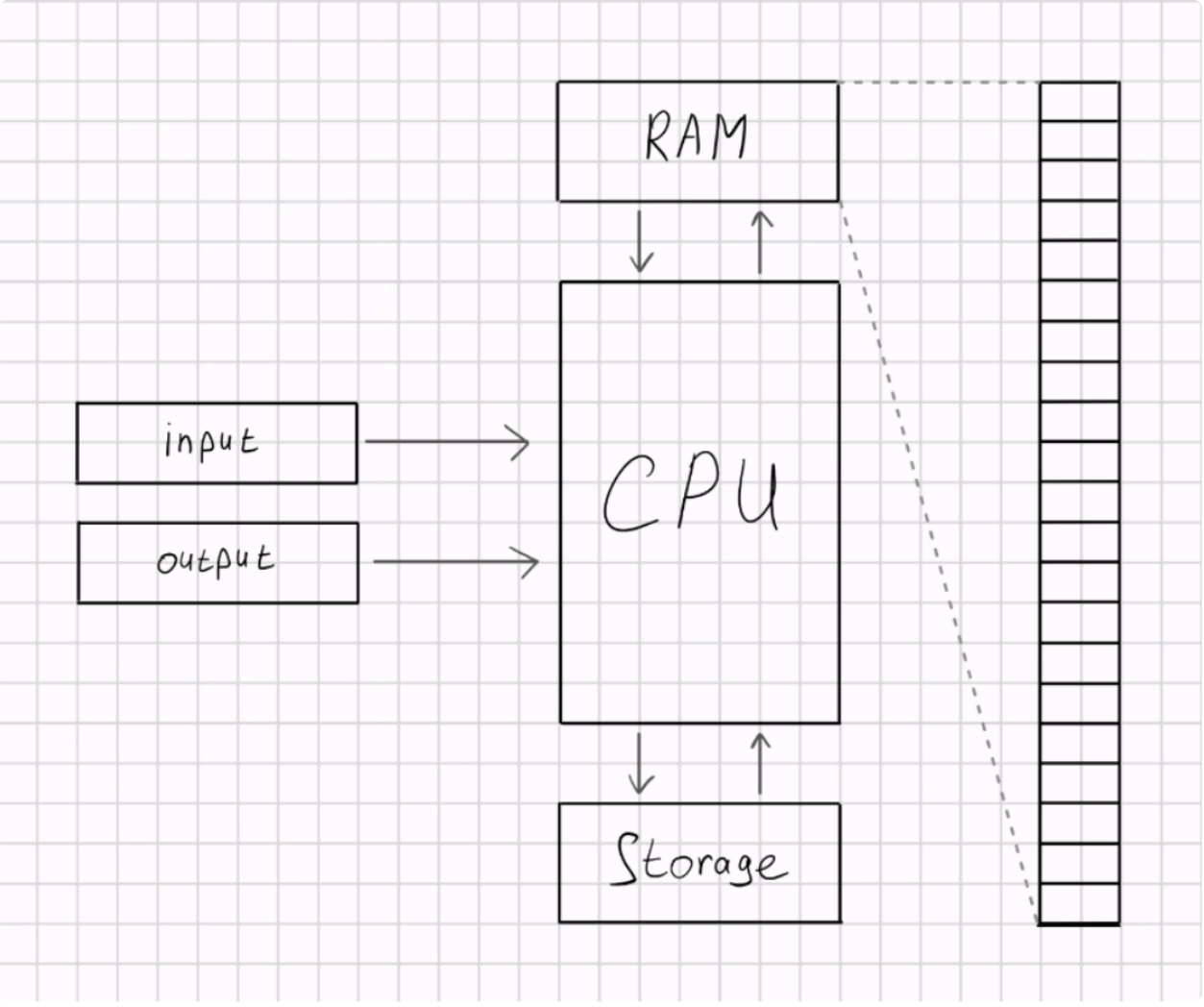
1. Развитие архитектуры ЭВМ

Lec 1

Раньше запускалась одна программа и не более.
Архитектура Фон Неймана



Идея супермаркета:

получение ресурсов в любое время, любой последовательностью действий
Реальность: борьба за ресурсы, потребность в эффективности и оптимизации

- Пропуск с меньшим количеством продуктов для освобождения ресурсов -> оптимизация и прибыль
  В другую сторону: 100 людей с бутылкой vs 1 покупатель с тележкой

> Критерий эффективности *почти всегда* противоречит критерию справедливости, кроме абсолютного равенства покупателей

- Попытки противоречить правилам и быстрее 'пройти кассу' с получением желаемого
- Реакция 'администрации' на нарушение правил (охрана)
- Желание получение прибыли со стороны супермаркета
- Возникновение высшей власти с исключающими правами (можно носить оружие)

Операционная система <-> Государству

Операционная система абстрагировала собой *всех от всех*:

1. *Software* - программное обеспечение
2. *Hardware* - аппаратное обеспечение
3. *Users* - пользователи

При их связи возникает паутина, которая путает все -> Возникает проводник в виде *Операционной системы*, через которую каждый компонент 'общается' друг с другом.

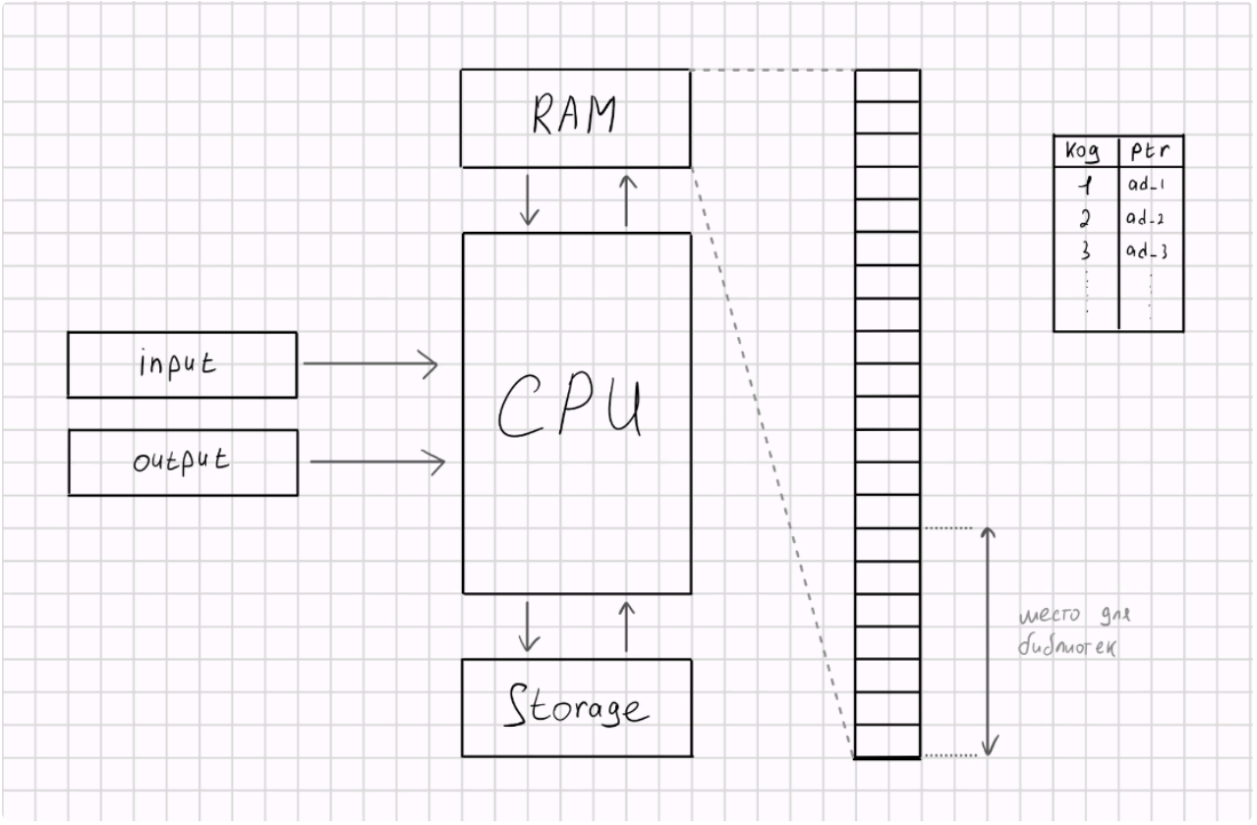
Операционная система - базовое системное программное обеспечение (основа любого вычислительного устройства), управляющая работой вычислительного узла и реализующая универсальный интерфейс между аппаратным, программным обеспечением и пользователем.

Принципы архитектуры:

1. Однородность (в любой ячейке памяти может быть любое содержимое)
2. Адресность (пространство памяти адресуемо)
3. Единство кодирование (в ячейках памяти содержится двоичный код)
4. Программное управление aka '*Берешь следующий*' (последовательное исполнения инструкций в памяти)
    - Только сама программа в которой выполняется инструкция, может использовать не следующая ячейку памяти

Использование библиотек

Выделение некоторого количества памяти под процессы которые хотим использовать как библиотеки
Выделение памяти для аргумента -> переход на ячейку для вычисления желаемого -> вычисление по инструкции -> возврат в ячейку вызова + 1

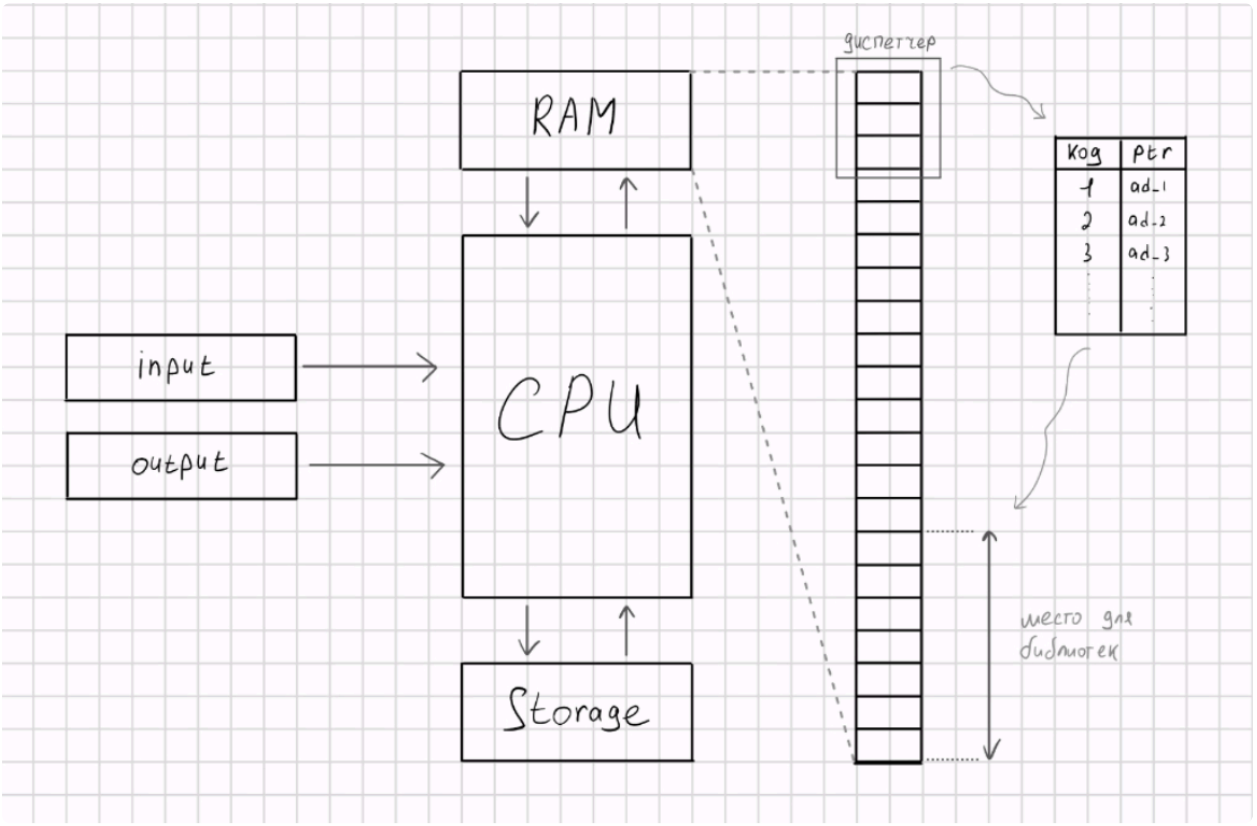


Проблема: изменение кода функции влечет сдвиг ячеек и проблемы при последующей адресации

Идея: выделить область памяти в начале (Диспетчер) + таблица с кодом программ с указателями

> Программа вызывает диспетчер для исполнения функций, который уже все решает и находит что нам нужно вызвать (+ чинит сдвиги и проблемы)

> Можно усложнить диспетчер на проверки входных данные и еще всякие примочки



Появление контроллера для взаимодействия оперативной памяти и хранилища. Управление осуществляет процессором **=>** разделение обязанностей (контроллер работает с памятью), процессор делает вычислительные действия.

Проблема: когда понимать, что контроллер завершил выполнение ? (Иначе запись данных может испортиться другой записью поверх)

Прерывание - сигнал, поступающий от внешнего устройства к центральному процессору, прерывающий выполнение текущего выполнения команд и передающий управление подпрограмме обработчика прерывания.

Выделение памяти для обработчика подпрограммы

Однопрограммная пакетная обработка

/* Продолжение лекции 1... (TODO) */
прерывание идет от cpu к controller

Lec 2

Напоминание: *Для обеспечения оптимизации и ускорения приходится нарушать порядок выполнения процессов -> возникновение новых проблем

Концепция мультипрограммной обработки

- Архитектура осталось той же (даже сохранились некоторые принципы)

Напоминание: появляется идея заниматься одной программой и обрабатывать другую (откачку подкачку файлов)

После этого диспетчер становится **мультипрограммной ОС**

Задача: понять почему нет нагруженности на все компоненты

> Все еще воспринимаем архитектуру одноядерной

Если мы хотим разделять выполнение как минимум двух программ (одна стоит, другая исполняется и наоборот), то они должны передавать друг другу управление

1. Кооперативная многозадачность

Допустим через каждый 100 тиков передаем управление другому
Проблемы:

- Будет злодей, который нарушит это правило
- Ветвление кода создает проблемы вернутся в точку обратно, где мы передали управление

2. Вытесняющая многозадачность

Буквально внедрили в железо кварцевые часы и они заставляют процессор прервать выполнение команд

1. Передача управления в диспетчер чтобы он нашел следующего
2. Реализуется сохранение и загрузку контекста (состояние всех регистров) для конкретной программы
3. Запуск процесса в зависимости от его сохраненного контекста

> Решена проблема processor sharing
> Появилась проблема оперативной памяти

На каком-то уровне код в любом случае приходил к конкретному адресу в памяти (чем будет адрес начала запуска программы)

Виртуализация памяти

Виртуальная память - абстракция позволяющая при разработке или компиляции программ использовать адресация от виртуального нуля, а при запуске или исполнении подменять адреса на физические

> Теперь *Диспетчер* знает, где лежит код каждой программы и какие адреса нужны (еще одна табличка)

Проблема: Выход за границы массива в адрес другой программы ломает содержимое её ячейки памяти. Потом когда-то запускается эта сломанная программа и неправильно исполняется создавая еще ошибки

Решение: Если разрешили программе исполняться, то будет контролировать её исполнение. Если она ломается -> убиваем

Мы хотим получать сообщение о том что случилось: Перед тем как умереть процесс посылает системный вызов с сохраненной проблемой, откуда можно найти где он сломался (абстрактно)

Проблема: Мы сломались, но должны вернутся в диспетчер

Решение: Понятие *привилегированного режима*. В нем мы отключаем защиту памяти. То есть любой процесс при проблеме может переключить режим и поставит указатель на диспетчер $=>$ Наложение ответственности на диспетчер (в частности на реализацию всю ОС).

Системный вызов - обращение пользовательского процесса к ядру операционной системы с требованием предоставить ресурс или выполнить привилегированную операцию

Начинаем наделять диспетчер уникальной властью.

Еще проблема: Планирование ресурсов.

```
В супермаркете все люди собрались ради одного ресурса -> Поставим
охранников чтобы регулировать действие людей -> Куда их отправлять если
```

```
не умеем предсказывать ?
```

И еще: Универсальный доступ к хранилищу. Код не должен быть привязан к конкретному экземпляру хранения.

- Передача код для разных проектов
- На другие устройства
- Сохранение информации местонахождения

Очередная задача для диспетчера: работа с файлами с помощью очередной мапы

По базе: *Передача данных. Опять обращение к диспетчеру с просьбой.

```
command1 | command2
```

Как реализовать этот конвейер ?

```
std_out command1 = std_in command2
```

- Чтение пустого буфера от command2
- Переполнение command1 и нехватка памяти

В итоге

- Программа работает, а потом спит и не знает кто там работает с ней "параллельно"
- Виртуализация делает иллюзию собственного начала и закрывает другие
- Иллюзия нескольких маленьких компьютеров в каждой виртуальной ячейке
- Привилегированный диспетчер

Получили настоящую концепцию ОС

в 1963г. выпускается B5000, в которой есть разросшийся диспетчер стала первой ОС (MCP)

3. Ситуация в реальной жизни

- Компьютеры занимали много места
- Требовалось обслуживание
- Появилась концепция покупки компьютера на время (запуск кода на время за твои деньги)
- Ввод-Вывод в одну точку
- Дефицит *суперкомпьютеров*

Появляется идея передавать электрическим сигналом закодированный двоичный код (синусоидой)

Параметры:

- Амплитуда
- Частота
- Фаза

Амплитудная модуляция + Частотная модуляция

Устройство для модуляции и демодуляция -> Модем

Появляются модемы, указывающие на терминалы через которые передаётся информация на дистанции

ПРОБЛЕМЫ:

- Кто-то позвонил - Он оплатил ? Появляются учетные записи + идентификация (фиксирование параметров) + аутентификация (проверка параметров идентификации) + логины (от logical income)
- Появляются биллинговые системы (от bill) для использования компьютерного времени
- Часовые пояса, разное время, разная нагрузка на сервера.

Получаем:

- Полную автономность компьютера (нету единственного входного и выходного потока)

Проблемс:

- Появляется все больше разных архитектур

- Разная логика для каждой ОС
- Нужно разбираться как работает каждая архитектура
- Потребность в абстракции (разделение кода от конкретного железа)