Глава 13: Основы вычислений

From SWEBOK

Содержание

АКРОНИМЫ

АОП	Аспектно-ориентированное программирование
АЛУ	Арифметико-логический блок
API	Интерфейс прикладного программирования
банкомат	асинхронный режим передачи
Б/С	Браузер-сервер
СЕРТИФИКАЦИЯ	группа реагирования на компьютерные чрезвычайные ситуации
КОТС	Коммерческий готовый

CRUD	Создать, прочитать, обновить, удалить
К/С	Клиент-сервер
КС	Информатика
СУБД	Система управления базами данных
ФПУ	Единица с плавающей запятой
ввод/вывод	Ввод и вывод
ЭТО	Архитектура набора инструкций
ИСО	Международная организация по стандартизации
Интернет-провайдер	Интернет-провайдер
локальная сеть	Локальная сеть
МУКС	Мультиплексор
сетевая карта	Сетевая карта
ООП	Объектно-ориентированного программирования
Операционные системы	Операционная система
ОСИ	Взаимодействие открытых систем
ПК	Персональный компьютер
КПК	Персональный цифровой помощник
ГЧП	Протокол точка-точка
RFID	Определение радиочастоты
БАРАН	Оперативная память
ПЗУ	Только для чтения памяти
SCSI	Системный интерфейс малого компьютера
SQL	Язык структурированных запросов
TCP	Протокол управления транспортом

TCP	Протокол управления транспортом
UDP	Протокол пользовательских датаграмм
VPN	Виртуальная частная сеть
глобальная сеть	Глобальная сеть

ВВЕДЕНИЕ

Объем области знаний Computing Foundations (KA) охватывает среду разработки и эксплуатации, в которой развивается и выполняется программное обеспечение. Поскольку никакое программное обеспечение не может существовать в вакууме или работать без компьютера, ядром такой среды является компьютер и его различные компоненты. Знания о компьютере и лежащих в его основе принципах аппаратного и программного обеспечения служат основой, на которой базируется программная инженерия. Таким образом, все инженеры-программисты должны хорошо понимать Computing Foundations KA.

Общепризнанно, что программная инженерия строится на основе компьютерных наук. Например, в «Программной инженерии 2004: Руководящие принципы учебных программ для программ бакалавриата по программной инженерии» [1] четко сказано: «Один особенно важный аспект заключается в том, что *программная инженерия основывается на компьютерных науках* и математике» (курсив добавлен).

Стив Токи написал в своей книге « *Возвращение к программному обеспечению»* :

И информатика, и программная инженерия имеют дело с компьютерами, вычислениями и программным обеспечением. Наука о вычислениях как совокупность знаний лежит в основе обоих. … Разработка программного обеспечения связана с применением компьютеров, вычислений и программного обеспечения для практических целей, в частности, с проектированием, созданием и эксплуатацией эффективных и экономичных программных систем. Таким образом, в основе разработки программного обеспечения лежит понимание информатики.

Хотя мало кто будет отрицать роль компьютерных наук в развитии разработки программного обеспечения как дисциплины и как совокупности знаний, важность информатики для разработки программного обеспечения невозможно переоценить; таким образом, этот Computing Foundations KA пишется.

Большинство тем, обсуждаемых в Computing Foundations KA, также являются темами обсуждения на базовых курсах программ бакалавриата и магистратуры по информатике. Такие курсы включают программирование, структуру данных, алгоритмы, организацию компьютеров, операционные системы, компиляторы, базы данных, сети, распределенные системы и так далее. Таким образом, при разбивке по темам может возникнуть соблазн разложить KA по основам вычислений в соответствии с этими часто встречающимися подразделениями в соответствующих курсах.

Однако чисто курсовое разделение тем имеет серьезные недостатки. Во-первых, не все курсы информатики связаны или одинаково важны для разработки программного обеспечения. Таким образом, некоторые темы, которые в противном случае рассматривались бы в курсе компьютерных наук, не рассматриваются в этом KA. Например, компьютерная графика, хотя и является важным курсом программы получения степени в области компьютерных наук, не включена в этот KA.

Во-вторых, некоторые темы, обсуждаемые в этом руководстве, не существуют как отдельные курсы в программах бакалавриата или магистратуры по информатике. Следовательно, такие темы не могут быть должным образом освещены в разбивке по курсам. Например, абстракция — это тема, включенная в несколько различных курсов информатики; неясно, к какой абстракции курса следует относиться при разбивке тем на основе курса.

Computing Foundations KA разделен на семнадцать различных тем. Непосредственная полезность темы для разработчиков программного обеспечения является критерием, используемым для выбора тем для включения в этот KA (см. рис. 13.1). Преимущество этой разбивки по темам заключается в том, что она

основана на вере в то, что «Основы вычислений» — если их нужно твердо усвоить — следует рассматривать как набор логически связанных тем, лежащих в основе разработки программного обеспечения в целом и разработки программного обеспечения в частности.

КА «Основы вычислений» тесно связаны с КА «Разработка программного обеспечения», «Создание программного обеспечения», «Тестирование программного обеспечения», «Сопровождение программного обеспечения», «Качество программного обеспечения» и «Основы математики».

РАЗБИВКА ТЕМ ПО ОСНОВАМ ВЫЧИСЛЕНИЙ

Разбивка тем для Computing Foundations KA показана на рис. 13.1.

1 Методы решения проблем

[2 , c3.2, c4] [3 , c5]

Введенные здесь понятия, понятия и терминология составляют основу для понимания роли и объема методов решения проблем.

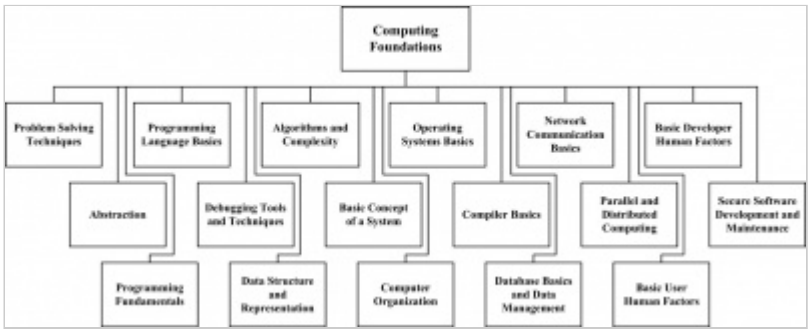

Рисунок 13.1: Разбивка тем для Computing Foundations KA

1.1 Определение решения проблем

Решение проблем относится к мышлению и действиям, направленным на поиск ответа или решения проблемы. Есть много способов подойти к проблеме, и каждый из них использует разные инструменты и использует разные процессы. Эти различные подходы к проблемам постепенно расширяются и определяют себя и, в конце концов, приводят к появлению различных дисциплин. Например, разработка программного обеспечения фокусируется на решении проблем с использованием компьютеров и программного обеспечения.

Хотя разные проблемы требуют разных решений и могут требовать разных инструментов и процессов, методология и методы, используемые при решении проблем, действительно следуют некоторым рекомендациям и часто могут быть обобщены как методы решения проблем. Например, общее руководство для решения общей инженерной задачи состоит в том, чтобы использовать трехэтапный процесс, описанный ниже [2*].

- Сформулируйте реальную проблему.
- Проанализируйте проблему.
- Разработайте стратегию поиска решения.

1.2 Формулировка реальной проблемы

Жерар Воланд пишет: «Важно признать, что конкретная проблема должна быть сформулирована, если нужно разработать конкретное решение» [2*]. Эта формулировка называется постановкой задачи, которая явно указывает, какова проблема и желаемый результат.

Хотя не существует универсального способа постановки проблемы, в целом проблема должна быть выражена таким образом, чтобы облегчить выработку решений. Некоторые общие методы, помогающие сформулировать реальную проблему, включают переформулирование утверждения, определение источника и причины, пересмотр утверждения, анализ настоящего и желаемого состояния и использование подхода «свежим взглядом».

1.3 Анализ проблемы

Как только формулировка проблемы доступна, следующим шагом будет анализ постановки проблемы или ситуации, чтобы помочь структурировать наш поиск решения. Четыре типа анализа включают *ситуационный анализ* , при котором в первую очередь определяются наиболее неотложные или критические аспекты ситуации; *анализ проблемы* , при котором необходимо определить причину проблемы; *анализ решения* , при котором должны быть определены действия, необходимые для

исправления проблемы или устранения ее причины; и *анализ потенциальных проблем* , в ходе которого должны быть определены действия, необходимые для предотвращения повторного возникновения проблемы или развития новых проблем.

1.4 Разработка стратегии поиска решения

Как только анализ проблемы завершен, мы можем сосредоточиться на структурировании стратегии поиска решения. Чтобы найти «лучшее» решение (здесь «лучшее» может означать разные вещи для разных людей, например, быстрее, дешевле, удобнее, разные возможности и т. д.), нам нужно устранить пути, которые не ведут к жизнеспособным решения, проектировать задачи таким образом, чтобы обеспечить максимальное руководство в поиске решения, и использовать различные атрибуты конечного состояния решения, чтобы направлять наш выбор в процессе решения проблемы.

1.5 Решение проблем с помощью программ

Уникальность компьютерного программного обеспечения придает решению проблем оттенок, отличный от решения общих инженерных задач. Чтобы решить задачу с помощью компьютеров, мы должны ответить на следующие вопросы.

- Как понять, что делать компьютеру?
- Как преобразовать постановку задачи в алгоритм?
- Как преобразовать алгоритм в машинные инструкции?

Первая задача при решении задачи с помощью компьютера — определить, что следует сказать компьютеру. Может быть много способов рассказать историю, но все они должны быть с точки зрения компьютера, чтобы компьютер мог в конечном итоге решить проблему. В общем случае проблема должна быть выражена таким образом, чтобы облегчить разработку алгоритмов и структур данных для ее решения.

Результатом первой задачи является постановка задачи. Следующим шагом является преобразование постановки задачи в алгоритмы, решающие проблему. Как только алгоритм найден, последний шаг преобразует алгоритм в машинные инструкции, которые формируют окончательное решение: программное обеспечение, которое решает проблему.

Абстрактно говоря, решение задач с помощью компьютера можно рассматривать как процесс преобразования проблемы, другими словами, пошаговое преобразование постановки задачи в решение проблемы. Для дисциплины разработки программного обеспечения конечной целью решения проблем является преобразование проблемы, выраженной на естественном языке, в электроны, бегущие по цепи. В целом, это преобразование можно разбить на три этапа:

- а) Разработка алгоритмов из постановки задачи.
- б) Применение алгоритмов к задаче.
- в) Преобразование алгоритмов в программный код.

Преобразование постановки задачи в алгоритмы, а алгоритмов в программные коды обычно следует за «пошаговым уточнением» (также известным как систематическая декомпозиция), при котором мы начинаем с постановки задачи, переписываем ее как задачу и рекурсивно разбиваем задачу на несколько более простых подзадач. пока задача не станет настолько простой, что ее решения будут очевидны. Существует три основных способа декомпозиции: последовательный, условный и итеративный.

2 Абстракция

[3 , c5.2–5.4]

Абстракция — незаменимая техника, связанная с решением проблем. Это относится как к процессу, так и к результату обобщения путем сокращения информации о понятии, проблеме или наблюдаемом явлении, чтобы можно было сосредоточиться на «общей картине». Один из наиболее важных навыков в любом инженерном начинании — это правильное построение уровней абстракции.

«Посредством абстракции, — по словам Воланда, — мы рассматриваем проблему и возможные пути ее решения с более высокого уровня концептуального понимания. В результате мы можем лучше подготовиться к распознаванию возможных взаимосвязей между различными аспектами проблемы и

тем самым генерировать более творческие дизайнерские решения» [2*]. Это особенно верно в области информатики в целом (например, аппаратное и программное обеспечение) и в разработке программного обеспечения в частности (структура данных и поток данных и т. д.).

2.1 Уровни абстракции

При абстрагировании мы концентрируемся на одном «уровне» общей картины за раз с уверенностью, что затем сможем эффективно соединиться с уровнями выше и ниже. Хотя мы фокусируемся на одном уровне, абстракция не означает, что мы ничего не знаем о соседних уровнях. Уровни абстракции не обязательно соответствуют дискретным компонентам в реальности или предметной области, а соответствуют четко определенным стандартным интерфейсам, таким как программные API. Преимущества, предоставляемые стандартными интерфейсами, включают переносимость, более простую интеграцию программного и аппаратного обеспечения и более широкое использование.

2.2 Инкапсуляция

Инкапсуляция — это механизм, используемый для реализации абстракции. Когда мы имеем дело с одним уровнем абстракции, инкапсулируется информация об уровнях ниже и выше этого уровня. Эта информация может быть концепцией, проблемой или наблюдаемым явлением; или это могут быть допустимые операции над этими соответствующими объектами. Инкапсуляция обычно сопровождается некоторой степенью сокрытия информации, при которой некоторые или все основные детали скрыты от уровня выше интерфейса, обеспечиваемого абстракцией. Для объекта сокрытие информации означает, что нам не нужно знать подробности того, как объект представлен или как реализованы операции над этими объектами.

2.3 Иерархия

Когда мы используем абстракцию при формулировании и решении проблемы, мы можем использовать разные абстракции в разное время — другими словами, мы работаем на разных уровнях абстракции в зависимости от ситуации. В большинстве случаев эти различные уровни абстракции организованы в виде иерархии. Существует множество способов структурирования конкретной иерархии, и критерии, используемые при определении конкретного содержания каждого уровня в иерархии, различаются в зависимости от лиц, выполняющих работу.

Иногда иерархия абстракции является последовательной, что означает, что каждый уровень имеет один и только один предшествующий (нижний) уровень и один и только один последующий (верхний) уровень, за исключением самого верхнего уровня (у которого нет последующего) и самого нижнего уровня (у которого нет предшественника). Однако иногда иерархия организована в виде древовидной структуры, что означает, что каждый уровень может иметь более одного уровня-предшественника, но только один уровень-последующий. Иногда иерархия может иметь структуру «многие ко многим», в которой каждый уровень может иметь несколько предшественников и последователей. В иерархии никогда не должно быть петли.

Иерархия часто формируется естественным образом при декомпозиции задач. Часто анализ задач можно разложить по иерархическому принципу, начиная с более крупных задач и целей организации и разбивая каждую из них на более мелкие подзадачи, которые можно снова подразделить. структура задач-подзадач.

2.4 Альтернативные абстракции

Иногда полезно иметь несколько альтернативных абстракций для одной и той же проблемы, чтобы иметь в виду разные точки зрения. Например, у нас может быть диаграмма классов, диаграмма состояний и диаграмма последовательности для одного и того же программного обеспечения на одном уровне абстракции. Эти альтернативные абстракции не образуют иерархию, а скорее дополняют друг друга, помогая понять проблему и ее решение. Хотя это и полезно, как всегда сложно синхронизировать альтернативные абстракции.

3 Основы программирования

[3 , с6–19]

Программирование состоит из методологий или действий по созданию компьютерных программ, выполняющих желаемую функцию. Это незаменимая часть в разработке программного обеспечения. В общем, программирование можно рассматривать как процесс проектирования, написания, тестирования, отладки и поддержки исходного кода. Этот исходный код написан на языке программирования.

Процесс написания исходного кода часто требует опыта во многих различных предметных областях, включая знание предметной области, соответствующих структур данных, специализированных алгоритмов, различных языковых конструкций, хороших методов программирования и разработки программного обеспечения.

3.1 Процесс программирования

Программирование включает в себя проектирование, написание, тестирование, отладку и обслуживание. *Дизайн* — это концепция или изобретение схемы превращения требований заказчика к компьютерному программному обеспечению в действующее программное обеспечение. Это действие, которое связывает требования приложения с кодированием и отладкой. *Написание* — это фактическое кодирование дизайна на соответствующем языке программирования. *Тестирование* — это деятельность по проверке того, что код, который вы пишете, действительно делает то, что он должен делать. *Отладка* — это деятельность по поиску и исправлению ошибок (ошибок) в исходном коде (или дизайне). *Обслуживание*это деятельность по обновлению, исправлению и улучшению существующих программ. Каждое из этих действий представляет собой огромную тему и часто требует объяснения всего КА в *Руководстве SWEBOK* и во многих книгах.

3.2 Парадигмы программирования

Программирование очень творческое и, следовательно, несколько личное. Разные люди часто пишут разные программы для одних и тех же требований. Это разнообразие программирования вызывает большие трудности при создании и обслуживании большого сложного программного обеспечения. За прошедшие годы были разработаны различные парадигмы программирования, чтобы внести некоторую стандартизацию в эту весьма творческую и личную деятельность. Когда кто-то программирует, он или она может использовать одну из нескольких парадигм программирования для написания кода. Основные типы парадигм программирования обсуждаются ниже.

- *Неструктурированное программирование* : в неструктурированном программировании программист следует своей догадке, чтобы писать код любым удобным для него способом.

нравится, пока функция работает. Часто практика заключается в написании кода для выполнения конкретной утилиты без учета всего остального. Программы, написанные таким образом, не имеют определенной структуры — отсюда и название «неструктурированное программирование». Неструктурированное программирование также иногда называют специальным программированием.

- *Структурированное/процедурное/императивное программирование* . Отличительной чертой структурированного программирования является использование четко определенных управляющих структур, включая процедуры (и/или функции), где каждая процедура (или функция) выполняет определенную задачу. Между процедурами существуют интерфейсы для облегчения корректных и плавных вызовов программ. При структурированном программировании программисты часто следуют установленным протоколам и эмпирическим правилам при написании кода. Этих протоколов и правил может быть множество, и они охватывают почти весь спектр программирования — от самых простых вопросов (например, как назвать переменные, функции, процедуры и т. д.) до более сложных вопросов (например, как структурировать интерфейс, как обрабатывать исключения и так далее).
- *Объектно-ориентированное программирование* : в то время как процедурное программирование организует программы вокруг процедур, объектно-ориентированное программирование (ООП)

организовать программу вокруг объектов, которые являются абстрактными структурами данных, которые объединяют как данные, так и методы, используемые для доступа или управления данными. Основные особенности ООП заключаются в том, что создаются объекты, представляющие различные абстрактные и конкретные объекты, и эти объекты взаимодействуют друг с другом для коллективного выполнения желаемых функций.

- *Аспектно-ориентированное программирование* : Аспектно-ориентированное программирование (АОП) — это парадигма программирования, построенная на основе ООП. АОП стремится изолировать второстепенные или вспомогательные функции от бизнес-логики основной программы, сосредоточив внимание на сечениях (проблемах) объектов. Основной мотивацией для АОП является устранение путаницы и рассеяния объектов, связанных с ООП, при котором взаимодействия между объектами становятся очень сложными. Суть АОП заключается в сильно подчеркнутом разделении задач, которое разделяет неосновные функциональные задачи или логику на различные аспекты.
- *Функциональное программирование* . Хотя функциональное программирование менее популярно, оно столь же жизнеспособно, как и другие парадигмы в решении задач программирования. В функциональном программировании все вычисления рассматриваются как вычисление математических функций. В отличие от императивного программирования, которое делает упор на изменение состояния, функциональное программирование делает упор на применение функций, избегает состояния и изменяемых данных и обеспечивает ссылочную прозрачность.

4 Основы языка программирования

[4 , c6]

Использование компьютеров для решения задач предполагает программирование, т. е. написание и систематизацию инструкций, сообщающих компьютеру, что делать на каждом этапе. Программы должны быть написаны на каком-то языке программирования, с помощью которого и посредством которого мы описываем необходимые вычисления. Другими словами, мы используем средства, предоставляемые языком программирования, для описания проблем, разработки алгоритмов и рассуждений о решениях проблем. Чтобы написать любую программу, нужно знать хотя бы один язык программирования.

4.1 Обзор языка программирования

Язык программирования предназначен для выражения вычислений, которые может выполнять компьютер. В практическом смысле язык программирования представляет собой нотацию для написания программ и, следовательно, должен быть способен выражать большинство структур данных и алгоритмов. Некоторые, но не все, ограничивают термин «язык программирования» теми языками, которые могут выражать все возможные алгоритмы.

Не все языки имеют одинаковое значение и популярность. Наиболее популярные из них часто определяются документом спецификации, установленным известной и уважаемой организацией. Например, язык программирования C определяется стандартом ISO под названием ISO/IEC 9899. Другие языки, такие как Perl и Python, не пользуются таким подходом и часто имеют доминирующую реализацию, которая используется в качестве эталона.

4.2 Синтаксис и семантика языков программирования

Так же, как и естественные языки, многие языки программирования имеют письменную спецификацию своего синтаксиса (формы) и семантики (значения). Такие спецификации включают, например, особые требования к определению переменных и констант (другими словами, к объявлению и типам) и требования к формату самих инструкций.

В общем, язык программирования поддерживает такие конструкции, как переменные, типы данных, константы, литералы, операторы присваивания, операторы управления, процедуры, функции и комментарии. Синтаксис и семантика каждой конструкции должны быть четко определены.

4.3 Языки программирования низкого уровня

Языки программирования можно разделить на два класса: языки низкого уровня и языки высокого уровня. Языки низкого уровня могут быть поняты компьютером без помощи или с минимальной помощью и обычно включают машинные языки и языки ассемблера. В машинном языке для представления инструкций и переменных используются единицы и нули, и он напрямую понятен компьютеру. Язык ассемблера содержит те же инструкции, что и машинный язык, но инструкции и переменные имеют символические имена, которые людям легче запомнить.

Языки ассемблера не могут быть непосредственно поняты компьютером и должны быть переведены на машинный язык служебной программой, называемой ассемблером. Часто существует соответствие между инструкциями языка ассемблера и машинного языка, и перевод из ассемблерного кода в машинный код является прямым. Например, «добавить r1, r2, r3» — это ассемблерная инструкция для сложения содержимого регистров r2 и r3 и сохранения суммы в регистре r1. Эту инструкцию можно легко перевести в машинный код «0001 0001 0010 0011». (Предположим, что код операции добавления равен 0001, см. рис. 13.2). Одной общей чертой, разделяемой этими двумя типами языков, является их тесная связь со спецификой типа компьютера или архитектуры набора инструкций (ISA).

4.4 Языки программирования высокого уровня

add	r1,	r2,	r3
0001	0001	0010	0011

Рисунок 13.2: Преобразование ассемблера в двоичный файл

Язык программирования высокого уровня имеет сильную абстракцию от деталей ISA компьютера. По сравнению с низкоуровневыми языками программирования, он часто использует элементы естественного языка и поэтому его намного легче понять людям. Такие языки допускают символическое наименование переменных, обеспечивают выразительность и позволяют абстрагироваться от базового оборудования. Например, хотя каждый микропроцессор имеет свой собственный ISA, код, написанный на языке программирования высокого уровня, обычно переносим между множеством различных аппаратных платформ. По этим причинам большинство программистов используют языки программирования высокого уровня, и большая часть программного обеспечения написана на них. Примеры языков программирования высокого уровня включают C, C++, C# и Java.

4.5 Декларативные и императивные языки программирования

Большинство языков программирования (высокоуровневых или низкоуровневых) позволяют программистам указывать отдельные инструкции, которые должен выполнять компьютер. Такие языки программирования называются императивными языками программирования, потому что нужно четко указывать компьютеру каждый шаг. Но некоторые языки программирования позволяют программистам только описывать выполняемую функцию, не указывая точные последовательности выполняемых инструкций. Такие языки программирования называются декларативными языками программирования. Декларативные языки — это языки высокого уровня. Фактическая реализация вычислений, написанных на таком языке, скрыта от программистов и, следовательно, их не волнует.

Важно отметить, что декларативное программирование описывает только *то* , что программа должна выполнить, не описывая *, как* это сделать. По этой причине многие считают, что декларативное программирование облегчает разработку программного обеспечения. К декларативным языкам программирования относятся Lisp (также функциональный язык программирования) и Prolog, а к императивным языкам программирования относятся C, C++ и JAVA.

5 инструментов и методов отладки

[3 , c23]

После того, как программа закодирована и скомпилирована (компиляция будет обсуждаться в разделе 10), следующим шагом будет отладка, представляющая собой методический процесс поиска и уменьшения количества ошибок или ошибок в программе. Цель отладки — выяснить, почему программа не работает или выдает неправильный результат или вывод. За исключением очень простых программ, всегда необходима отладка.

5.1 Типы ошибок

Когда программа не работает, это часто происходит из-за того, что программа содержит ошибки или ошибки, которые могут быть либо синтаксическими ошибками, либо логическими ошибками, либо ошибками данных. Логические ошибки и ошибки данных также известны как две категории «ошибок» в терминологии разработки программного обеспечения (см. раздел 1.1 «Терминология, связанная с тестированием» в KA по тестированию программного обеспечения).

- *Синтаксические ошибки* — это просто любые ошибки, которые мешают транслятору (компилятору/интерпретатору) успешно проанализировать инструкцию. Каждый оператор в

программе должен поддаваться анализу, прежде чем его значение можно будет понять и интерпретировать (и, следовательно, выполнить). В языках программирования высокого уровня синтаксические ошибки обнаруживаются во время компиляции или трансляции с языка высокого уровня в машинный код. Например, в языке программирования C/C++ оператор «123=constant;» содержит синтаксическую ошибку, которая будет обнаружена компилятором во время компиляции.

- *Логические ошибки* — это семантические ошибки, которые приводят к неправильным вычислениям или поведению программы. Ваша программа законна, но неверна! Таким образом, результаты не соответствуют постановке задачи или ожиданиям пользователей. Например, в языке программирования C/C++ встроенная функция «int f(int x) {return f(x-1);}» для вычисления факториала x! является законным, но логически неверным. Этот тип ошибок не может быть обнаружен компилятором во время компиляции и часто обнаруживается при отслеживании выполнения программы (современные статические средства проверки действительно выявляют некоторые из этих ошибок. Однако остается факт, что в целом они не поддаются машинной проверке).

- *Ошибки данных* — это ошибки ввода, которые приводят либо к вводу данных, отличных от ожидаемых программой, либо к обработке неправильных данных.

5.2 Методы отладки

Отладка включает в себя множество действий и может быть статической, динамической или посмертной. *Статическая отладка* обычно принимает форму проверки кода, тогда как *динамическая отладка* обычно принимает форму отслеживания и тесно связана с тестированием. *Посмертная отладка* — это акт отладки дампа ядра (дампа памяти) процесса. Дампы ядра часто генерируются после завершения процесса из-за необработанного исключения. Все три метода используются на различных этапах разработки программы.

Основным действием динамической отладки является трассировка, то есть выполнение программы по частям, проверка содержимого регистров и памяти для проверки результатов на каждом шаге. Существует три способа трассировки программы.

- *Пошаговый* : выполняйте по одной инструкции за раз, чтобы убедиться, что каждая инструкция выполняется правильно. Этот метод утомителен, но

полезно для проверки каждого шага программы.

- *Точки останова* : говорят программе прекратить выполнение, когда она достигает определенной инструкции. Эта техника позволяет быстро выполнить

выбранные последовательности кода, чтобы получить общее представление о поведении при выполнении.

- *Точки наблюдения* : сообщите программе, чтобы она останавливалась, когда изменяется регистр или ячейка памяти или когда она равна определенному значению. Этот метод полезен, когда неизвестно, где и когда значение было изменено, и когда это изменение значения, вероятно, вызывает ошибку.

5.3 Инструменты отладки

Отладка может быть сложной, трудной и утомительной. Как и в программировании, отладка также требует большого творчества (иногда даже более творческого, чем программирование). Таким образом, некоторая помощь от инструментов в порядке. Для динамической отладки широко используются *отладчики , которые позволяют программисту отслеживать выполнение программы, останавливать выполнение, перезапускать выполнение, устанавливать точки останова, изменять значения в памяти и даже, в некоторых случаях, возвращаться назад во времени.*

Для статической отладки существует множество *инструментов статического анализа кода* , которые ищут определенный набор известных проблем в исходном коде. Как коммерческие, так и бесплатные инструменты существуют на разных языках. Эти инструменты могут быть чрезвычайно полезны при

проверке очень больших деревьев исходного кода, где нецелесообразно выполнять обход кода. Программа UNIX *lint* является ранним примером.

6 Структура данных и представление

[5 , c2.1–2.6]

Программы работают с данными. Но данные должны быть выражены и организованы в компьютерах, прежде чем они будут обработаны программами. Эта организация и представление данных для использования в программах является предметом структуры и представления данных. Проще говоря, структура данных пытается хранить и организовывать данные в компьютере таким образом, чтобы эти данные можно было эффективно использовать. Существует много типов структур данных, и каждый тип структуры подходит для определенных приложений. Например, деревья B/B+ хорошо подходят для реализации массивных файловых систем и баз данных.

6.1 Обзор структуры данных

Структуры данных — это компьютерные представления данных. Структуры данных используются почти во всех программах. В некотором смысле никакая осмысленная программа не может быть построена без использования какой-либо структуры данных. Некоторые методы проектирования и языки программирования даже организуют всю программную систему вокруг структур данных. По сути, структуры данных — это абстракции, определенные для набора данных и связанных с ним операций.

Часто структуры данных разрабатываются для повышения эффективности программы или алгоритма. Примеры таких структур данных включают стеки, очереди и кучи. В других случаях структуры данных используются для концептуального единства (абстрактный тип данных), например, имя и адрес человека. Часто структура данных может определить, запустится ли программа через несколько секунд, несколько часов или даже несколько дней.

С точки зрения физического и логического порядка структура данных может быть либо линейной, либо нелинейной. Другие точки зрения приводят к различным классификациям, которые включают гомогенные и гетерогенные, статические и динамические, постоянные и временные, внешние и внутренние, примитивные и совокупные, рекурсивные и нерекурсивные; пассивный против активного; и структуры с состоянием против структур без состояния.

6.2 Типы структуры данных

Как упоминалось выше, для классификации структур данных можно использовать разные точки зрения. Однако преобладающая точка зрения, используемая в классификации, основывается на физическом и логическом упорядочении элементов данных. Эта классификация делит структуры данных на линейные и нелинейные структуры. Линейные структуры организуют элементы данных в одном измерении, в котором каждая запись данных имеет одного (физического или логического) предшественника и одного последующего, за исключением первой и последней записи. Первая запись не имеет предшественника, а последняя запись не имеет преемника. Нелинейные структуры организуют элементы данных в двух или более измерениях, и в этом случае одна запись может иметь несколько предшественников и последователей. Примеры линейных структур включают списки, стеки и очереди. Примеры нелинейных структур включают кучи, хеш-таблицы и деревья (такие как бинарные деревья,

Другой тип структуры данных, часто встречающийся в программировании, — это составная структура. Составная структура данных строится поверх других (более примитивных) структур данных и в некотором смысле может рассматриваться как та же структура, что и базовая структура. Примеры составных структур включают наборы, графы и разделы. Например, раздел можно рассматривать как набор наборов.

6.3 Операции со структурами данных

Все структуры данных поддерживают некоторые операции, которые создают определенную структуру и порядок или извлекают соответствующие данные из структуры, сохраняют данные в структуре или удаляют данные из структуры. Основные операции, поддерживаемые всеми структурами данных, включают создание, чтение, обновление и удаление (CRUD).

- Создать: вставить новую запись данных в структуру.

- Чтение: получение записи данных из структуры.
- Обновление: изменение существующей записи данных.
- Удалить: удалить запись данных из структуры.

Некоторые структуры данных также поддерживают дополнительные операции:

- Найдите определенный элемент в структуре.
- Отсортируйте все элементы в соответствии с некоторым порядком.
- Пройдите все элементы в определенном порядке.
- Реорганизуйте или перебалансируйте структуру.

Разные структуры поддерживают разные операции с разной эффективностью. Разница в эффективности работы может быть значительной. Например, легко получить последний элемент, вставленный в стек, но поиск конкретного элемента в стеке довольно медленный и утомительный.

7 Алгоритмы и сложность

[5 , c1.1–1.3, c3.3–3.6, c4.1–4.8, c5.1–5.7, c6.1–6.3, c7.1–7.6, c11.1, c12.1]

Программы — это не случайные фрагменты кода: они тщательно написаны для выполнения ожидаемых пользователем действий. Руководство, используемое для составления программ, — это алгоритмы, которые организуют различные функции в последовательность шагов и учитывают предметную область, стратегию решения и используемые структуры данных. Алгоритм может быть очень простым или очень сложным.

7.1 Обзор алгоритмов

Абстрактно говоря, алгоритмы управляют работой компьютеров и состоят из последовательности действий, составленных для решения проблемы. Альтернативные определения включают, но не ограничиваются:

- Алгоритм — это любая четко определенная вычислительная процедура, которая принимает некоторое значение или набор значений в качестве входных данных и выдает некоторое значение или набор значений в качестве выходных данных.
- Алгоритм представляет собой последовательность вычислительных шагов, которые преобразуют входные данные в выходные данные.
- Алгоритм — это инструмент для решения хорошо определенной вычислительной задачи.

Конечно, разные люди предпочитают разные определения. Хотя общепринятого определения не существует, существует некоторое соглашение о том, что алгоритм должен быть правильным, конечным (другими словами, завершаться в конце концов или его нужно написать за конечное число шагов) и однозначным.

7.2 Атрибуты алгоритмов

Атрибуты алгоритмов многочисленны и часто включают в себя модульность, правильность, ремонтопригодность, функциональность, надежность, удобство для пользователя (т. е. легкость для понимания людьми), время программиста, простоту и расширяемость. Обычно подчеркивают атрибут «производительность» или «эффективность», под которым мы подразумеваем эффективность использования времени и ресурсов, при этом обычно подчеркивая ось времени. В некоторой степени эффективность определяет, выполним алгоритм или нет. Например, алгоритм, на завершение которого уходит сто лет, практически бесполезен и даже считается неверным.

7.3 Алгоритмический анализ

Анализ алгоритмов - это теоретическое исследование производительности компьютерной программы и использования ресурсов; в некоторой степени это определяет качество алгоритма. Такой анализ обычно абстрагируется от конкретных деталей конкретного компьютера и фокусируется на асимптотическом, машинно-независимом анализе.

Существует три основных типа анализа. В *анализе наихудшего случая* определяется максимальное время или ресурсы, требуемые алгоритмом для любого входа размера n . В *анализе среднего случая* определяется ожидаемое время или ресурсы, требуемые алгоритмом для всех входных данных размера n ; при выполнении анализа среднего случая часто необходимо делать предположения о статистическом распределении входных данных. Третий тип анализа — это анализ *наилучшего случая* , при котором определяется минимальное время или ресурсы, требуемые алгоритмом для любого входа размера n . Среди трех типов анализа анализ среднего случая является наиболее подходящим, но и наиболее трудным для выполнения.

Помимо основных методов анализа, существует также *амортизированный анализ* , при котором определяется максимальное время, необходимое алгоритму для последовательности операций; и *конкурентный анализ* , в котором определяется относительная производительность алгоритма по сравнению с оптимальным алгоритмом (который может быть неизвестен) в той же категории (для тех же операций).

7.4 Стратегии алгоритмического проектирования

Разработка алгоритмов обычно следует одной из следующих стратегий: грубая сила, разделяй и властвуй, динамическое программирование и жадный выбор. Стратегия *грубой силы* на самом деле не является стратегией. Он исчерпывающе пытается всеми возможными способами решить проблему. Если у проблемы есть решение, эта стратегия гарантированно найдет его; однако временные затраты могут оказаться слишком высокими. Стратегия « *разделяй и властвуй* » совершенствует стратегию грубой силы, разделяя большую проблему на более мелкие однородные проблемы. Он решает большую проблему, рекурсивно решая меньшие проблемы и объединяя решения меньших проблем, чтобы сформировать решение большой проблемы. В основе принципа «разделяй и властвуй» лежит предположение, что небольшие проблемы легче решить.

Стратегия *динамического программирования* совершенствует стратегию «разделяй и властвуй», признавая, что некоторые из подзадач, возникающих в результате разделения, могут быть одинаковыми, и, таким образом, позволяет избежать повторного решения одних и тех же проблем. Это устранение избыточных подзадач может значительно повысить эффективность.

Стратегия *жадного выбора* еще больше улучшает динамическое программирование, признавая, что не все подзадачи способствуют решению большой проблемы. Устраняя все подзадачи, кроме одной, стратегия жадного выбора достигает наивысшей эффективности среди всех стратегий разработки алгоритмов. Иногда использование *рандомизации* может улучшить стратегию жадного выбора, устранив сложность определения жадного выбора посредством подбрасывания монеты или рандомизации.

7.5 Стратегии алгоритмического анализа

Стратегии анализа алгоритмов включают *базовый анализ подсчета* , при котором фактически подсчитывается количество шагов, которые алгоритм выполняет для выполнения своей задачи; *асимптотический анализ* , при котором рассматривается только порядок количества шагов, которые алгоритм выполняет для выполнения своей задачи; *вероятностный анализ* , в котором используются вероятности для анализа средней производительности алгоритма; *амортизированный анализ* , в котором используются методы агрегирования, потенциала и учета для анализа наихудшей производительности алгоритма в последовательности операций; и *конкурентный анализ*, в котором используются такие методы, как потенциал и учет, для анализа относительной производительности алгоритма по отношению к оптимальному алгоритму.

Для сложных задач и алгоритмов может потребоваться использование комбинации вышеупомянутых стратегий анализа.

8 Основная концепция системы

[6 , c10]

Ян Соммервиль пишет: «Система — это целенаправленный набор взаимосвязанных компонентов, которые работают вместе для достижения некоторой цели» [6*]. Система может быть очень простой и включать всего несколько компонентов, как чернильная ручка, или достаточно сложной, как самолет. В зависимости от того, являются ли люди частью системы, системы можно разделить на технические компьютерные системы и социотехнические системы. Технические компьютерные системы

функционируют без участия человека, такие как телевизоры, мобильные телефоны, термостат и некоторое программное обеспечение; социотехническая система не будет функционировать без участия человека. Примеры таких систем включают пилотируемые космические аппараты, чипы, встроенные в человека, и так далее.

8.1 Свойства возникающей системы

Система – это больше, чем просто сумма ее частей. Таким образом, свойства системы — это не просто сумма свойств ее компонентов. Вместо этого система часто проявляет свойства, которые являются свойствами системы в целом. Эти свойства называются *эмерджентными,* потому что они развиваются только после интеграции составных частей в систему. Возникающие системные свойства могут быть как функциональными, так и нефункциональными. Функциональные свойства описывают то, что делает система. Например, функциональные свойства самолета включают плавучесть в воздухе, перевозку людей или грузов, а также использование в качестве оружия массового поражения. Нефункциональные свойства описывают, как система ведет себя в операционной среде. К ним могут относиться такие качества, как консистенция, вместимость, вес, безопасность и т.д.

8.2 Системная инженерия

«Системная инженерия — это междисциплинарный подход, регулирующий общие технические и управленческие усилия, необходимые для преобразования набора потребностей, ожиданий и ограничений клиентов в решение и для поддержки этого решения на протяжении всего срока его службы». [7]. Стадии жизненного цикла системной инженерии различаются в зависимости от создаваемой системы, но, как правило, включают определение системных требований, проектирование системы, разработку подсистемы, системную интеграцию, тестирование системы, установку системы, эволюцию системы и вывод системы из эксплуатации.

В прошлом было разработано множество практических руководств, помогающих людям выполнять действия на каждом этапе. Например, проектирование системы можно разбить на более мелкие задачи по идентификации подсистем, присвоению системных требований подсистемам, спецификации функциональных возможностей подсистем, определению интерфейсов подсистем и т. д.

8.3 Обзор компьютерной системы

Среди всех систем одной, которая, очевидно, имеет отношение к сообществу разработчиков программного обеспечения, является компьютерная система. Компьютер — это машина, которая выполняет программы или программное обеспечение. Он состоит из целенаправленного набора механических, электрических и электронных компонентов, каждый из которых выполняет заданную функцию. Вместе эти компоненты способны выполнять инструкции, данные программой.

Абстрактно говоря, компьютер получает некоторые входные данные, хранит некоторые данные и обрабатывает их, а также предоставляет некоторые выходные данные. Наиболее отличительной особенностью компьютера является его способность хранить и выполнять последовательности инструкций, называемых *программами* . Интересным явлением, связанным с компьютером, является универсальная эквивалентность по функциональности. Согласно Тьюрингу, все компьютеры с определенными минимальными возможностями эквивалентны в своей способности выполнять вычислительные задачи. Другими словами, при наличии достаточного количества времени и памяти все компьютеры — от нетбуков до суперкомпьютеров — способны выполнять одни и те же операции, независимо от скорости, размера, стоимости или чего-либо еще.

Большинство компьютерных систем имеют структуру, известную как «модель фон Неймана», которая состоит из пяти компонентов: *памяти* для хранения инструкций и данных, «центрального процессора для выполнения арифметических и логических операций, *блока управления* для последовательности и интерпретации ». инструкции, *ввод* для получения внешней информации в памяти и *вывод* для получения результатов для пользователя. Основные компоненты компьютерной системы, основанной на модели фон Неймана, изображены на рис. 13.3.

9 Компьютерная организация

[8 , c1-c4]

С точки зрения компьютера существует большой семантический разрыв между его предполагаемым поведением и работой основных электронных устройств, которые фактически выполняют работу внутри компьютера. Этот разрыв преодолевается за счет компьютерной организации, объединяющей различные электрические, электронные и механические устройства в одно устройство, образующее компьютер. Объектами, с которыми имеет дело компьютерная организация, являются устройства, соединения и элементы управления. Абстракция, встроенная в компьютерную организацию, — это компьютер.

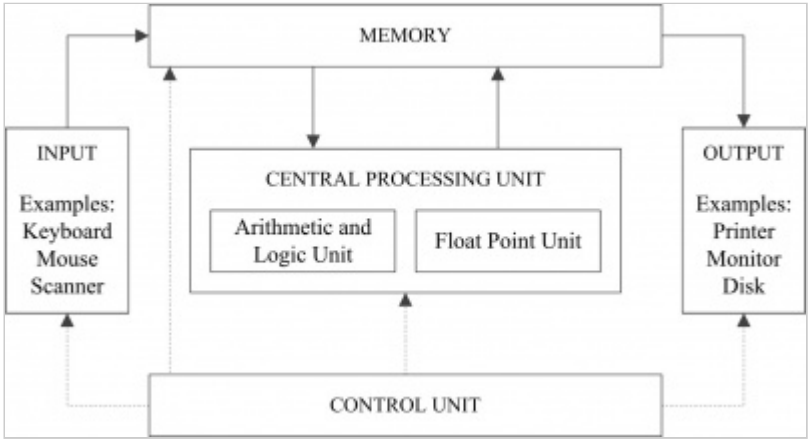

Рисунок 13.3: Основные компоненты компьютерной системы на основе модели фон Неймана

9.1 Обзор компьютерной организации

Компьютер обычно состоит из процессора, памяти, устройств ввода и вывода. Абстрактно говоря, организацию компьютера можно разделить на четыре уровня (рис. 13.4). Уровень *макроархитектуры* представляет собой формальную спецификацию всех функций, которые может выполнять конкретная машина, и известен как архитектура набора команд (ISA). Уровень *микроархитектуры* — это реализация ISA в конкретном процессоре, другими словами, то, как на самом деле выполняются спецификации ISA. Уровень *логических схем* — это уровень, на котором каждый функциональный компонент микроархитектуры состоит из схем, принимающих решения на основе простых правил. Устройства _уровень — это уровень, на котором, наконец, каждая логическая схема фактически построена из электронных устройств, таких как комплементарные металлооксидные полупроводники (CMOS), n-канальные металлооксидные полупроводники (NMOS) или транзисторы на основе арсенида галлия (GaAs) и т. д.

Каждый уровень обеспечивает абстракцию уровня выше и зависит от уровня ниже. Для программиста наиболее важной абстракцией является ISA, которая определяет такие вещи, как собственные типы данных, инструкции, регистры, режимы адресации, архитектуру памяти, обработку прерываний и исключений, а также операции ввода-вывода. В целом, ISA определяет возможности компьютера и то, что можно сделать на компьютере с помощью программирования.

9.2 Цифровые системы

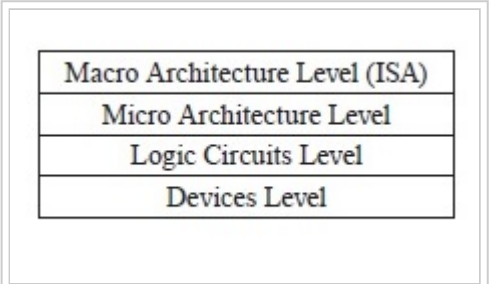

Рисунок 13.4: Уровни архитектуры машины

На самом низком уровне вычисления выполняются электрическими и электронными устройствами внутри компьютера. Компьютер использует схемы и память для хранения зарядов, отражающих наличие или отсутствие напряжения. Наличие напряжения равно 1, а отсутствие напряжения равно нулю. На диске полярность напряжения представлена 0 и 1, что, в свою очередь, представляет сохраненные данные. Все, включая инструкции и данные, выражается или кодируется цифровыми нулями и единицами. В этом смысле компьютер становится цифровой системой. Например, десятичное значение 6 может быть закодировано как 110, инструкция сложения может быть закодирована как 0001 и так далее. Компоненты компьютера, такие как блок управления, АЛУ, память и ввод-вывод, используют информацию для вычисления инструкций.

9.3 Цифровая логика

Очевидно, что логика необходима для манипулирования данными и управления работой компьютеров. Эта логика, стоящая за правильным функционированием компьютера, *называется цифровой* логикой, потому что она имеет дело с операциями цифровых нулей и единиц. Цифровая логика задает правила как построения различных цифровых устройств из простейших элементов (например, транзисторов), так и управления работой цифровых устройств. Например, цифровая логика определяет, каким будет значение, если ноль и единица соединены по И, по ИЛИ или исключительно по ИЛИ вместе. В нем также указывается, как создавать декодеры, мультиплексоры (MUX), память и сумматоры, которые используются для сборки компьютера.

9.4 Компьютерное выражение данных

Как упоминалось ранее, компьютер выражает данные электрическими сигналами или цифровыми нулями и единицами. Поскольку в выражении данных используются только две разные цифры, такая система называется *двоичной системой выражения* . Из-за внутренней природы двоичной системы максимальное числовое значение, выражаемое n-битным двоичным кодом, равно 2^{n-1}. В частности, двоичному числу $a_n a_{n-1} \ldots a_1 a_0$ соответствует $a_n \times 2^n + a_{n-1} \times 2^{n-1} + \ldots + a_1 \times 2^1 + a_0 \times 2^0$. Таким образом, числовое значение двоичного выражения 1011 равно $1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11$. Чтобы выразить нечисловое значение, нам нужно решить, сколько нулей и единиц использовать, а также порядок, в котором эти нули и единицы расположены.

Конечно, существуют разные способы кодирования, и это приводит к различным схемам и подсхемам выражения данных. Например, целые числа могут быть выражены в виде числа без знака, дополнения до единицы или дополнения до двух. Для символов существуют стандарты ASCII, Unicode и IBM EBCDIC. Для чисел с плавающей запятой существуют стандарты IEEE-754 FP 1, 2 и 3.

9.5 Центральный процессор (ЦП)

Центральный процессор — это место, где фактически выполняются инструкции (или программы). Выполнение обычно занимает несколько шагов, включая выборку инструкции программы, декодирование инструкции, выборку операндов, выполнение арифметических и логических операций над операндами и сохранение результата. Основные компоненты ЦП состоят из регистров, из которых часто считываются и записываются инструкции и данные, арифметико-логического устройства (ALU), которое выполняет фактические арифметические действия (такие как сложение, вычитание, умножение и деление) и логики (такие как операций И, ИЛИ, сдвига и т. д., блок управления, отвечающий за выработку надлежащих сигналов для управления операциями, и различные шины (данные, адрес и управление), которые связывают компоненты вместе и передают данные в и из эти компоненты.

9.6 Организация системы памяти

Память — это единица хранения данных компьютера. Речь идет о сборке крупномасштабной системы памяти из более мелких и одноразрядных единиц хранения. Основные темы, охватываемые архитектурой системы памяти, включают следующее:

- Ячейки и чипы памяти
- Платы и модули памяти
- Иерархия памяти и кеш
- Память как подсистема компьютера.

Ячейки и микросхемы памяти имеют дело с одноразрядным хранением и сборкой одноразрядных единиц в одномерные массивы памяти, а также сборкой одномерных массивов хранения в многомерные микросхемы памяти хранения. *Платы и модули* памяти предназначены для сборки микросхем памяти в системы памяти с упором на организацию, работу и управление отдельными микросхемами в системе. *Иерархия памяти и кеш* используются для поддержки эффективных операций с памятью. *Память как подсистема* имеет дело с интерфейсом между системой памяти и другими частями компьютера.

9.7 Ввод и вывод (ввод/вывод)

Компьютер бесполезен без ввода-вывода. Общие устройства ввода включают клавиатуру и мышь; общие устройства вывода включают диск, экран, принтер и динамики. Различные устройства ввода-вывода работают с разной скоростью передачи данных и надежностью. Как компьютеры подключают различные устройства ввода и вывода и управляют ими для облегчения взаимодействия между компьютерами и людьми (или другими компьютерами), является предметом обсуждения в I/O. Основные проблемы, которые должны быть решены при вводе и выводе, — это то, как может и должен выполняться ввод-вывод.

Как правило, ввод-вывод выполняется как на аппаратном, так и на программном уровне. Аппаратный ввод-вывод может быть выполнен любым из трех способов. *Выделенный* ввод-вывод выделяет ЦП для фактических операций ввода-вывода во время ввода-вывода; *ввод-вывод с отображением памяти* обрабатывает операции ввода-вывода как операции с памятью; а *гибридный* ввод-вывод объединяет выделенный ввод-вывод и ввод-вывод с отображением памяти в единый целостный режим работы ввода-вывода.

Так совпало, что программный ввод-вывод также может выполняться одним из трех способов. *Запрограммированный* ввод-вывод позволяет ЦП ждать, пока устройство ввода-вывода выполняет ввод-вывод; *ввод-вывод, управляемый прерываниями,* позволяет процессору управлять вводом-выводом устройством ввода-вывода; а *прямой доступ к памяти* (DMA) позволяет обрабатывать ввод-вывод вторичному процессору, встроенному в устройство DMA (или канал). (За исключением первоначальной настройки, основной ЦП не нарушается во время операции ввода-вывода DMA.)

Независимо от типов используемых схем ввода-вывода, основные проблемы, связанные с вводом-выводом, включают *адресацию* ввода-вывода (которая касается вопроса о том, как идентифицировать устройство ввода-вывода для конкретной операции ввода-вывода), *синхронизацию* . (который касается вопроса о том, как заставить ЦП и устройство ввода-вывода работать согласованно во время ввода-вывода), а также *обнаружения и исправления ошибок* (который касается возникновения ошибок передачи).

10 основ компиляции

[4 , c6.4] [8 , c8.4]

10.1 Обзор компилятора/интерпретатора

Программисты обычно пишут программы в коде языка высокого уровня, который ЦП не может выполнить; поэтому этот исходный код должен быть преобразован в машинный код, чтобы его мог понять компьютер. Из-за различий между разными ISA перевод должен выполняться для каждой рассматриваемой ISA или конкретного машинного языка.

Перевод обычно выполняется программой, называемой компилятором или интерпретатором. Этот процесс перевода с языка высокого уровня на машинный язык называется компиляцией или, иногда, интерпретацией.

10.2 Интерпретация и компиляция

Есть два способа перевести программу, написанную на языке более высокого уровня, в машинный код: интерпретация и компиляция. *Интерпретация* переводит исходный код по одному оператору за раз на машинный язык, выполняет его на месте, а затем возвращается к другому оператору. При каждом запуске программы требуются как исходный код языка высокого уровня, так и интерпретатор.

Компиляция переводит исходный код языка высокого уровня в полную программу на машинном языке (исполняемый образ) с помощью программы, называемой компилятором. После компиляции для запуска программы нужен только исполняемый образ. Большинство прикладных программ продается именно в этой форме.

Хотя и компиляция, и интерпретация преобразуют код языка высокого уровня в машинный код, между этими двумя методами есть некоторые важные различия. Во-первых, компилятор выполняет преобразование только один раз, в то время как интерпретатор обычно выполняет преобразование при каждом выполнении программы. Во-вторых, интерпретация кода выполняется медленнее, чем запуск скомпилированного кода, потому что интерпретатор должен анализировать каждое выражение в программе при его выполнении, а затем выполнять желаемое действие, тогда как скомпилированный код

просто выполняет действие в фиксированном контексте, определяемом компиляцией. В-третьих, доступ к переменным также медленнее в интерпретаторе, потому что сопоставление идентификаторов с местами хранения должно выполняться неоднократно во время выполнения, а не во время компиляции.

Основные задачи компилятора могут включать предварительную обработку, лексический анализ, синтаксический анализ, семантический анализ, генерацию кода и оптимизацию кода. Ошибки программы, вызванные неправильным поведением компилятора, бывает очень трудно отследить. По этой причине разработчики компиляторов тратят много времени на обеспечение корректности своего программного обеспечения.

10.3 Процесс компиляции

Компиляция — сложная задача. Большинство компиляторов делят процесс компиляции на несколько этапов. Типичная разбивка выглядит следующим образом:

- Лексический анализ
- Синтаксический анализ или разбор
- Семантический анализ
- Генерация кода

Лексический анализ разбивает входной текст (исходный код), представляющий собой последовательность символов, на отдельные *комментарии* , которые следует игнорировать в последующих действиях, и *базовые символы, имеющие лексические значения* . Эти базовые символы должны соответствовать некоторым терминальным символам грамматики конкретного языка программирования. Здесь терминальные символы относятся к элементарным символам (или токенам) в грамматике, которые не могут быть изменены.

Синтаксический анализ основан на результатах лексического анализа и обнаруживает структуру в программе и определяет, соответствует ли текст ожидаемому формату. *Является ли это текстуально правильной программой на C++?* или *эта запись верна по тексту?* типичные вопросы, на которые можно ответить с помощью синтаксического анализа. Синтаксический анализ определяет правильность исходного кода программы и преобразует его в более структурированное представление (дерево синтаксического анализа) для семантического анализа или преобразования.

Семантический анализ добавляет семантическую информацию в дерево синтаксического анализа, построенное во время синтаксического анализа, и строит таблицу символов. Он выполняет различные семантические проверки, включая проверку типов, привязку объектов (связывание ссылок на переменные и функции с их определениями) и определенное присваивание (требование инициализации всех локальных переменных перед использованием). При обнаружении ошибок семантически некорректные операторы программы отбрасываются и помечаются как ошибки.

После завершения семантического анализа начинается этап *генерации кода* , который преобразует промежуточный код, созданный на предыдущих этапах, в собственный машинный язык рассматриваемого компьютера. Это включает в себя решения о ресурсах и хранении, такие как решение о том, какие переменные должны помещаться в регистры и память, а также выбор и планирование соответствующих машинных инструкций, а также связанных с ними режимов адресации.

Часто можно объединить несколько фаз в один проход кода в реализации компилятора. Некоторые компиляторы также имеют фазу предварительной обработки в начале или после лексического анализа, которая выполняет необходимую вспомогательную работу, такую как обработка программных инструкций для компилятора (директив). Некоторые компиляторы предоставляют необязательную фазу оптимизации в конце всей компиляции, чтобы оптимизировать код (например, перестроить последовательность инструкций) для повышения эффективности и других желательных целей, запрошенных пользователями.

11 Основы операционных систем

[4 , c3]

Каждая система значимой сложности нуждается в управлении. Компьютер, как достаточно сложная электромеханическая система, нуждается в собственном менеджере для управления ресурсами и происходящими на нем действиями. Этот менеджер называется *операционной системой* (ОС).

11.1 Обзор операционных систем

Операционные системы — это набор программного обеспечения и микропрограмм, которые контролируют выполнение компьютерных программ и предоставляют такие услуги, как распределение компьютерных ресурсов, управление заданиями, управление вводом/выводом и управление файлами в компьютерной системе. Концептуально операционная система — это компьютерная программа, которая управляет аппаратными ресурсами и упрощает ее использование приложениями, предоставляя удобные абстракции. Эту приятную абстракцию часто называют виртуальной машиной, и она включает в себя такие вещи, как процессы, виртуальную память и файловые системы. ОС скрывает сложность базового оборудования и присутствует на всех современных компьютерах.

Основные роли, которые играют ОС, — это управление и иллюзия. *Управление* относится к управлению ОС (распределение и восстановление) физических ресурсов между несколькими конкурирующими пользователями/приложениями/задачами. *Иллюзия* относится к хорошим абстракциям, которые предоставляет ОС.

11.2 Задачи операционной системы

Задачи операционной системы существенно различаются в зависимости от машины и времени ее изобретения. Однако современные операционные системы пришли к соглашению относительно задач, которые должна выполнять ОС. Эти задачи включают управление ЦП, управление памятью, управление дисками (файловой системой), управление устройствами ввода-вывода, а также безопасность и защиту. Каждая задача ОС управляет одним типом физического ресурса.

В частности, управление ЦП связано с распределением и освобождением ЦП среди конкурирующих программ (называемых процессами/потоками на жаргоне ОС), включая саму операционную систему. Основной абстракцией, предоставляемой управлением ЦП, является модель процесса/потока. Управление памятью имеет дело с выделением и освобождением пространства памяти между конкурирующими процессами, и основной абстракцией, предоставляемой управлением памятью, является виртуальная память. Управление дисками связано с совместным использованием магнитных, оптических или твердотельных дисков несколькими программами/пользователями, и его основной абстракцией является файловая система. Управление устройствами ввода-вывода имеет дело с распределением и освобождением различных устройств ввода-вывода среди конкурирующих процессов. Безопасность и защита связаны с защитой компьютерных ресурсов от незаконного использования.

11.3 Абстракции операционной системы

Арсенал ОС — это абстракция. В соответствии с пятью физическими задачами ОС используют пять абстракций: процесс/поток, виртуальная память, файловые системы, ввод/вывод и домены защиты. Общая абстракция ОС — это виртуальная машина.

Для каждой области задач ОС существует как физическая реальность, так и концептуальная абстракция. Физическая реальность относится к управляемым аппаратным ресурсам; концептуальная абстракция относится к интерфейсу, который ОС представляет пользователям/программам выше. Например, в потоковой модели ОС физическая реальность — это ЦП, а абстракция — несколько ЦП. Таким образом, пользователю не нужно беспокоиться о совместном использовании ЦП с другими при работе с абстракцией, предоставляемой ОС. В абстракции виртуальной памяти ОС физическая реальность — это физическая ОЗУ или ПЗУ (независимо от того), абстракция — это множественное неограниченное пространство памяти. Таким образом, пользователю не нужно беспокоиться о совместном использовании физической памяти с другими или об ограниченном объеме физической памяти.

Абстракции могут быть виртуальными или прозрачными; в этом контексте виртуальный применяется к чему-то, что кажется там, но не является (например, полезной памяти за пределами физической), тогда как прозрачный применяется к чему-то, что есть, но кажется, что его нет (например, выборка содержимого памяти с диска или физической памяти).

11.4 Классификация операционных систем

Разные операционные системы могут иметь разную реализацию функционала. В первые дни компьютерной эры операционные системы были относительно простыми. С течением времени сложность и изощренность операционных систем значительно возрастает. С исторической точки зрения операционная система может быть классифицирована как одна из следующих.

- *Пакетная ОС* : организует и обрабатывает работу в пакетном режиме. Примеры таких ОС включают FMS от IBM, IBSYS и UMES от Мичиганского университета.
- *Многопрограммная пакетная ОС* : добавляет возможность многозадачности в более ранние простые пакетные ОС. Примером такой ОС является IBM OS/360.
- *ОС с разделением времени* : добавляет многозадачность и интерактивные возможности в ОС. Примеры таких ОС включают UNIX, Linux и NT.
- *ОС реального времени* : добавляет в ОС предсказуемость времени за счет планирования отдельных задач в соответствии со сроками завершения каждой задачи. Примеры таких ОС включают VxWorks (WindRiver) и DART (EMC).
- *Распределенная ОС* : добавляет в ОС возможность управления сетью компьютеров.
- *Встроенная ОС* : имеет ограниченную функциональность и используется для встроенных систем, таких как автомобили и КПК. Примеры таких ОС включают Palm OS, Windows CE и TOPPER.

В качестве альтернативы операционная система может быть классифицирована по соответствующей целевой машине/среде следующим образом.

- *ОС для мейнфреймов* : работает на мейнфреймах и включает OS/360, OS/390, AS/400, MVS и VM.
- *Серверная ОС* : работает на рабочих станциях или серверах и включает такие системы, как UNIX, Windows, Linux и VMS.
- *Многокомпьютерная ОС* : работает на нескольких компьютерах и включает такие примеры, как Novell Netware.
- *ОС для персональных компьютеров* : работает на персональных компьютерах и включает такие примеры, как DOS, Windows, Mac OS и Linux.
- *ОС для мобильных устройств* : работает на персональных устройствах, таких как сотовые телефоны, IPAD, и включает в себя такие примеры, как iOS, Android, Symbian и т. д.

12 Основы баз данных и управление данными

[4 , c9]

База данных состоит из организованного набора данных для одного или нескольких целей. В некотором смысле база данных — это обобщение и расширение структур данных. Но разница в том, что база данных обычно является внешней по отношению к отдельным программам и существует постоянно по сравнению со структурами данных. Базы данных используются, когда объем данных велик или важны логические отношения между элементами данных. Факторы, учитываемые при проектировании базы данных, включают производительность, параллелизм, целостность и восстановление после аппаратных сбоев.

12.1 Сущность и схема

То, что база данных пытается смоделировать и сохранить, называется сущностями. Сущности могут быть объектами реального мира, такими как люди, машины, дома и т. д., или они могут быть абстрактными понятиями, такими как лица, зарплата, имена и т. д. Сущность может быть примитивной, такой как имя, или составной, такой как сотрудник, который состоит из имени, идентификационного номера, зарплаты, адреса и т. д.

Единственным наиболее важным понятием в базе данных является *схема* , которая представляет собой описание всей структуры базы данных, на основе которой строятся все остальные действия с базой данных. Схема определяет отношения между различными сущностями, составляющими базу данных. Например, схема для системы расчета заработной платы компании будет состоять из таких элементов, как идентификатор сотрудника, имя, ставка заработной платы, адрес и т. д. Программное обеспечение базы данных поддерживает базу данных в соответствии со схемой

Еще одним важным понятием в базе данных является *модель базы данных,* которая описывает тип отношений между различными сущностями. Обычно используемые модели включают реляционные, сетевые и объектные модели.

12.2 Системы управления базами данных (СУБД)

Компоненты системы управления базами данных (СУБД) включают приложения баз данных для хранения структурированных и неструктурированных данных, а также необходимые функции управления базами данных, необходимые для просмотра, сбора, хранения и извлечения данных из баз данных. СУБД управляет созданием, обслуживанием и использованием базы данных и обычно классифицируется в соответствии с моделью базы данных, которую она поддерживает, например, реляционной, сетевой или объектной моделью. Например, система управления реляционными базами данных (RDBMS) реализует функции реляционной модели. Система управления базами данных объектов (ODBMS) реализует функции объектной модели.

12.3 Язык запросов к базе данных

Пользователи/приложения взаимодействуют с базой данных с помощью языка запросов к базе данных, который представляет собой специализированный язык программирования, предназначенный для использования в базе данных. Модель базы данных имеет тенденцию определять языки запросов, доступные для доступа к базе данных. Одним из наиболее часто используемых языков запросов для реляционных баз данных является язык структурированных запросов, чаще называемый SQL. Распространенным языком запросов для объектных баз данных является язык объектных запросов (сокращенно OQL). Существует три компонента SQL: язык определения данных (DDL), язык манипулирования данными (DML) и язык управления данными (DCL). Пример запроса DML может выглядеть следующим образом:

'ВЫБЕРИТЕ Component_No, Количество ИЗ КОМПОНЕНТА, ГДЕ Item_No = 100'

Приведенный выше запрос выбирает все Component_No и соответствующее количество из таблицы базы данных с именем COMPONENT, где Item_No равен 100.

12.4 Задачи пакетов СУБД

Система СУБД предоставляет следующие возможности:

- *Разработка базы данных* используется для определения и организации содержимого, взаимосвязей и структуры данных, необходимых для построения базы данных.
- *Опрос базы данных* используется для доступа к данным в базе данных для поиска информации и создания отчетов. Конечные пользователи могут выборочно извлекать и отображать информацию и создавать печатные отчеты. Это операция, которую большинство пользователей знают о базах данных.
- *Обслуживание базы данных* используется для добавления, удаления, обновления и исправления данных в базе данных.
- *Application Development* используется для разработки прототипов экранов ввода данных, запросов, форм, отчетов, таблиц и меток для прототипа приложения. Это также относится к использованию языка 4-го поколения или генераторов приложений для разработки или генерации программного кода.

12.5 Управление данными

База данных должна управлять хранящимися в ней данными. Это управление включает как организацию, так и хранение.

Организация фактических данных в базе данных зависит от модели базы данных. В реляционной модели данные организованы в виде таблиц с разными таблицами, представляющими разные объекты или отношения между набором объектов. Хранение данных связано с хранением этих таблиц базы данных на дисках. Обычный способ добиться этого — использовать файлы. Для этой цели используются последовательные, индексированные и хэш-файлы с различными файловыми структурами, обеспечивающими разную производительность и удобство доступа.

12.6 Интеллектуальный анализ данных

Часто нужно знать, что искать, прежде чем запрашивать базу данных. Этот тип «точечного» доступа не позволяет полностью использовать огромное количество информации, хранящейся в базе данных, и фактически сводит базу данных к набору отдельных записей. Чтобы в полной мере воспользоваться преимуществами базы данных, можно выполнить статистический анализ и обнаружение шаблонов содержимого базы данных, используя метод, называемый интеллектуальным анализом *данных* . Такие операции могут использоваться для поддержки ряда бизнес-операций, которые включают, помимо прочего, маркетинг, обнаружение мошенничества и анализ тенденций.

За последнее десятилетие было изобретено множество способов интеллектуального анализа данных, в том числе такие распространенные методы, как описание классов, различение классов, кластерный анализ, анализ ассоциаций и анализ выбросов.

13 Основы сетевого взаимодействия

[8 , c12]

Компьютерная сеть соединяет набор компьютеров и позволяет пользователям разных компьютеров совместно использовать ресурсы с другими пользователями. Сеть облегчает связь между всеми подключенными компьютерами и может создать иллюзию единого вездесущего компьютера. Каждый компьютер или устройство, подключенное к сети, называется *сетевым узлом* .

Появился ряд вычислительных парадигм, использующих функции и возможности, предоставляемые компьютерными сетями. Эти парадигмы включают распределенные вычисления, грид-вычисления, интернет-вычисления и облачные вычисления.

13.1 Типы сети

Компьютерные сети не одинаковы и могут быть классифицированы по широкому кругу характеристик, включая способ подключения к сети, проводные технологии, беспроводные технологии, масштаб, топологию сети, функции и скорость. Но знакомая большинству классификация основана на масштабах сетей.

- *Персональная сеть/Домашняя сеть* — это компьютерная сеть, используемая для связи между компьютерами и различными информационными технологическими устройствами, находящимися в непосредственной близости от одного человека. Устройства, подключенные к такой сети, могут включать ПК, факсы, КПК и телевизоры. Это основа, на которой строится Интернет вещей.
- *Локальная вычислительная сеть (LAN)* соединяет компьютеры и устройства в ограниченной географической зоне, такой как школьный городок, компьютерная лаборатория, офисное здание или близко расположенная группа зданий.
- *Campus Network* — это компьютерная сеть, состоящая из соединения локальных сетей (LAN) в пределах ограниченной географической области.
- *Глобальная вычислительная сеть (WAN)* — это компьютерная сеть, охватывающая большую географическую область, например город или страну, или даже межконтинентальные расстояния. WAN, ограниченная городом, иногда называется городской сетью.
- *Интернет* — это глобальная сеть, соединяющая компьютеры, расположенные во многих (возможно, во всех) странах.

Другие классификации могут подразделять сети на сети управления, сети хранения данных, виртуальные частные сети (VPN), беспроводные сети, сети точка-точка и Интернет вещей.

13.2 Основные сетевые компоненты

Все сети состоят из одних и тех же основных аппаратных компонентов, включая компьютеры, сетевые карты (NIC), мосты, концентраторы, коммутаторы и маршрутизаторы. Все эти компоненты называются узлами на сетевом жаргоне. Каждый компонент выполняет определенную функцию, необходимую для упаковки, соединения, передачи, усиления, управления, распаковки и интерпретации данных. Например, повторитель усиливает сигналы, коммутатор выполняет соединения «многие ко многим», концентратор выполняет соединения «один ко многим», интерфейсная карта подключается к компьютеру и осуществляет упаковку и передачу данных, мост соединяет одну сеть с другой, а маршрутизатор сам по себе является компьютером и выполняет анализ данных и управление потоком для регулирования данных из сети.

Функции, выполняемые различными сетевыми компонентами, соответствуют функциям, заданным одним или несколькими уровнями семиуровневой сетевой модели взаимодействия открытых систем (OSI), которая обсуждается ниже.

13.3 Сетевые протоколы и стандарты

Компьютеры взаимодействуют друг с другом с помощью протоколов, которые определяют формат и правила, используемые для упаковки и распаковки данных. Чтобы упростить связь и улучшить структуру, сетевые протоколы разделены на разные уровни, причем каждый уровень имеет дело с одним аспектом связи. Например, физический уровень имеет дело с физическим соединением между сторонами, которые должны взаимодействовать, канальный уровень имеет дело с передачей необработанных данных и управлением потоком, а сетевой уровень имеет дело с упаковкой и распаковкой данных в конкретный пакет. формате, понятном заинтересованным сторонам. В наиболее часто используемой сетевой модели OSI сетевые протоколы организованы в семь уровней, как показано на рис. 13.5.

Следует отметить, что не все сетевые протоколы реализуют все уровни модели OSI. Например, протокол TCP/IP не реализует ни уровень представления, ни уровень сеанса.

Для каждого уровня может быть более одного протокола. Например, UDP и TCP работают на транспортном уровне над сетевым уровнем IP, обеспечивая ненадежную транспортировку (UDP) с максимальной эффективностью по сравнению с надежной транспортной функцией (TCP). Протоколы физического уровня включают Token Ring, Ethernet, Fast Ethernet, гигабитный Ethernet и беспроводной Ethernet. Протоколы уровня канала передачи данных включают ретрансляцию кадров, режим асинхронной передачи (ATM) и протокол двухточечной связи (PPP). Протоколы прикладного уровня включают оптоволоконный канал, интерфейс малых компьютерных систем (SCSI) и Bluetooth. Для каждого уровня или даже для каждого отдельного протокола могут существовать стандарты, установленные национальными или международными организациями для руководства проектированием и разработкой соответствующих протоколов.

13.4 Интернет

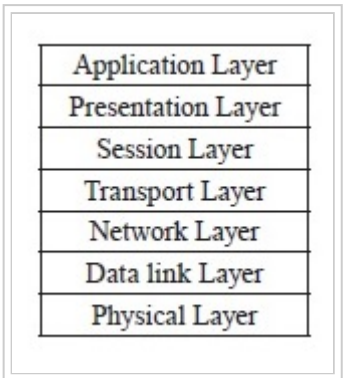

Рисунок 13.5: Семиуровневая сетевая модель OSI

Интернет представляет собой глобальную систему взаимосвязанных правительственных, академических, корпоративных, общедоступных и частных компьютерных сетей. В общественном достоянии доступ к Интернету осуществляется через организации, известные как интернет-провайдеры (ISP). Интернет-провайдер поддерживает один или несколько коммутационных центров, называемых точкой присутствия, которые фактически подключают пользователей к Интернету.

13.5 Интернет вещей

Интернет вещей относится к объединению в сеть повседневных объектов, таких как автомобили, сотовые телефоны, КПК, телевизоры, холодильники и даже здания, с использованием проводных или беспроводных сетевых технологий. Функция и цель *Интернета вещей* состоит в том, чтобы соединить все вещи, чтобы облегчить автономную и лучшую жизнь. Технологии, используемые в Интернете вещей, включают RFID, беспроводные и проводные сети, сенсорные технологии и, конечно же, большое количество программного обеспечения. Поскольку парадигма Интернета вещей все еще формируется, необходимо проделать большую работу, чтобы Интернет вещей получил широкое признание.

13.6 Виртуальная частная сеть (VPN)

Виртуальная частная сеть — это заранее спланированное виртуальное соединение между узлами в локальной/глобальной сети или в Интернете. Это позволяет сетевому администратору разделять сетевой трафик на группы пользователей, которые имеют общее сходство друг с другом, например, все пользователи в одной организации или рабочей группе. Этот тип схемы может повысить производительность и безопасность между узлами и упрощает обслуживание цепей при устранении неполадок.

14 Параллельные и распределенные вычисления

[8 , c9]

Параллельные вычисления — это вычислительная парадигма, возникшая с развитием многофункциональных блоков внутри компьютера. Основная цель параллельных вычислений — одновременное выполнение нескольких задач на разных функциональных устройствах и, таким образом, повышение пропускной способности или отклика, или того и другого. С другой стороны, распределенные вычисления — это вычислительная парадигма, которая возникает с развитием компьютерных сетей. Его основная цель состоит в том, чтобы либо использовать несколько компьютеров в сети для выполнения задач, которые в противном случае были бы невозможны на одном компьютере, либо повысить эффективность вычислений за счет использования мощности нескольких компьютеров.

14.1 Обзор параллельных и распределенных вычислений

Традиционно параллельные вычисления исследуют способы максимизации параллелизма (одновременного выполнения нескольких задач) в пределах границ компьютера. Распределенные вычисления изучают распределенные системы, состоящие из нескольких *автономных* компьютеров, которые обмениваются данными через компьютерную сеть. В качестве альтернативы распределенные вычисления могут также относиться к использованию распределенных систем для решения вычислительных или транзакционных задач. В первом определении распределенные вычисления исследуют протоколы, механизмы и стратегии, которые обеспечивают основу для распределенных вычислений; в последнем определении распределенные вычисления изучают способы разделения задачи на множество задач и назначения таких задач различным компьютерам, участвующим в вычислениях.

По сути, распределенные вычисления — это еще одна форма параллельных вычислений, хотя и в более широком масштабе. В распределенных вычислениях функциональными блоками являются не ALU, FPU или отдельные ядра, а отдельные компьютеры. По этой причине некоторые люди считают распределенные вычисления такими же, как и параллельные вычисления. Поскольку и распределенные, и параллельные вычисления предполагают некоторую форму параллелизма, их также называют параллельными вычислениями.

14.2 Разница между параллельными и распределенными вычислениями

Хотя параллельные и распределенные вычисления внешне похожи друг на друга, между ними есть тонкая, но реальная разница: параллельные вычисления не обязательно относятся к выполнению программ на разных компьютерах — вместо этого они могут выполняться на разных процессорах одного компьютера. . Фактически, консенсус среди профессионалов в области вычислений ограничивает область параллельных вычислений случаем, когда общая память используется всеми процессорами, участвующими в вычислениях, в то время как распределенные вычисления относятся к вычислениям, в которых для каждого процессора, участвующего в вычислениях, существует частная память.

Еще одно тонкое различие между параллельными и распределенными вычислениями заключается в том, что параллельные вычисления требуют одновременного выполнения нескольких задач, в то время как распределенные вычисления не имеют такой необходимости.

Основываясь на приведенном выше обсуждении, можно классифицировать параллельные системы как «параллельные» или «распределенные» в зависимости от наличия или отсутствия общей памяти между всеми процессорами: параллельные вычисления имеют дело с вычислениями на одном компьютере; распределенные вычисления имеют дело с вычислениями в наборе компьютеров. Согласно этой точке зрения, многоядерные вычисления — это форма параллельных вычислений.

14.3 Модели параллельных и распределенных вычислений

Поскольку в распределенных/параллельных вычислениях участвует несколько компьютеров/процессоров/ядер, необходима некоторая координация между вовлеченными сторонами для обеспечения правильного поведения системы. Разные способы координации порождают разные вычислительные модели. Наиболее распространенными моделями в этом отношении являются модель с общей памятью (параллельная) и модель с передачей сообщений (распределенная).

В модели с *общей памятью (параллельной)* все компьютеры имеют доступ к общей центральной памяти, где для увеличения вычислительной мощности используются локальные кэши. Эти кэши используют протокол для обеспечения свежести и актуальности локализованных данных, обычно это протокол MESI. Разработчик алгоритма выбирает программу для выполнения каждым компьютером. Доступ к центральной памяти может быть синхронным или асинхронным, и должен координироваться таким образом, чтобы сохранялась согласованность. Для этой цели были придуманы различные модели доступа.

В модели *с передачей сообщений (распределенной)* все компьютеры запускают некоторые программы, которые коллективно достигают определенной цели. Система должна работать корректно вне зависимости от структуры сети. Эту модель можно разделить на клиент-серверную (C/S), браузерно-серверную (B/S) и многоуровневую. В модели C/S сервер предоставляет услуги, а клиент запрашивает услуги у сервера. В модели B/S сервер предоставляет услуги, а клиент — это браузер. В n-уровневой модели каждый уровень (т. е. уровень) предоставляет услуги уровню, расположенному непосредственно над ним, и запрашивает услуги уровня, находящегося непосредственно под ним. По сути, n-уровневую модель можно рассматривать как цепочку моделей клиент-сервер. Часто уровни между самым нижним и самым верхним уровнями называются *промежуточным ПО* ., которая сама по себе является отдельным предметом изучения.

14.4 Основные проблемы распределенных вычислений

Координация между всеми компонентами в распределенной вычислительной среде часто является сложной и трудоемкой задачей. По мере увеличения количества ядер/процессоров/компьютеров возрастает и сложность распределенных вычислений. Среди многих проблем, с которыми приходится сталкиваться, когерентность памяти и консенсус между всеми компьютерами являются самыми сложными. Для решения этих проблем было изобретено множество парадигм вычислений, и они являются основными обсуждаемыми вопросами в распределенных/параллельных вычислениях.

15 основных человеческих факторов пользователя

[3 , c8] [9 , c5]

Программное обеспечение разрабатывается для удовлетворения человеческих желаний или потребностей. Таким образом, все проектирование и разработка программного обеспечения должны учитывать факторы человека и пользователя, такие как то, как люди используют программное обеспечение, как люди смотрят на программное обеспечение и что люди ожидают от программного обеспечения. Существует множество факторов взаимодействия человека и машины, и серия документов ISO 9241 определяет все подробные стандарты такого взаимодействия.[10] Но основные учитываемые здесь факторы «человек-пользователь» включают ввод/вывод, обработку сообщений об ошибках и надежность программного обеспечения в целом.

15.1 Ввод и вывод

Вход и выход — это интерфейсы между пользователями и программным обеспечением. Программное обеспечение бесполезно без ввода и вывода. Люди разрабатывают программное обеспечение для обработки некоторых входных данных и получения желаемого результата. Все инженеры-программисты должны рассматривать ввод и вывод как неотъемлемую часть программного продукта, который они проектируют или разрабатывают. Вопросы, рассматриваемые для ввода, включают (но не ограничиваются):

- Какой ввод требуется?
- Как ввод передается от пользователей к компьютерам?
- Как пользователям удобнее всего вводить данные?
- Какой формат требуется компьютеру для входных данных?

Разработчик должен запрашивать минимум данных от человека, только если данные еще не сохранены в системе. Разработчик должен форматировать и редактировать данные во время ввода, чтобы уменьшить количество ошибок, возникающих из-за неправильного или злонамеренного ввода данных.

Для вывода нам нужно учитывать, что пользователи хотят видеть:

- В каком формате пользователи хотели бы видеть результат?

- Каков наиболее приятный способ отображения вывода?

Если стороной, взаимодействующей с программным обеспечением, является не человек, а другое программное обеспечение, компьютер или система управления, тогда нам необходимо рассмотреть тип и формат ввода/вывода, которые должно создавать программное обеспечение, чтобы обеспечить надлежащий обмен данными между системами.

Разработчики должны следовать многим практическим правилам, чтобы обеспечить хороший ввод/вывод для программного обеспечения. Эти эмпирические правила включают в себя простой и естественный диалог, общение на языке пользователей, минимизацию нагрузки на память пользователя, согласованность, минимальное удивление, соответствие стандартам (независимо от того, согласовано это или нет: например, автомобили имеют стандартный интерфейс для акселератора, тормоза, рулевого управления).

15.2 Сообщения об ошибках

Понятно, что большая часть программного обеспечения содержит сбои и время от времени дает сбой. Но пользователи должны быть уведомлены, если есть что-то, что мешает плавному выполнению программы. Нет ничего более неприятного, чем неожиданное прекращение работы или отклонение в поведении программного обеспечения без какого-либо предупреждения или объяснения. Чтобы быть удобным для пользователя, программное обеспечение должно сообщать обо всех состояниях ошибок пользователям или приложениям верхнего уровня, чтобы можно было принять некоторые меры для исправления ситуации или корректного выхода. Существует несколько рекомендаций, определяющих, что представляет собой хорошее сообщение об ошибке: сообщения об ошибках должны быть ясными, содержательными и своевременными.

Во-первых, сообщения об ошибках должны четко объяснять, что происходит, чтобы пользователи знали, что происходит в программном обеспечении. Во-вторых, сообщения об ошибках должны точно указывать причину ошибки, если это вообще возможно, чтобы можно было предпринять надлежащие действия. В-третьих, сообщения об ошибках должны отображаться сразу после возникновения ошибки. По словам Якоба Нильсена, «хорошие сообщения об ошибках должны быть выражены простым языком (без кодов), точно указывать на проблему и конструктивно предлагать решение» [9*]. В-четвертых, сообщения об ошибках не должны перегружать пользователей слишком большим количеством информации и заставлять их полностью игнорировать сообщения.

Однако сообщения, относящиеся к ошибкам безопасного доступа, не должны содержать дополнительную информацию, которая могла бы помочь неавторизованным лицам проникнуть в систему.

15.3 Надежность программного обеспечения

Надежность программного обеспечения относится к способности программного обеспечения допускать ошибочные входные данные. Программное обеспечение считается надежным, если оно продолжает функционировать даже при вводе ошибочных данных. Таким образом, недопустимо, чтобы программное обеспечение просто аварийно завершало работу при возникновении проблемы с вводом, поскольку это может привести к неожиданным последствиям, таким как потеря ценных данных. Программное обеспечение, демонстрирующее такое поведение, считается ненадежным.

Нильсен дает более простое описание надежности программного обеспечения: «Программное обеспечение должно иметь низкий уровень ошибок, чтобы пользователи совершали мало ошибок во время использования системы и чтобы, если они делают ошибки, они могли легко исправить их. Далее, не должно происходить катастрофических ошибок» [9*].

Существует много способов оценить надежность программного обеспечения и столько же способов сделать его более надежным. Например, чтобы повысить надежность, всегда следует проверять достоверность входных и возвращаемых значений, прежде чем двигаться дальше; всегда следует генерировать исключение, когда происходит что-то неожиданное, и никогда не следует выходить из программы, не предоставив предварительно пользователям/приложениям возможность исправить состояние.

16 основных человеческих факторов разработчиков

[2 – c31, 32]

[3 , c31-32]

Человеческий фактор разработчика относится к учету человеческого фактора при разработке программного обеспечения. Программное обеспечение разрабатывается людьми, читается людьми и поддерживается людьми. Если что-то не так, люди несут ответственность за исправление этих ошибок. Таким образом, важно писать программное обеспечение таким образом, чтобы оно было легко понятно людям или, по крайней мере, другим разработчикам программного обеспечения. Программа, которую легко читать и понимать, демонстрирует удобочитаемость.

Средства, обеспечивающие соответствие программного обеспечения этой цели, многочисленны и варьируются от правильной архитектуры на макроуровне до определенного стиля кодирования и использования переменных на микроуровне. Но два важных фактора — это *структура* (или макеты программы) и *комментарии* (документация).

16.1 Структура

Хорошо структурированные программы легче понять и модифицировать. Если программа плохо структурирована, то никакие пояснения или комментарии не сделают ее понятной. Существует множество способов организации программы, начиная от правильного использования пробелов, отступов и круглых скобок и заканчивая удобным расположением группировок, пустых строк и фигурных скобок. Какой бы стиль ни был выбран, он должен быть последовательным во всей программе.

16.2 Комментарии

Для большинства людей программирование — это кодирование. Эти люди не понимают, что программирование также включает написание комментариев и что комментарии являются неотъемлемой частью программирования. Правда, комментарии не используются компьютером и, конечно, не являются окончательными инструкциями для компьютера, но они улучшают читабельность программ, объясняя смысл и логику операторов или участков кода. Следует помнить, что программы предназначены не только для компьютеров, они также читаются, пишутся и модифицируются людьми.

Типы комментариев включают повторение кода, объяснение кода, маркер кода, сводку кода, описание назначения кода и информацию, которая не может быть выражена самим кодом. Некоторые комментарии хорошие, некоторые нет. Хорошие те, которые объясняют назначение кода и объясняют, почему этот код выглядит именно так. Плохие — это повторение кода и указание неактуальной информации. Лучшие комментарии — это самодокументирующийся код. Если код написан настолько ясно и точно, что его значение самопровозглашается, то комментарии излишни. Но это легче сказать, чем сделать. Большинство программ не говорят сами за себя, и часто их трудно читать и понимать, если не даются комментарии.

Вот несколько общих правил написания хороших комментариев:

- Комментарии должны быть одинаковыми во всей программе.
- Каждая функция должна быть связана с комментариями, объясняющими назначение функции и ее роль в общей программе.
- Внутри функции комментарии должны быть даны для каждого логического раздела кодирования, чтобы объяснить значение и цель (намерение) раздела.
- Комментарии должны указывать, какая свобода есть (или нет) у поддерживающих программистов в отношении внесения изменений в этот код.
- Комментарии редко требуются для отдельных заявлений. Если заявление нуждается в комментариях, следует пересмотреть заявление.

17 Безопасная разработка и обслуживание программного обеспечения

[11 , c29]

Из-за увеличения числа вредоносных действий, направленных на компьютерные системы, безопасность стала серьезной проблемой при разработке программного обеспечения. Помимо обычной правильности и надежности, разработчики программного обеспечения также должны уделять внимание безопасности разрабатываемого ими программного обеспечения. Безопасная разработка программного обеспечения обеспечивает безопасность программного обеспечения, следуя набору установленных и/или

рекомендуемых правил и методов разработки программного обеспечения. Безопасное обслуживание программного обеспечения дополняет безопасную разработку программного обеспечения, гарантируя отсутствие проблем с безопасностью во время обслуживания программного обеспечения.

Общепринятая точка зрения на безопасность программного обеспечения состоит в том, что гораздо лучше внедрить безопасность в программное обеспечение, чем внедрять ее после разработки программного обеспечения. Чтобы внедрить безопасность в программное обеспечение, необходимо учитывать каждый этап жизненного цикла разработки программного обеспечения. В частности, безопасная разработка программного обеспечения включает в себя *безопасность требований к* программному обеспечению , безопасность разработки программного обеспечения , *безопасность разработки программного обеспечения* и *безопасность тестирования программного обеспечения* . Кроме того, безопасность также должна приниматься во внимание при обслуживании программного обеспечения, поскольку ошибки и лазейки безопасности могут возникать и часто возникают во время обслуживания.

17.1 Безопасность требований к программному обеспечению

Безопасность требований к программному обеспечению связана с разъяснением и спецификацией политики и целей безопасности в требованиях к программному обеспечению, что закладывает основу для соображений безопасности при разработке программного обеспечения. Факторы, которые следует учитывать на этом этапе, включают требования к программному обеспечению и угрозы/риски. Первый относится к конкретным функциям, которые необходимы для обеспечения безопасности; последнее относится к возможным способам угрозы безопасности программного обеспечения.

17.2 Безопасность дизайна программного обеспечения

Безопасность проектирования программного обеспечения связана с проектированием программных модулей, которые сочетаются друг с другом для достижения целей безопасности, указанных в требованиях безопасности. Этот шаг разъясняет детали соображений безопасности и разрабатывает конкретные шаги для реализации. Рассматриваемые факторы могут включать структуры и режимы доступа, которые устанавливают общие стратегии контроля/применения безопасности, а также отдельные механизмы применения политик.

17.3 Безопасность конструкции программного обеспечения

Безопасность разработки программного обеспечения касается вопроса о том, как писать фактический программный код для конкретных ситуаций, чтобы позаботиться о соображениях безопасности. Термин «безопасность разработки программного обеспечения» может означать разные вещи для разных людей. Это может означать способ кодирования конкретной функции, так что само кодирование является безопасным, или это может означать кодирование безопасности в программном обеспечении.

Большинство людей путают их вместе без различия. Одна из причин такой запутанности заключается в том, что неясно, как можно убедиться, что конкретное кодирование является безопасным. Например, в языке программирования C выражения i<<1 (сдвиг двоичного представления значения i влево на один бит) и 2*i (умножение значения переменной i на константу 2) семантически означают одно и то же. , но имеют ли они такое же разветвление безопасности? Ответ может быть разным для разных комбинаций ISA и компиляторов. Из-за этого непонимания безопасность разработки программного обеспечения — в ее нынешнем состоянии — в основном относится ко второму упомянутому выше аспекту: кодированию безопасности в программном обеспечении.

Запрограммировать безопасность в программное обеспечение можно, следуя рекомендуемым правилам. Вот несколько таких правил:

- Структурируйте процесс так, чтобы все разделы, требующие дополнительных привилегий, были модулями. Модули должны быть как можно меньше и выполнять только те задачи, которые требуют этих привилегий.
- Убедитесь, что все предположения в программе проверены. Если это невозможно, задокументируйте их для установщиков и сопровождающих, чтобы они знали предположения, которые злоумышленники попытаются опровергнуть.
- Убедитесь, что программа не разделяет объекты в памяти с какой-либо другой программой.

- Состояние ошибки каждой функции должно быть проверено. Не пытайтесь восстановить, если ни причина ошибки, ни ее последствия не влияют на какие-либо соображения безопасности. Программа должна восстановить состояние программного обеспечения до состояния, которое было до начала процесса, а затем завершиться.

17.4 Безопасность тестирования программного обеспечения

Безопасность тестирования программного обеспечения определяет, что программное обеспечение защищает данные и поддерживает заданную спецификацию безопасности. Для получения дополнительной информации см. КА по тестированию программного обеспечения.

17.5 Встраивание безопасности в процесс разработки программного обеспечения

Программное обеспечение безопасно настолько, насколько идет процесс его разработки. Чтобы обеспечить безопасность программного обеспечения, безопасность должна быть встроена в процесс разработки программного обеспечения. Одной из тенденций, возникающих в этом отношении, является концепция безопасного жизненного цикла разработки (SDL), представляющая собой классическую спиральную модель, которая использует целостный взгляд на безопасность с точки зрения жизненного цикла программного обеспечения и гарантирует, что безопасность является неотъемлемой частью проектирования и разработки программного обеспечения, а не запоздалая мысль позже в производстве. Утверждается, что процесс SDL снижает затраты на обслуживание программного обеспечения и повышает надежность программного обеспечения в отношении ошибок, связанных с безопасностью программного обеспечения.

17.6 Рекомендации по безопасности программного обеспечения

Хотя не существует надежных способов безопасной разработки программного обеспечения, существуют некоторые общие рекомендации, которые могут помочь в таких усилиях. Эти рекомендации охватывают все этапы жизненного цикла разработки программного обеспечения. Группа реагирования на компьютерные чрезвычайные ситуации (CERT) публикует некоторые заслуживающие доверия рекомендации, и ниже приведены 10 основных методов обеспечения безопасности программного обеспечения (подробности можно найти в [12]:

- 1. Подтвердите ввод.
- 2. Обращайте внимание на предупреждения компилятора.
- 3. Архитектор и дизайн политик безопасности.
- 4. Будьте проще.
- 5. Запрет по умолчанию.
- 6. Придерживайтесь принципа наименьших привилегий.
- 7. Очистите данные, отправленные в другое программное обеспечение.
- 8. Практикуйте глубокую защиту.
- 9. Используйте эффективные методы обеспечения качества.
- 10. Примите стандарт безопасности разработки программного обеспечения.

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

[1] Совместная рабочая группа по вычислительным учебным программам, IEEE Computer Society и Association for Computing Machinery, *Software Engineering 2004: Учебные рекомендации для программ бакалавриата по программной инженерии* , 2004, http://sites.computer.org/ccse/SE2004Volume.pdf .

[2] Г. Воланд, *Проектирование по дизайну* , 2-е изд., Прентис Холл, 2003.

[3] С. МакКоннелл, *Code Complete* , 2-е изд., Microsoft Press, 2004.

[4] Дж. Г. Брукшир, *Информатика: обзор* , 10-е изд., Addison-Wesley, 2008.

[5] Э. Горовиц и др., *Компьютерные алгоритмы* , 2-е изд., Silicon Press, 2007.

[6] И. Соммервиль, *Разработка программного обеспечения* , 9-е изд., Addison-Wesley, 2011.

[7] ISO/IEC/IEEE., *24765:2010 Системная и программная инженерия. Словарь* , ISO/IEC/IEEE, 2010.

[8] Л. Нулл и Дж. Лобур, *Основы компьютерной организации и архитектуры* , 2-е изд., издательство Jones and Bartlett Publishers, 2006.

[9] Дж. Нильсен, *Юзабилити-инжиниринг* , Морган Кауфманн, 1993.

[10] ISO, *9241-420:2011 Эргономика взаимодействия человека и системы* , ISO, 2011.

[11] М. Бишоп, *Компьютерная безопасность: искусство и наука* , издательство Addison-Wesley, 2002.

[12] RC Seacord, *Стандарт безопасного кодирования CERT C* , Addison-Wesley Professional, 2008.