информации об ошибках системы.

Дополнительные функции:

Параллельное или псевдопараллельное выполнение задач (многозадачность).

Эффективное распределение ресурсов вычислительной системы между процессами.

Разграничение доступа различных процессов к ресурсам.

Организация надёжных вычислений (невозможности одного вычислительного процесса намеренно или по ошибке повлиять на вычисления в другом процессе), основана на разграничении доступа к ресурсам.

Взаимодействие между процессами: обмен данными, взаимная синхронизация.

Защита самой системы, а также пользовательских данных и программ от действий пользователей (злонамеренных или по незнанию) или приложений.

Многопользовательский режим работы и разграничение прав доступа (см.: аутентификация, авторизация).

Понятие ОС

Существуют две группы определений операционной системы: «набор программ, управляющих оборудованием» и «набор программ, управляющих другими программами». Обе они имеют свой точный технический смысл, который связан с вопросом, в каких случаях требуется операционная система.

Есть приложения вычислительной техники, для которых операционные системы излишни. Например, встроенные микрокомпьютеры, содержащиеся во многих бытовых приборах, автомобилях (иногда по десятку в каждом), простейших сотовых телефонах, постоянно исполняют лишь одну программу, запускающуюся по включении. Многие простые игровые приставки — также представляющие собой специализированные микрокомпьютеры — могут обходиться без операционной системы, запуская при включении программу, записанную на вставленном в устройство «картридже» или компакт-диске.

Операционные системы нужны:

если нужен универсальный механизм сохранения данных;

для предоставления программам системных библиотек с часто используемыми подпрограммами;

для распределения полномочий;

необходима возможность имитации «одновременного» исполнения нескольких программ на одном компьютере;

для управления процессами выполнения отдельных программ.

Таким образом, современные универсальные операционные системы можно охарактеризовать, прежде всего, как:

использующие файловые системы (с универсальным механизмом доступа к данным),

многопользовательские (с разделением полномочий),

многозадачные (с разделением времени).

Многозадачность и распределение полномочий требуют определённой иерархии привилегий компонентов в самой операционной системе. В составе операционной системы различают три группы компонентов:

ядро, содержащее планировщик; драйверы устройств, непосредственно управляющие оборудованием; сетевая подсистема, файловая система;

системные библиотеки;

оболочка с утилитами.

Большинство программ, как системных (входящих в операционную систему), так и прикладных, исполняются в непривилегированном («пользовательском») режиме работы процессора и получают доступ к оборудованию (и, при необходимости, к другим ресурсам ядра, а также ресурсам иных программ) только посредством системных вызовов. Ядро исполняется в привилегированном режиме: именно в этом смысле говорят, что система (точнее, её ядро) управляет оборудованием.

В определении состава операционной системы значение имеет критерий операциональной целостности (замкнутости): система должна позволять полноценно использовать (включая модификацию) свои компоненты. Поэтому в полный состав операционной системы включают и набор инструментальных средств (от текстовых редакторов до компиляторов, отладчиков и компоновщиков).

Ядро

Ядро — центральная часть операционной системы, управляющая выполнением процессов, ресурсами вычислительной системы и предоставляющая процессам координированный доступ к этим ресурсам. Основными ресурсами являются процессорное время, память и устройства ввода-вывода. Доступ к файловой системе и сетевое взаимодействие также могут быть реализованы на уровне ядра.

Как основополагающий элемент операционной системы, ядро представляет собой наиболее низкий уровень абстракции для доступа приложений к ресурсам вычислительной системы, необходимым для их работы. Как правило, ядро предоставляет такой доступ исполняемым процессам соответствующих приложений за счёт использования механизмов межпроцессного взаимодействия и обращения приложений к системным вызовам ОС.

Описанная задача может различаться в зависимости от типа архитектуры ядра и способа её реализации.

Объекты ядра ОС:

Процессы - совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.,

Файлы - именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.,

События – это сообщение, посланное объектом, чтобы сообщить о совершении действия. Это действие может быть вызвано пользовательским взаимодействием, например нажатием кнопки, или какой-то другой программной логикой, например изменением значения свойства.

Потоки наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют последовательность инструкций процесса (его код) и его контекст — значения переменных (регистров процессора и стека вызовов), которые они имеют в любой момент времени.,

Семафоры - примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующейся. Служит для построения более сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.,

Мьютексы – примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода. Классический мьютекс отличается от двоичного семафора наличием эксклюзивного владельца, который и должен его освобождать (т.е. переводить в незаблокированное состояние).

Каналы – один из методов межпроцессного взаимодействия, расширение понятия конвейера в Unix и подобных ОС. Именованный канал позволяет различным процессам обмениваться данными, даже если программы, выполняющиеся в этих процессах, изначально не были написаны для взаимодействия с другими программами.

файлы, проецируемые в память – это способ работы с файлами в некоторых операционных системах, при котором всему файлу или некоторой непрерывной его части ставится в соответствие определённый участок памяти (диапазон адресов оперативной памяти). При этом чтение данных из этих адресов фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл. Отображать на память часто можно не только обычные файлы, но и файлы устройств.

Существующие операционные системы

1. UNIX, стандартизация операционных систем и POSIX

Задуманная и реализованная в 1969 году Кеном Томпсоном при участии нескольких коллег (включая Денниса Ритчи и Брайана Кернигана), операционная система UNIX (первоначально UNICS, что обыгрывало название Multics) вобрала в себя многие черты более ранних систем, но обладала целым рядом свойств, отличающих её от большинства предшественниц:

простая метафорика (два ключевых понятия: вычислительный процесс и файл);

компонентная архитектура: принцип «одна программа — одна функция» плюс мощные средства связывания различных программ для решения возникающих задач («оболочка»);

минимизация ядра (кода, выполняющегося в «реальном» (привилегированном) режиме процессора) и количества системных вызовов;

независимость от аппаратной архитектуры и реализация на машиннонезависимом языке программирования (язык программирования Си стал побочным продуктом разработки UNIX);

унификация файлов.

UNIX, благодаря своему удобству прежде всего в качестве инструментальной среды (среды разработки), обрела популярность сначала в университетах, а затем и в отрасли, получившей прототип единой операционной системы, которая могла использоваться на самых разных вычислительных системах и, более того, могла быть быстро и с минимальными усилиями перенесена на любую вновь разработанную аппаратную архитектуру.

Благодаря конкурентности реализаций архитектура UNIX стала вначале фактическим отраслевым стандартом, а затем обрела статус и стандарта юридического — ISO/IEC 9945[1] (POSIX).

Только системы, отвечающие спецификации Single UNIX Specification, имеют право носить имя UNIX. К таким системам относятся AIX, HP-UX, IRIX, Mac OS X, SCO OpenServer, Solaris, Tru64 и z/OS.

Самым заметным эффектом существования этого стандарта стало эффективное разворачивание Интернета в 1990-х годах.

1. Windows
2. Web-браузеры