Arquitetura de Computadores

MIEI – 2017/18

DI-FCT/UNL

A equipa

Teóricas

Vitor Duarte

Práticas

João Lourenço

Hervé Paulino

Cecília Gomes

Informações

- **Tudo está no CLIP. NÃO ESQUECER!**
  - Horário, objetivos, programa, bibliografia, avaliação, datas de testes, etc.....
- Atenção aos avisos e ao seu email @campus.fct.unl.pt
- Pode sempre contactar os docentes
  - Pessoalmente
  - piazza.com/fct.unl.pt/spring2018/ac/home
  - email

AC - Resumo

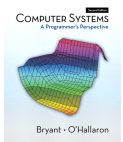
- O tema:
  - Compreender funcionamento dos computadores e como estes conseguem executar os programas
  - Estudar a organização e funcionamento do CPU, da memória e dos periféricos
- Pré-requisitos:
  - Saber as matérias de IP e SL
  - E ainda: **ter curiosidade e não ser preguiçoso/a**
- Esforço:
  - 9 créditos ECTS → ≈ **12h/semana**

Teóricas e Práticas ?

- Teóricas:
  - Apresentar os conceitos e discutir o porquê, vantagens e desvantagens
- Práticas:
  - Praticar o uso dos conceitos e experimentar → compreender melhor, saber fazer, …
- Testes/exame: avaliam ambas as componentes

Bibliografia

- Computer Systems: A programmer's perspective, **2.Ed.**, Bryant & O'Hallaron
- The C Programming Language, 2.Ed., Kernighan & Ritchie
- Acetatos, apontamentos, manuais, ...
- outros documentos, *sites*, ...



Avaliação

- 5 mini trabalhos individuais (com eventual defesa)
  - 3 trabs >= 9,5 dá frequência
  - NP = média dos 4 melhores
  - Se freq. em 15/16 ou 16/17 está dispensado
- 2 testes ou exame dá NT
- Nota final:
  - Se NT>=8,5: NF=80% NT+ 20% NP
  - Se NT<8,5: NF=NT
- Qualquer fraude leva ao chumbo de todos os envolvidos

Como reprovar (escolha uma! ;-)

- Não acompanhar as teóricas
  - Julgar que percebem tudo só pelos acetatos
- Não acompanhar as práticas
  - Julgar que basta ver os programas já feitos
- Estudar a matéria das aulas, fazer os exercícios e tirar todas as dúvidas (sem recorrer aos docentes), etc... *na véspera do prazo do trabalho, do teste ou do exame*

No estudo...

- **É necessário trabalhar BEM! ...**
  ... para não trabalhar muito.
- Não se atrase. Comece cedo, para ter o máximo de ajuda dos docentes.
- Recomenda-se, concentração
  - Evitar distrações (Facebook, email, sms, conversa fiada, ...)
  - Use várias estratégias de estudo (não leia só os slides)
    - Ler, escrever, falar, ouvir, pensar, ver/rever, ...

Objectivos

- Geral:
  - Conhecimento!...
  - ...competência e produtividade
  - ...vantagem competitiva no mercado de trabalho
- Específicos de AC:
  - Organização e funcionamento interno dos vários componentes do computador
  - Níveis de abstração, interfaces de programação, mecanismos de execução, incluindo os suportados pelo *hardware*
  - Diferente SO, diferentes linguagens, linha de comandos

AC no curso (u.c. relacionadas)

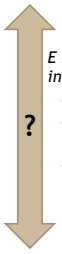
- 1º semestre:
  - Introdução à Programação
  - Sistemas Lógicos
- 2º semestre:
  - **Arquitetura de Computadores**
- Seguintes:
  - Fund. Sistemas de Operação, Ling. Ambientes de Programação, Redes de Computadores, Concorrência e Paralelismo,...
  - etc...
  - etc...
  - etc...

Programa de AC

- Componentes e funcionamento do computador
  - Arquitetura de Von Neumann
- Organização interna do processador (CPU)
  - *Instruction Set Architecture*. A linguagem máquina e o *assembly*
- Sistema de entradas e saídas, e os tipos de periféricos
- Hierarquia das unidades de memória
- Metodologia e prática laboratorial:
  - Programação em C e *assembly*, baseada na arquitetura do tipo PC (Intel 32bits)

Da programação à execução

- Java – modelo de programação e execução



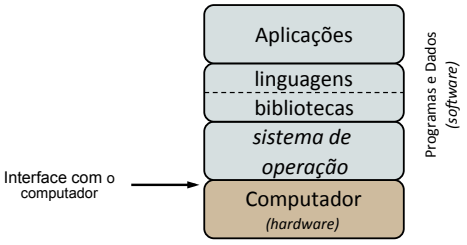
E no meio? Como transformar, traduzir, interpretar, executar...??

- *Vários níveis de abstração*
- *Cada nível de abstração oferece uma interface*
- *Facilita o nível superior e usa o nível inferior*

- Sistemas Lógicos/Electrónica – modelo de execução

Níveis nos sistemas informáticos

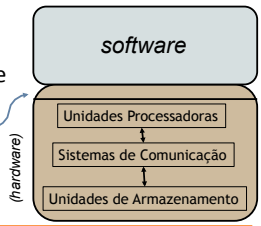
- Num sistema de computação:



- *Podemos detalhar em mais níveis …*

O tema

- Principais atividades num sistema de computação:
  - Computação
  - Armazenamento
  - Comunicação
- Suportadas pelo hardware
  - Via uma interface: *Instruction Set Architecture* (ISA)



Porquê AC …

- O Eng. Informático tem de:
  - Cumprir as especificações
  - Correção: produzir resultados corretos
  - Desempenho: reduzir o tempo de execução
  - Compromissos Custo/Realização: esforço de desenvolvimento vs. Tempo vs. custos
  - Dominar os produtos e as tecnologias

Para …

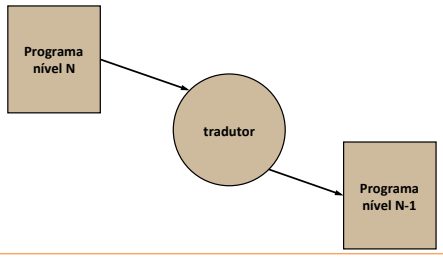
- Compreender melhor as abstrações e funcionalidades existentes nos vários níveis
- Desenvolver aplicações mais eficientes e fiáveis
  - Ao saber mais sobre o sistema que executa a aplicação, pode ser um melhor programador:
    - Tirar melhor partido do computador e dos vários níveis
    - Escrever programas mais fiáveis e rápidos
- Preparar para as cadeiras posteriores

Cada nível oferece

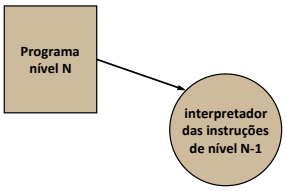
- Interface
  - oferece um conjunto de “operações”
  - define uma “linguagem” e abstrações suportadas
- Transparência
  - oferece uma “nova máquina” (virtual)
  - esconde do nível superior a implementação das suas operações/abstrações

Execução de programas por tradução

- Os programas são transformados por outros programas

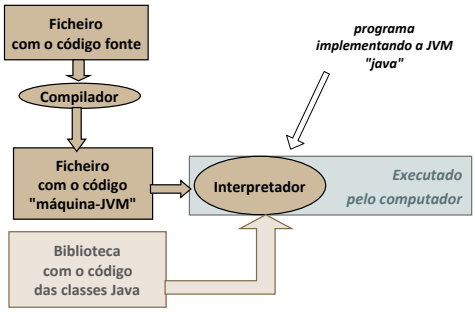


Execução de programas por interpretação

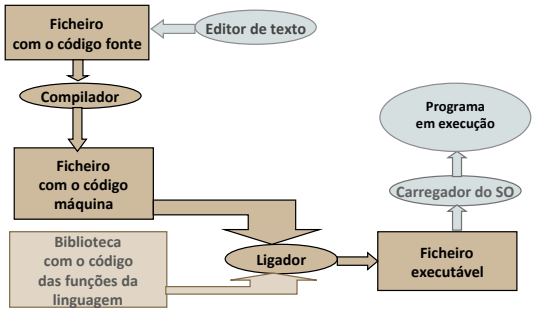


- o *hardware* do computador é sempre o interpretador final

Interpretação: Exemplo do Java



Passos típicos para execução nativa no hardware



Mais Motivação

- Muitas disciplinas fazem ênfase na abstração
  - Exemplos: Tipos de dados abstractos, Classes/Objetos, Análise de complexidade
- A abstração é boa, mas convém não esquecer a realidade!
- Estas abstrações têm limitações
  - Especialmente quando há "*bugs*" ou o desempenho não satisfaz
  - É preciso compreender as implementações que as suportam

Compreender o funcionamento do computador

- Conhecer *assembly* ajuda a perceber o modelo de execução ao nível da máquina
  - Provavelmente, nunca irão escrever um programa em "assembly". Os compiladores são normalmente melhores
- Tal ajuda a compreender:
  - Comportamento de programas na presença de "bugs"
    - O modelo da linguagem de alto nível deixa de ser aplicável
  - Aumento do desempenho de um programa
    - Perceber os motivos da ineficiência de um programa
  - Implementação de software de sistema
    - Compiladores, linguagens e bibliotecas
    - Sistema de operação

Exemplo #1

`int`'s não são inteiros, **`float`**'s não são reais

- Exemplos:
  - $x^2 \geq 0$?
    - float's: Sim ... (em princípio ☺)
    - int's:
      - 40000 × 40000 → 1600000000
      - 50000 × 50000 → 2500000000 ?
  - Será (x + y) + z = x + (y + z)?
    - int's: Sim ... (em princípio ☺)
    - float's:
      - $(10^{20} + (-10^{20})) + 3{,}14 \rightarrow 3{,}14$
      - $10^{20} + (-10^{20} + 3{,}14) \rightarrow 3{,}14$?

A importância das **memórias**

- A memória não é ilimitada
  - Pode ter de ser reservada e gerida
  - Muitas aplicações dependem da quantidade e do desempenho da memória
- Os "*bugs*" relacionados com referências à memória são especialmente perniciosos
  - Exemplo: em C e C++ o programador pode corromper a memória
  - Os efeitos aparecem muitas vezes longe no espaço e no tempo
- O desempenho da memória não é uniforme
  - Depende da memória "*Cache*", da memória real e da memória virtual. Tal pode afectar imenso a execução de um programa
  - A adaptação do programa às características do sistema de memória pode levar a grandes aumentos de velocidade

Exemplo #2

- Exemplo de um "*bug*" numa referência a memória

```
int main()
{
  float d = 3.14;
  float a[2];

  a[2] = -1.0;  /* Referencia "out of bound" */
  printf("d = %f\n", d);
}
```

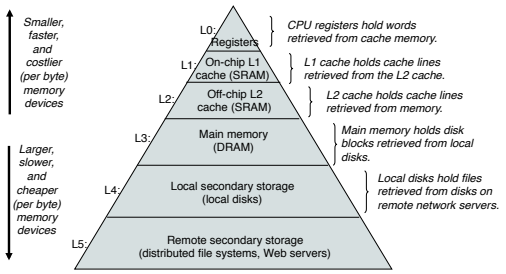
Possíveis resultados:

`-1.000000` `3.140000` *Outros...*

(O programa pode também terminar com um erro: "*Segmentation fault*." ou "Abort". O comportamento varia com a arquitetura, o SO e o compilador.)

Hierarquia de níveis de memória

- Quanto maior é capacidade maior é o tempo de acesso



Memórias e o desempenho

- O desempenho de um programa não depende só da complexidade do algoritmo
  - Facilmente se têm tempos de execução com relações de 100:1 dependendo da forma como o código é escrito
  - É possível optimizar a múltiplos níveis: algoritmo, representações dos dados, funções e ciclos
- É preciso compreender a arquitetura e as ferramentas de desenvolvimento
  - Como é que os programas são compilados e executados
  - Como medir o desempenho e identificar os problemas

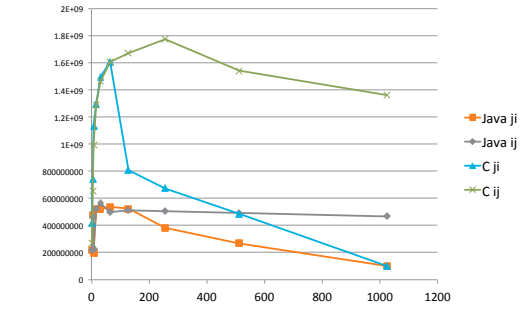
Exemplo #3

- Desempenhos diferentes de acordo com o padrão de acesso à memória
- Somar todos os elementos de uma matriz:
  - Múltiplas formas de percorrer a matriz (**nxn**). Exemplos:

```
/* ij */
sum = 0.0;
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
      sum += a[i][j];
  }
}
```

```
/* ji */
sum = 0.0;
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
      sum += a[i][j];
  }
}
```

Desempenho (somas/s para nxn)



Entradas/saídas e concurrência

- Os computadores fazem mais do que executar instruções de um programa
  - O sistema de entradas/saídas (I/O) é crítico para a fiabilidade e o desempenho de um programa
- Os programas comunicam
  - Na mesma máquina e através de redes
  - Muitas problemas aparecem nestes casos
    - Atividades simultâneas e independentes
    - Comunicação não fiável
    - Máquinas envolvidas podem usar representações de dados diferentes

Representação de informação

- Toda a informação no sistema é um conjunto de bits
- O mesmo conjunto de bits pode ser:
  - Um inteiro
  - Um real representado em vírgula flutuante
  - Uma sequência de caracteres
  - Um conjunto de instruções máquina
- Só se distingue pela forma como os bits são interpretados

Exemplo #4

- O código fonte do programa hello.c

```
#include <stdio.h>
int main()
{
      printf("hello,world\n");
}
```

O programa foi criado por um editor de texto. O ficheiro com código é uma sequência de bits. Interpretando cada byte do seu conteúdo obtemos:

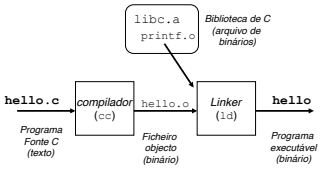
cada byte interpretando como inteiro:
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46 104 62 10 105 …
cada byte interpretando como carácter (norma ISO/ASCII):
i n c l u d e < s t d i o . h > i n t…

Mas ainda não é um programa executável pelo computador!

Transformações da linguagem C

- Os programas são transformados por outros programas
- Exemplo da compilação de programas em C:

```
cc –o hello hello.c
```



Hello.o

- Interpretando o conteúdo de hello.o obtemos para main:
  - Inteiro hex: 55 89 e5 83 e4 f0 83 ec 10 c7 04 24
  - Inteiro dec: 85 -119 -27 -125 -28 -16 -125 -20 16 -57 4 36
  - interpretando como caracteres (norma ISO/ASCII):
    - U ? ? ? ? ? ? ? ? ? $
  - Interpretando como instruções (ia32):

```
main:
     0: 55                      push   %ebp
     1: 89 e5                   mov    %esp,%ebp
     3: 83 e4 f0                and    $0xfffffff0,%esp
     6: 83 ec 10                sub    $0x10,%esp
     9: c7 04 24 00 00 00 00    movl   $0x0,(%esp)
     10: e8 fc ff ff ff         call   11 <main+0x11>
     15: b8 00 00 00 00         mov    $0x0,%eax
     1a: c9                     leave
     1b: c3                     ret
```

Transformações da linguagem Java

- Java é compilado para um código máquina virtual

```
javac Hello.java
```

- Para executar indica-se o nome da classe um programa que interpreta esse código

```
java Hello
```