1 Tipos de memória

Os computadores que utilizamos hoje em dia se baseiam em circuitos elétricos e no fluxo de corrente elétrica que passa por eles, utilizando um componente chamado transistor. Os transistores são agrupados em circuitos lógicos que estão presentes em praticamente todos os equipamentos eletrônicos da atualidade. Com eles, é possível liberar ou interromper a passagem de corrente elétrica. Se nenhuma tensão for aplicada ao terminal de controle, não há circulação de corrente elétrica, o que confere ao transistor duas propriedades: amplificação de sinal elétrico e controle do fluxo da corrente, como se fosse um botão de ligado/desligado. Os processadores contam hoje com bilhões de transistores ligados entre si, formando circuitos capazes de fazer cálculos simples ou extremamente complexos.

Bit é uma unidade que representa a passagem ou não de corrente. O bit 0 indica uma passagem quase nula, e o bit 1 indica a passagem de uma corrente alta. Um bit é a menor unidade de representação entendida pelos processadores, e também utilizada nas memórias de uma computador. Por representar apenas dois valores, costumam ser agrupados em unidades maiores. O termo bit significa **BI**nary digi**T** (dígito binário).

Um **byte**, em geral, representa uma sequência de $2^3 = 8$ bits consecutivos. Como cada bit representa dois valores, 8 bits consecutivos podem representar até $2^8 = 256$ valores diferentes. Bytes costumam também ser acompanhados pelos prefixos *kilo*, *mega*, *giga*, etc. Um kilobyte (ou apenas **1 kB**) representa $2^{10} = 1.024$ bytes. De forma análoga, um megabyte (**1 MB**) representa $2^{10}kB = 2^{20}$ bytes = 1.048.576 bytes. Um gigabyte (**1GB**) representa $2^{30} = 1.073.741.824 \approx 1$ bilhão de bytes.

A **memória** é o conjunto de todos os dispositivos que permitem que o computador guarde dados, de forma permanente ou temporária. A unidade básica das memórias que utilizamos hoje em dia é o *bit*. As memórias **voláteis** são aquelas que perdem seus dados com a ausência de energia, ou seja, quando o computador é desligado (exemplos: RAM, registradores, CACHE). As memórias **não voláteis**, como o HD, não perdem seus dados na ausência de energia. Podemos dividi-las também em memória **principal** ou **secundária**. Uma memória principal pode ser acessada diretamente pelo processador. RAM, CACHE e registradores são exemplos de memória principal voláteis, e a ROM é uma memória principal não volátil.

Os programas salvos no computador precisam ser armazenados em uma memória não volátil, pois não queremos que eles sejam apagados quando desligamos o computador. Mas para serem executados, precisam ser acessados diretamente pelo processador, e por isso só são executados quando alocados em uma memória principal. A memória ROM seria ideal para isso, por ser uma memória principal não volátil. Entretanto, como ela é muito cara, costumamos armazenar os programas em um HD, que é uma memória secundária. Isso significa um programa jamais poderá ser acessado diretamente pelo processador enquanto estiver no HD. Toda vez que quisermos executá-lo, ele primeiro precisa ser carregado na memória RAM.

A capacidade de armazenamento das memórias é medida em bytes. Uma memória

RAM de 8GB é capaz de armazenar $8 \times 2^{30} \times 2^3 \approx 68$ bilhões de bits. Mas quando encontramos lemos que um computador tem 32 bits, isso significa que ele possui registradores de 32 bits, ou seja, 4 bytes. Geralmente, uma linguagem de programação irá armazenar números inteiros de no máximo 4 bytes em um computador de 32 bits. Por isso, um Sistema Operacional de 32 bits, projetado para rodar em um computador de 32 bits, costuma reconhecer apenas memórias RAM de até 4GB, pois existe um registrador de 32 bits responsável por armazenar os endereços da memória que serão acessados, e só seria possível armazenar 2^{32} números diferentes neste registrador, ou seja, 2^{32} bits $= 2^2 \times 2^{30}$ bits $= 4 \times 2^{30}$ bits $= 4$GB.

2 Persistência dos Dados

Quando um programa é carregado na memória e executado pelo Sistema Operacional, os dados das variáveis criadas no programa são armazenados na memória RAM, junto com o código executável do programa. Se fechamos o programa e depois o carregamos novamente na memória, os dados criados anteriormente são todos perdidos. Porém, algumas aplicações exigem que as informações cadastradas pelo usuário não sejam perdidas. Um sistema acadêmico, por exemplo, não pode perder as notas dos alunos quando faltar energia no servidor. Já uma aplicação web pode armazenar informações a respeito da navegação do usuário (como configurações personalizadas, idioma, etc). Para que os dados presentes na memória RAM não sejam perdidos, costumamos salvá-los em arquivos em alguma memória não volátil (geralmente, no HD).

Quando há muita informações para armazenar, podemos utilizar um sistema gerenciador de bancos de dados. Além das informações propriamente ditas, o sistema vai criar arquivos que armazenam as relações entre os dados e facilitar o acesso a eles. Quando uma aplicação web é executada a partir de um servidor remoto, os dados do usuário podem ser salvos localmente no HD do próprio usuário, na forma de *cookies*. Um *cookie* é apenas um arquivo texto com tais informações que a aplicação salva no computador do próprio usuário.

2.1 Arquivos Textos

A forma mais simples de realizar a **persistência dos dados**, ou seja, fazer com que eles continuem salvos após o programa ser encerrado, é salvar as informações em arquivos textos. Um arquivo texto é um arquivo que pode ser aberto num editor de texto qualquer, e que contém apenas caracteres que podem ser lidos por um humano (com extensões como *.txt*, *.py*, *.html*, etc). É como se o programa imprimisse todos os dados cadastrados, e que a saída da impressão fosse salva em um arquivo texto ao invés de ser exibida na tela. Antes de imprimir em um arquivo, porém, é necessário inicializar o arquivo na memória. Em Python, fazemos isso através da função **open()**.

```
1 f = open("nome_do_arquivo.txt", "w")
```

O primeiro parâmetro é o nome do arquivo a ser carregado, e o segundo indica o modo em que ele será aberto:

- **"w"** (write): Para escrever dados no arquivo. Apaga o arquivo e cria um novo caso ele já exista.
- **"a"** (append): Para adicionar dados ao arquivo. Caso ele já exista, as informações serão adicionadas ao final do arquivo.
- **"r"** (read): Para ler informações de um arquivo. O arquivo não é modificado, e a função dá erro caso o arquivo não exista.
- **"r+"** (read/write): Para permitir a leitura e escrita simultâneas no arquivo.

No exemplo acima, a variável `f` indica um lugar da memória RAM reservado para manipularmos o arquivo. Para salvar textos no arquivo `f`, usamos o métodos `write()`

```
1 f.write("String a ser salva no arquivo.")
```

Os dados só serão efetivamente salvos no HD quando o arquivo for fechado:

```
1 f.close()
```

Se o programa for encerrado antes do método `f.close()`, o arquivo não será salvo no disco e os dados serão perdidos. Exemplo:

```
1 f = open("teste.txt", "w")
2 f.write("Ola!")
3 f.write("Tudo bem?")
4 f.close()
```

Neste caso, será criado um arquivo "teste.txt"no HD com os seguintes caracteres:

```
Ola!Tudo bem?
```

Geralmente, usamos um \n para identificar cada linha:

```
1 f = open("teste.txt", "w")
2 f.write("Ola!\n")
3 f.write("Tudo bem?")
4 f.close()
```

```
Ola!
Tudo bem?
```

Para salvar algum dado que não seja uma String, é preciso convertê-lo primeiro:

```
1 f = open("teste.txt", "w")
2 x = 23
3 y = 34
4 f.write(str(x))
5 f.write(str(y))
6 f.write(str(x+y))
7 f.close()
```

Neste caso, o arquivo irá conter:

```
2333457
```

Verifique no seu computador qual seria a saída do arquivo no seguinte caso:

```
1 f = open("teste.txt", "w")
2 x = str(23)
3 y = str(34)
4 f.write(x)
5 f.write(y)
6 f.write(x+y)
7 f.close()
```

Por padrão, as informações são salvas como texto. Entretanto, como sabemos que uma memória armazena uma sequência de bits, esse texto é codificado como uma sequência de bits. Cada caractere impresso no arquivo é na verdade um número que ocupa um ou mais bytes do arquivo. Para isso, temos que escolher uma codificação para o arquivo. Até algum tempo, a mais comum era a Tabela ASCII (*American Standard Code for Information Interchange*), que atende bem os caracteres do teclado americanos. Esse padrão define a correspondência entre símbolos e números de 0 a 127:

```
1 for i in range(32,128, 5):
2     for j in range(5):
3         print("{:3d} - {}\t".format(i+j, chr(i+j)),end="")
4     print()
```

```
 32 -          33 - !         34 - "         35 - #         36 - $
 37 - %        38 - &         39 - ’         40 - (         41 - )
 42 - *        43 - +         44 - ,         45 - -         46 - .
 47 - /        48 - 0         49 - 1         50 - 2         51 - 3
 52 - 4        53 - 5         54 - 6         55 - 7         56 - 8
 57 - 9        58 - :         59 - ;         60 - <         61 - =
 62 - >        63 - ?         64 - @         65 - A         66 - B
 67 - C        68 - D         69 - E         70 - F         71 - G
```

```
 72 - H        73 - I        74 - J        75 - K        76 - L
 77 - M        78 - N        79 - O        80 - P        81 - Q
 82 - R        83 - S        84 - T        85 - U        86 - V
 87 - W        88 - X        89 - Y        90 - Z        91 - [
 92 - \        93 - ]        94 - ^        95 - _        96 - ‘
 97 - a        98 - b        99 - c       100 - d       101 - e
102 - f       103 - g       104 - h       105 - i       106 - j
107 - k       108 - l       109 - m       110 - n       111 - o
112 - p       113 - q       114 - r       115 - s       116 - t
117 - u       118 - v       119 - w       120 - x       121 - y
122 - z       123 - {       124 - |       125 - }       126 - ~
```

A função **chr()** transforma um índice da tabela ASCII em seu símbolo correspondente. Quando um editor de texto encontra a sequência de bits correspondente a $(97)_{10}$, será exibido na tela o símbolo “a”. Os caracteres 0 a 31 na tabela ASCII são caracteres de controle, e os demais são os caracteres comuns no teclado americano (não há caracteres acentuados).

O Unicode é um conjunto de caracteres que inclui os símbolos de todas as línguas humanas (chinês, hieróglifos usados pelos egípicios, etc). Para representar os caracteres do Unicode, foram desenvolvidos diversas formas de codificação. Uma delas é a UTF-32 (U de Universal Character Set e o TF de Transformation Format - usando 32 bits). Essa forma utiliza sempre 32 bits (ou 4 bytes) para codificar um caractere. Esse formato não é muito utilizado hoje pois seus arquivos tornam-se muito grandes. O UTF-16 exige ao menos 2 bytes (16 bits) para codificar cada caractere, criando arquivos menores em geral. Esse padrão é utilizado na linguagem Java e em sistemas operacionais como o Microsoft Windows e o Mac OS X. Já o padrão mais utilizado para codificar páginas na Internet atualmente é o UTF-8, que usa pelo menos 8 bits para representar cada caractere. Python 2.X utilizava a codificação ASCII, enquanto Python 3.X passou a utilizar UTF-8.

Existe uma forma de escrever em um arquivo texto sem ter que se lembrar de fechá-lo:

```
1 with open("arquivo.txt", "w") as arq:
2     arq.write("Minha terra tem palmeiras\n")
3     arq.write("onde canta o sabia\n")
4     arq.write("sen(A+B) = sen(A).cos(B) + sen(B).cos(A)\n")
```

Para a leitura dos dados, temos 3 métodos:

- O método **readline()** lê uma única linha do arquivo. Quando for chamado novamente, lê a linha seguinte. Considere que o arquivo a ser lido contém os seguintes caracteres:

```
Ola!
Tudo bem?
```

Se executarmos o código a seguir:

```
1 with open("arquivo.txt", "r") as arq:
2     x = arq.readline()
3     y = arq.readline()
```

A variável x irá conter a String "Olá!\n" e a variável y irá contar a String "Tudo bem?".

- O método `readlines()` retorna uma lista com cada linha lida do arquivo. Se executarmos o código a seguir:

```
1 with open("arquivo.txt", "r") as arq:
2     x = arq.readlines()
```

A variável x irá conter a lista ["Olá!\n", "Tudo bem?"].

- O método `read()` vai ler todo o arquivo de uma só vez em uma única String. Se executarmos o código a seguir:

```
1 with open("arquivo.txt", "r") as arq:
2     x = arq.read()
3     y = x.splitlines()
```

A variável x irá conter a String "Olá!\nTudo bem?" e a variável y irá conter a lista ["Olá!", "Tudo bem?"] (sem os fins de linhas).

Outra forma de percorrer as linhas de um arquivo aberto para leitura é:

```
1 with open("arquivo.txt", "r") as arq:
2     for linha in arq:
3         x = linha.strip()
4         print(x)
```

O método `strip()` irá retirar o fim de linha da String. Antes de abrir o arquivo, podemos verificar se ele existe com o método `os.path.isfile("nome_do_arquivo.txt")`.

2.2 Arquivos Binários

Num arquivo binário, as informações (números inteiros, doubles, listas, etc) não são transformados para sequências de caracteres. Ao invés disso, são salvos exatamente com da forma em que são representados na memória. Ao contrário de um arquivo texto, não podemos abri-los num editor de texto para conferir se suas informações estão realmente lá. Entretanto, não precisamos nos preocupar em transformar cada dado para String durante a escrita, nem tratar as Strings na hora da leitura.

Para abrir um arquivo binário, adicionamos um 'b' no modo de abertura do arquivo:

```
1 with open("arquivo.txt", "wb") as arq:
2     # Salvar algo no arquivo...
```

Analogamente, podemos abrir o arquivo com "ab", "rb", "r+b". Um arquivo texto, na verdade, pode ser aberto com "wt", "rt", etc. Entretanto, o padrão é abrir o arquivo no modo texto, então não é necessário informar o 't'.

Uma vez aberto, podemos adicionar informações ao arquivo com `f.write()`. Entretanto, teríamos que adicionar as informações de *byte* em *byte*, o que não seria tão viável. Ao invés disso, utilizamos módulos que nos ajudam a manipular a leitura e escrita em arquivos binários. Um deles é o módulo *pickle*:

```
1  import pickle
2
3  with open("teste.bin", "wb") as f:
4      x = 35
5      y = 18
6      z = x+y
7      a = 3.14159
8      s = "Olaaaaa"
9
10     pickle.dump(x, f) # salvando um dado
11     pickle.dump(y, f)
12     pickle.dump(z, f)
13     pickle.dump(a, f)
14     pickle.dump(s, f)
```

Se abrimos o arquivo "teste.bin" num editor de texto, só conseguimos visualizar o "Olaaaaa". Os dados serão lidos na mesma ordem em que foram salvos:

```
1  import pickle
2
3  with open("teste.bin", "rb") as f:
4      x1 = pickle.load(f) # lendo um dado
5      x2 = pickle.load(f)
6      x3 = pickle.load(f)
7      x4 = pickle.load(f)
8      x5 = pickle.load(f)
9      x6 = pickle.load(f) # Esta linha vai dar erro!!
10
11     print(x1)
12     print(x2)
13     print(x3)
14     print(x4)
15     print(x5)
```

A linha 9 do código acima dá erro pois apenas 5 dados foram escritos, então apenas 5 dados podem ser lidos. Retirando a linha, a saída do código será:

```
35
18
53
3.14159
Olaaaaa
```

O módulo *pickle* também permite a serialização dos dados, transformando-os em sequência de *bytes*. Com isso, até mesmo uma lista/tupla/dicionário (que não é salva sequencialmente na memória) pode ser salva num arquivo binário com um único comando:

```
1 import pickle
2
3 l = [2, 5, 9, "Olaaaaa", 3.14159, (5,8)]
4 with open("teste.bin", "wb") as f:
5     pickle.dump(l,f)
```

Da mesma forma, podemos ler a lista/tupla/dicionário com um único comando:

```
1 import pickle
2
3 with open("teste.bin", "rb") as f:
4     l = pickle.load(f)
5     for item in l:
6         print(item)
```

A saída do código acima é:

```
2
5
9
Olaaaaa
3.14159
(5, 8)
```