Система — это набор связанных компонентов;

- система обладает свойствами, которыми не обладают компоненты по отдельности
- у системы есть интерфейс, через который она взаимодействует с окружением.

Имержентность – проявление новых свойств из частей системы.

Системы не существуют в изоляции.

Системы работают в окружении – среда, в которой они находятся.

В *вычислительных* системах мы через интерфейс наблюдаем процессы внутри и *интерпретируем результаты* -- так и происходят вычисления.

Природные вычислительные системы:

- random.org: случайные числа из атмосферного шума
- пульсары: точно отсчитывают время
- колония муравьёв: оптимизирует пути до еды

При запуске программы она встраивается в систему компьютера.

Программа (выполняющаяся) является частью программно-аппаратной системы; исходный код -- нет. Это как чертёж самолёта и сам самолёт. Два аспекта изучения программирования: как кодировать и как оно будет работать.

Можно разделить систему на разные уровни иерархии

Система стоит из частей. Два важных способа разделить:

- Композиционный(структурный): из чего система состоит?
- Функциональный: какая часть реализует какую функцию?

Эта совершенно разные разбиения.

Пример ножницы

- функциональные части: держатели и режущие
- композиционные(структурный): два куска металла и винт

Пример группа студентов, которая может скооперироваться и делать вместе какой-то проект

- функциональные части: разделить по ролям студентов, каждой роли может соответствовать несколько человек и каждому человеку может соответствовать несколько ролей

- композиционные: проект, отдельные студенты

Когда ты создаёшь свою систему, лучше думать о её функциональном разбиении. И из неё уже будет прорастать структурное разбиение.

Пример "устройство ввода" -- функциональный компонент, может быть мышь, планшет, а может быть программа. управляющая курсором.

Пример виртуальная память – функциональный компонент.

Пример процесс – функциональный компонент.

Процесс – контейнер со всем необходимым, чтобы работала программа – это набор потоков, виртуальная память, открытые файловые дескрипторы, информации про сигналы и т. д.

Пример: хранилище – функциональный компонент.

Хранилище – например Диск С, жесткий диск, база данных, распределённые файловые системы (данные лежат на множестве дисках, и каждые данные имеют резервную копию как минимум на 3, следовательно сложно сказать где именно находятся данные), RAM FS.

Сложность систем

Сложность — это главная проблема при создании (в т. ч. вычислительных) систем. Бывают разные определения:

1. Как сложно описать систему? Иногда измеряется в битах.

Ключевые слова: information, entropy, algorithmic complexity, Fisher's information, Renyi's entropy, code length (Hamming, Shennon-Fano, Hamming), Chernov's information, Lempel-Ziv complexity, Kolmogorov complexity.

2. Как сложно создать систему?

Keywords: Algorithmic space/time complexity, logical depth, termodynamic depth, crypticity.

3. Насколько система регулярна?

Состоит ли из маленьких однотипных вещей?

Например: есть движок для MapReduce (суть MapReduce: есть большая задача, которая хорошо разбивается на маленькие подзадачи, а потом из подзадач можно собрать решение большой задачи). Таким образом можем посчитать сумму массива из миллиарда элементов. Получилась регулярная несложная система.

Признаки сложной системы:

1. Много компонент.
2. Много связей.
3. Много нерегулярностей (разбор по частным случаям).
4. Большое описание.
5. Большая команда работает над системой.

Борьба со сложностью при конструировании систем

Конструировать систему можно как угодно. Можно сделать компьютер, в котором все ассемблерные инструкции будут прибавлять ко всем регистрам 1 (но неясно, как извлечь из этого пользу).

Путём проб и ошибок нашли некоторые рецепты, которые нередко эффективны и достаточно универсальны.

Одни из способов борьбы со сложностью: **модульность** + **абстракция**

Модульность: делаем систему по кусочкам (с описанными интерфейсами), затем собираем как конструктор.

Абстракция: скрываем сложность модулей и описываем их как чёрные ящики (важно то, как они себя ведут при взаимодействии с окружением, а не почему). Например ассемблер – мощная абстракция над миллиардом транзисторов.

Три фундаментальных типа функциональных компонентов

Вычислять можно как угодно. Но полезно и эффективно выделить следующие правила.

Как всё же выделить из всех возможных вычислительных систем хоть какие-то полезные и их категоризовать? Оказывается, в существующих вычислительных системах почти все функциональные компоненты попадают в одну из трёх категорий:

- **Исполнитель** (interpreter) Реакция на события, выполнение команд
- **Память** (memory) Хранение данных
- **Транспорт** (communication link) Связь между компонентами

Мы также говорим, что это три фундаментальные абстракции; в данном случае под абстракцией мы понимаем не *принцип построения систем*, а конкретный функциональный компонент, в чьё устройство мы не лезем. Скажем, оперативная память хранит данные по линейным адресам (но там есть конденсаторы, а адреса не линейные)

Примеры:

- Исполнитель (interpreter)

Процессор, контроллеры, интерпретаторы и компиляторы, виртуальные машины, Word, браузеры, игры

- Память (memory)

Триггеры, регистры, оперативная память, кэши, виртуальная память, HDD/SDD, RAID, базы данных, файловые системы, облачное хранилище

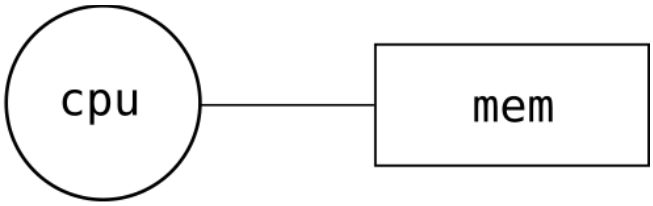
- Транспорт (communication link)

Разные виды кабелей, wifi, usb, internet, telegram, email, socket, pipe, просто файл-буфер…

Вычислительные системы из конструктора

А теперь посмотрим на комбинации этих абстракций. Помните, что блоки на схемах — это функциональная декомпозиция: за каждым элементом или связью может скрываться сложная, многослойная структура.

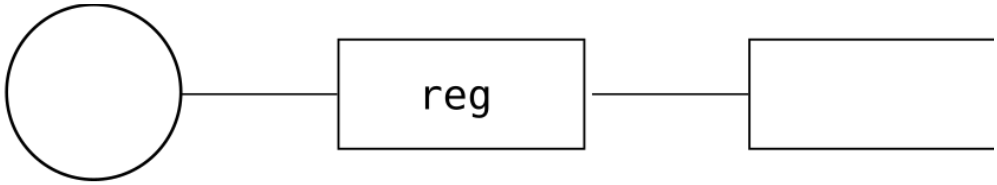
Архитектура фон Неймана



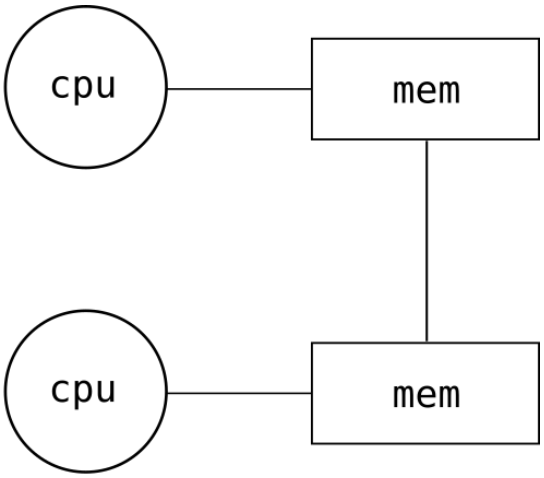
cpu, memory и канал обмена данными между ними

Но нам нужна большая детализация

Архитектура фон Неймана с регистрами

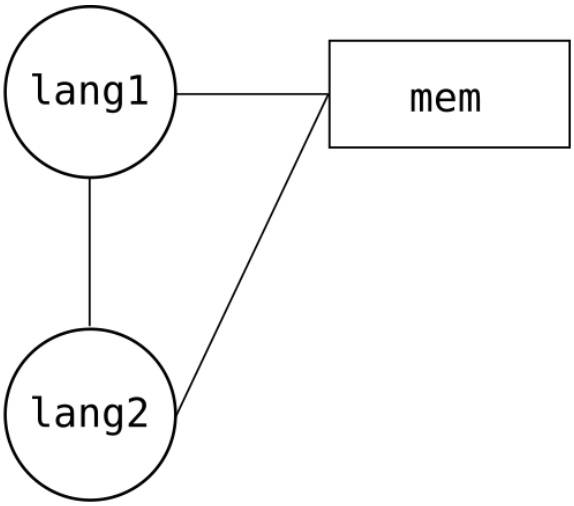


Сеть между двумя компьютерами



Обратите внимание, что соединяющий компьютеры транспорт может быть очень сложным; там могут проходить процессы поиска пути для пакетов, повторной посылки потерянных пакетов, несколько уровней протоколов (сетевая модель OSI).

Мультипроцессорная система с общей памятью



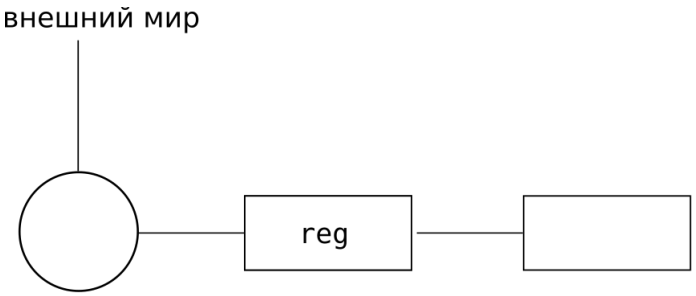
lang2 – интерпретатор, в него подгружаем программу. Он написан на языке lang1 и выполняется, например, поверх голого железа. Интерпретатор может быть написан на C и скомпилирован под какую-то архитектуру, а выполнять, например, Python.

Например, lang2 – Python, lang1 – машинные команды.

Модель нашей вычислительной системы

Это модель выполнения программ на процессоре архитектуры Intel 64. Вырастим её из фон Неймана.

Возьмём схему фон Неймана: один процессор, одна память, канал обмена данными, и добавим регистры. Но существует недостаток - неинтерактивность: простой при работе с медленными внешними устройствами. Для интерактивности мы добавим связь с внешним миром и прерывания.

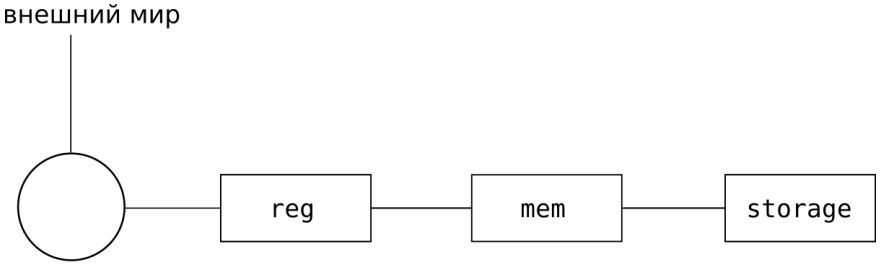


архитектура фон Неймана + регистры + прерывания

В том числе в рамках кампании по борьбе со сложностью модуляризировали архитектуру и каждый стал развивать своё. А повысить скорость процессора оказалось гораздо проще, чем памяти, отсюда разры в в скорости работы компонентов и попытка сгладить его дополнительными частями системы.

Это, кстати, общий принцип: хотим выжать производительность -- как правило усложняем систему, а ещё и в худшем случае понижаем производительность

Начинаем организовывать пирамиду памяти. Память медленная, добавим регистры, кэш. Медленная внешняя память. В среднем, мы загружаем данные в кэш, много с ними работаем, а потом синхронизируем с памятью.



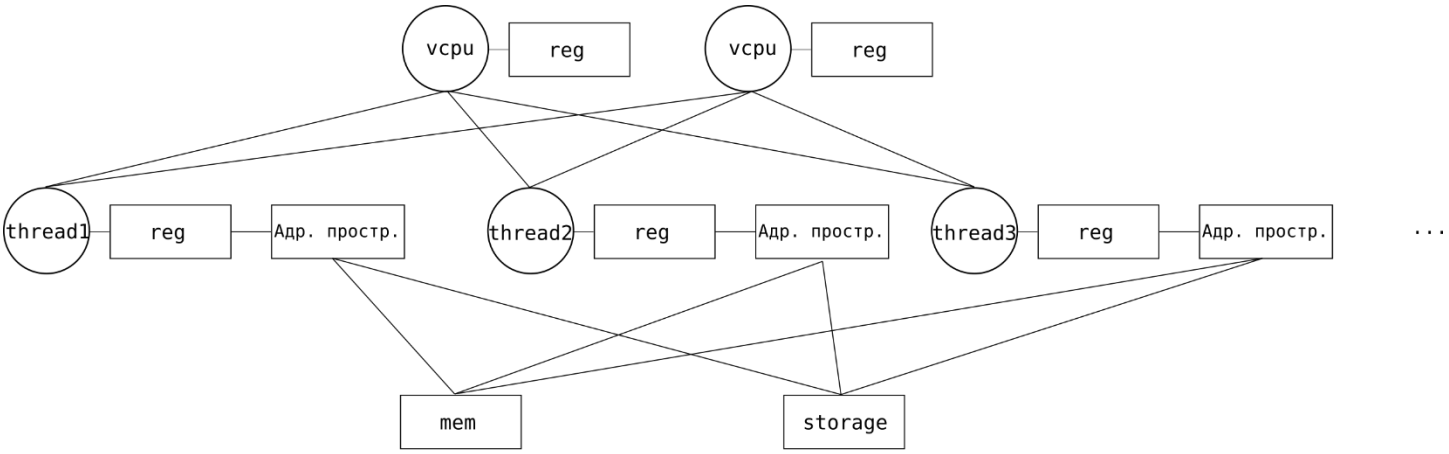
архитектура фон Неймана + регистры + прерывания + HDD

К этой системе мы добавляем стек – он не является структурным компонентом. Это функциональный компонент системы, который нам помогает изолировать куски кода в процедурах (они не пересекаются между собой), а также сохранять контекст выполнения и потом к нему возвращаться.

Тем самым добавился аппаратный стек (но в рамках той же линейно адресуемой памяти).

И теперь столкнулись с проблемой при запуске множества программ, они будут друг другу мешаться в памяти. Чтобы их изолировать друг от друга мы придумали виртуальную память. Мы виртуализируем память с помощью виртуальной памяти, мы виртуализируем процессор с помощью процессов и потоков. Мы даём каждой программе иллюзию что она единственная в оперативной памяти.

Не позволяем выполнять всем опасные инструкции (привилегированный режим, кольца защиты, реализованы с помощью сегментов).



Примерно так обычно смотрит на компьютер программист на ASM

mem – оперативная память, storage – HDD or SSD. С помощью них мы организовываем адресные пространства для каждой программы. У нас есть некоторое количество виртуальных процессоров, каждый из которых обладает своим набором регистров, и они соответствуют ядрам процессора. Они могут параллельно выполнять столько потоков, сколько ядер. Но обычно потоков больше, чем ядер и ядра выполняют то один, то другой поток (в зависимости от того, как решит планировщик).

Существует виртуализация регистров в процессор. Когда мы переключаемся на другой процесс, состояние процесса сохраняется, потом восстанавливается. То есть регистры с которыми работает процесс – виртуальные, они работают поверх железных регистров, но поверх их работают и регистры других процессов.

Так же есть виртуализация регистров на самих процессорах. Есть регистр rax, rbx, r10. Регистр r10 – виртуальный, он то одному настоящему (железному) регистру, то другому. Так как программы на машинных кодах в современном (intel) процессоре не выполняются, они транслируются в более низкого уровня машинный код (в процессе трансляции там проходят оптимизации), а потом более низкоуровневый код исполняется с реальными регистрами.

Например, если мы переслали данные из rbx в rax, а потом в регистр rbx записали 10. Вместо того чтобы естно скопировать содержимое в rax из rbx, а потом в регистр rbx записать 10 процессор может решить изменить соответствие между виртуальными регистрами (один из которых rbx) и настоящими регистрами (r1, r2, r3, …. r64). И процессе этой оптимизации теперь у нас виртуальному регистру rax соответствует настоящий регистр, который соответствовал rbx. И мы как бы совершили пересылку из rbx в rax.

Регистры, с которыми мы работаем на уровне асемблера не являются настоящими. Они как-то отображаются на реальные регистры.