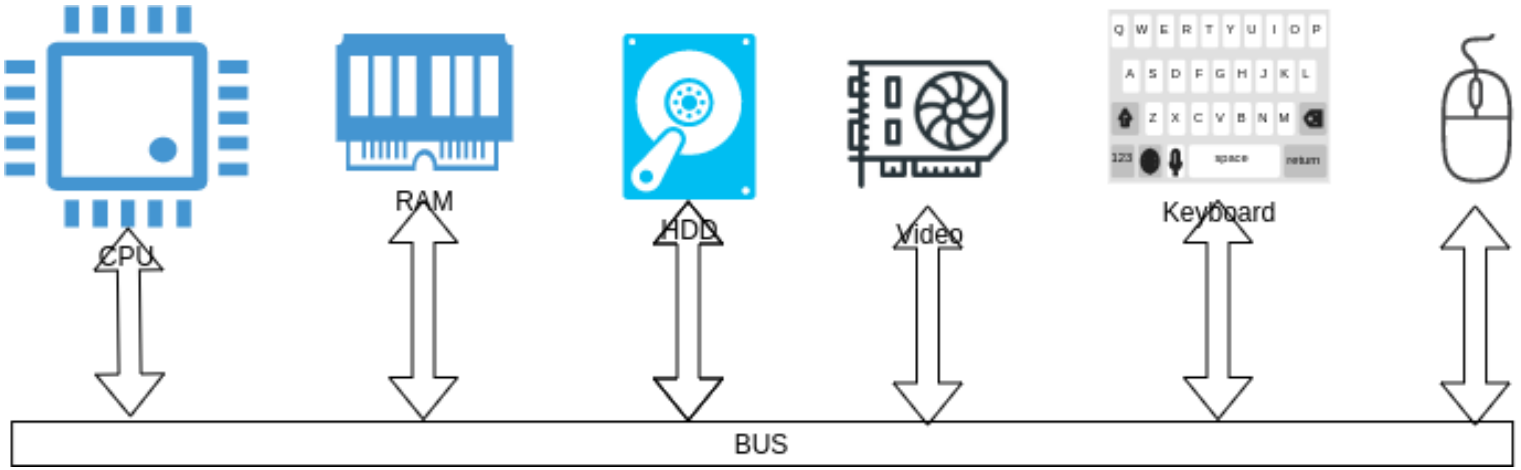
Лекция 1. ЯВУ. Программа из нескольких файлов.

Евгений Линский

Все хранится в числах

- Компьютер умеет работать только с числами
  - Текст. Кодировка задает соответствие между изображением символа и числовым кодом ('A' – 65).
  - Изображение. Цвет точки на экране — три числа Red Green Blue (черный — 0 0 0).
  - Команды, из которых состоит программа, тоже хранится в виде чисел.
- Числа хранятся в двоичной системе счисления
  - "Есть сигнал/нет сигнала" (1/0)
  - "Есть намагниченность/нет намагниченности" (1/0)

Схема компьютера



Можно сказать, что все что умеет процессор, это выполнять над числами арифметические и логические операции и пересылать данные между периферийными устройствами:

- арифметическая операция: загрузить числа из памяти в процессор, выполнить операции, выгрузить в память
- вывести пиксель: переслать координаты и цвет точки (RGB) в видеокарту
- сохранить файл: послать данные и их положение на диске в контроллер жесткого диска

Программирование в двоичных кодах

- Архитектура процессора (x86, ARM, RISC-V): набор команд и регистры
  - Регистры — ячейки памяти внутри процессора (x86: размер – 64 бита, количество – 20)
- Программа

загрузить из ячейки RAM 300 в регистр1	1 300 1
загрузить из ячейки RAM 500 в регистр2	1 500 2
сложить регистр1 и регистр2 в регистр3	3 1 2 3
выгрузить регистр3 в ячейку RAM 100	2 3 100

- 1 – код команды загрузить, 2 – код команды выгрузить, 3 – ... сложить

Проблемы при программировании в кодах

- Надо помнить о ячейках памяти и регистрах (какие заняты, какие нет)
- Коды команд плохо запоминаются
- Отсутствует переносимость: в разных архитектурах – разные наборы команды, коды команд, наборы регистров

Ассемблер

- Ассемблер – транслятор (программа) из текста на языке ассемблер в двоичные коды.
- У команд и ячеек памяти есть символьные имена, которые легко запомнить.
- Программа:

загрузить из ячейки RAM 300 в регистр1	load data1, r1
загрузить из ячейки RAM 500 в регистр2	load data2, r2
сложить регистр1 и регистр2 в регистр3	add r1, r2, r3
выгрузить регистр3 в ячейку RAM 100	store r3, data3
	.data data1 300 data2 500
	.data data3 100

Какие проблемы остались?

Ассемблер

- Ассемблер – транслятор (программа) из текста на языке ассемблер в двоичные коды.
- У команд и ячеек памяти есть символьные имена, которые легко запомнить.
- Программа:

загрузить из ячейки RAM 300 в регистр1	load data1, r1
загрузить из ячейки RAM 500 в регистр2	load data2, r2
сложить регистр1 и регистр2 в регистр3	add r1, r2, r3
выгрузить регистр3 в ячейку RAM 100	store r3, data3
	.data data1 300 data2 500
	.data data3 100

Какие проблемы остались? Переносимость!

Язык Высокого Уровня (ЯВУ)

```
1  int a = 3;
2  int b = 5;
3  int c = a + b;
```

- Компилятор – транслятор (программа) из текста на ЯВУ переводит в текст на языке ассемблера.
- Подбирает команды, ячейки памяти и номера регистров так, чтобы программа быстро выполнялась и занимала минимум памяти (за несколько проходов по тексту).

Программа –> [Компилятор] –> [Ассемблер] –> Исполняемый файл

Язык C

- Bell Labs. Задача: создать переносимую ОС (должна работать на разных архитектурах).
- Язык для ОС Unix. Язык C (“Write once, compile everywhere!”). Деннис Ритчи. 197X.
- Если для новой архитектуры существует (в современном мире — обычно да) компилятор языка C, то программу не надо переписывать, а надо просто перекомпилировать этим компилятором.

Язык C

- Bell Labs. Задача: создать переносимую ОС (должна работать на разных архитектурах).
- Язык для ОС Unix. Язык C (“Write once, compile everywhere!”). Деннис Ритчи. 197X.
- Если для новой архитектуры существует (в современном мире — обычно да) компилятор языка C, то программу не надо переписывать, а надо просто перекомпилировать этим компилятором.

На самом деле нет: в стандартную библиотеку языка C не входит графика, сеть и т.д.

Программа из нескольких файлов

Зачем программу разбивают на части?

Части (на примере шахмат):

- функции (подпрограммы): проверить корректность хода, ввести ход, вывести доску, ...
- файлы: вместе сгруппированы функции, решающие близкие задачи (контроль правил шахмат, графический интерфейс, компьютерный игрок)

Программа из нескольких файлов

Зачем программу разбивают на части?

- Небольшую часть проще понять.
- Небольшую часть проще тестировать и отлаживать.
- Разные части могут разрабатывать разные люди.
- Отдельные части проще использовать в новых проектах.

Части (на примере шахмат):

- функции (подпрограммы): проверить корректность хода, ввести ход, вывести доску, ...
- файлы: вместе сгруппированы функции, решающие близкие задачи (контроль правил шахмат, графический интерфейс, компьютерный игрок)

Файлы

```
1  //main.c
2  int main() {
3    int a = 3; int b = 5;
4    int c = a + b;
5    int d = sum(c, b);
6    return 0;
7  }
```

```
1  //util.c
2  int sum(int a, int b) {
3    return a + b;
4  }
```

Построение программы (build)

- Скомпилировать (компилятор + ассемблер) main.c –> main.o (объектный файл)
- Скомпилировать util.c –> util.o
- Слинковать [Линкер] main.o, util.o –> main.elf

Линкер

main.o

```
1 300 1
1 500 2
3 1 2 3
2 3 100
call sum
...
```

sum.o

```
func sum
43 500 2
4 1 2 3
4 3 100
...
```

В объектном файле – имена функций не заменены на адреса.

Линкер

main.elf

```
[20] 1 300 1
[24] 1 500 2
[28] 3 1 2 3
[32] 2 3 100
[36] 13  50
...
[50] 43 500 2
[54] 4 1 2 3
[58] 4 3 100
...
```

Линкер (линковщик, компоновщик) должен склеить файлы вместе и заменить вызовы функции по имени на вызовы по адресу (процессор понимает только числа – адреса).

Build

- Файлы компилируются независимо друг от друга (когда один файл компилируется, то информация из другого не используется). Затем вместе линкуются.
  - Компиляция более вычислительно сложный процесс чем линковка (подбор команд, регистров, ячеек памяти)
  - Линковка относительно вычислительно простой процесс. Два прохода по исполняемому файлу: 1. составить таблицу имя функции <–> адрес; 2. заменить вызовы по имена на адрес из таблицы.
- Обычо программист за один “шаг разработки” меняет не все файлы в проекте
  - Изменили исходный код функции sum
  - Перекомпилировали только util.c (с main.o ничего не делали)
  - Произвели линковку util.o и main.o

Типовые ошибки линковки

- undefined reference (есть вызов, нет функции)
- multiple definition (несколько функций с одинаковым именем)

Компилятор языка C не помещает в объектный код информацию (количество, типы) о параметрах функции, а линкер не проверяет соответствие между вызовом и определением функции (definition).

```
1  //main.c
2  int main() {
3    int a = 3; int b = 5;
4    int c = a + b;
5    int d = sum(c);
6    return 0;
7  }
```

```
1  //util.c
2  int sum(int a, int b) { // definition
3    return a + b;
4  }
```

Ошибка времени выполнения: sum возьмет из памяти произвольное число в качестве второго параметра.

Заголовочные файлы

Решение: поместить объявление (declaration) функции в заголовочный файл

```
//main.c
#include "util.h"
int main() {
  int a = 3; int b = 5;
  int c = a + b;
  int d = sum(c); // compilation error
  return 0;
}
```

```
//util.c
#include "util.h"
int sum(int a, int b) { // definition
  return a + b;
}
```

```
//util.h
int sum(int a, int b); // declaration
```

Заголовочные файлы

- Директива include вставляет содержимое файла util.h в исходный код.
- Результат: на стадии компиляции будут выявлены несоответствия между объявлением и определением, между вызовом и объявлением => между вызовом и определением.

Программа –> [Препроцессор] –> [Компилятор] –> [Ассемблер] –> Объектный файл