



UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
DISCIPLINA: Arquitetura de Computadores
PROFESSOR: Ewerton Monteiro Salvador

TRABALHO COM LINGUAGEM ASSEMBLY

O programa especificado abaixo deverá ser implementado utilizando-se a linguagem Assembly, no Windows ou no Linux (versão 32 bits). O trabalho será individual e deverá ser enviado pelo SIGAA até as **23:59h do dia 28/11/2022**.

ESPECIFICAÇÃO

Escreva um programa que receba como entrada uma imagem no formato *bitmap* (extensão .bmp) e produza como saída uma cópia da imagem recebida tendo uma de suas cores básicas (azul – código 0, verde – código 1, ou vermelho – código 2) aumentada por um determinado valor (0 a 255).

Por exemplo, suponha que um usuário forneça a imagem “catita.bmp” abaixo:



Suponha também que o usuário tenha selecionado “catita2.bmp” como nome do arquivo de saída, tenha selecionado a cor azul (código 0), e tenha definido o valor 50 para aumento de intensidade dessa cor. O programa deverá produzir como saída, no arquivo “catita2.bmp”, a imagem abaixo:



Observações importantes:

- A imagem original e a nova imagem devem estar no mesmo diretório do arquivo executável, de modo que o usuário só precise informar o nome do arquivo sem se preocupar com o caminho do arquivo;
- O(a) aluno(a) não precisa se preocupar com tratamento de erros de entrada. Assuma que todas as entradas serão fornecidas corretamente, dentro das faixas de valores esperadas;
- Tanto no Windows quanto no Linux deverão ser utilizadas as chamadas oficiais do sistema operacional para abrir, ler, escrever e fechar **arquivos**, não sendo permitido o uso de outras bibliotecas para esse fim;
- O programa deve conter uma função que receba três parâmetros: 1) o endereço de um array de 3 bytes representando um pixel, 2) um inteiro de 4 bytes representando uma cor a ser intensificada (0 para azul, 1 para verde e 2 para vermelho), e 3) um inteiro de 1 byte representando o valor a ser somado à cor. O objetivo da função é substituir o valor de um dos bytes do array de 3 bytes recebido como entrada, de modo que o novo valor seja o original somado com o valor do terceiro parâmetro. A função não deve retornar nenhum valor;
- **Importante:** a função descrita no tópico anterior não deve permitir somas que ultrapassem o valor máximo para um byte, que é 255 (0FFH). Nos casos em que a soma ultrapassaria 255, o valor da cor deve ser definido como 255 (máximo);
- A entrada e saída de console no Windows deve utilizar as funções ReadConsole e WriteConsole da biblioteca kernel32. No Linux podem ser utilizadas as funções printf e scanf da biblioteca padrão da linguagem C, utilizando o gcc para “linkagem” do programa.

A implementação deve ser feita em Assembly **versão 32 bits** para Windows (MASM32) ou para Linux (NASM). O trabalho deve ser desenvolvido de forma individual. O código implementado deve ser original, não sendo permitidas cópias de códigos inteiros ou trechos de códigos de outras fontes (exceto quando expressamente autorizado pelo professor da disciplina). Por esse motivo, **recomenda-se enfaticamente que não haja compartilhamento de código entre os alunos da disciplina**. Os debates entre alunos devem estar restritos a ideias e estratégias, e nunca envolver códigos, para evitarem penalidades na nota relacionadas à plágio.

--- Boa sorte! ---

INFORMAÇÕES COMPLEMENTARES

Como lidar com arquivos?

Tanto no Windows como no Linux o tratamento de arquivos é similar, sendo essencialmente o mesmo utilizado em linguagens de programação de alto nível, como C:

- Solicita-se ao sistema operacional a abertura de um arquivo (em modo de leitura, de escrita ou ambos). O sistema operacional devolve um *handle*, que serve como um número de identificação do arquivo aberto para ser utilizado nas chamadas de sistemas seguintes que envolvam esse arquivo;
- O sistema operacional define um “apontador de arquivo” para todos os arquivos abertos, o qual é controlado automaticamente pelo próprio sistema operacional. A abertura de um arquivo tipicamente faz com que esse apontador seja posicionado na primeira posição (posição 0, primeiro byte) desse arquivo, e é **incrementado sempre que uma leitura ou uma escrita é realizada**. Existe uma função do sistema operacional que permite que o(a) programador(a) reposicione esse apontador de arquivo, contudo essa função não será necessário para este projeto;

- Leituras e escritas são realizadas através de chamadas de sistemas operacionais próprias. A leitura ou escrita sempre começa na posição atual do apontador de arquivo controlado pelo sistema operacional. O apontador de arquivo é incrementado ao final de uma operação de leitura ou escrita de acordo com a quantidade de bytes envolvida nessa operação;
- Por fim, arquivos devem ser fechados através de uma chamada ao sistema operacional. O fechamento do arquivo garante que dados escritos sejam corretamente gravados, além de liberar recursos do sistema operacional que foram alocados para o tratamento do arquivo.

No Windows 32 bits as chamadas de sistema relacionadas a arquivos se encontram na biblioteca **kernel32** (com constantes definidas no arquivo de cabeçalho `windows.inc`). No Linux 32 bits as chamadas de sistema relacionadas a arquivos são invocadas através da **interrupção 80h**.

Criação/Abertura de Arquivo: Windows

Realizada através da função **CreateFile**

Parâmetros:

1. Apontador (endereço) de string contendo o nome do arquivo a ser aberto (no nosso projeto, **o nome não precisa incluir o caminho de diretórios**, considerando que o arquivo .bmp estará no mesmo diretório do arquivo executável do projeto);
2. Constante de 4 bytes informando o nível de acesso desejado para o arquivo. Exemplos dessas constantes são `GENERIC_READ` e `GENERIC_WRITE`, **as quais devem ser utilizadas nesse projeto para operações de escrita ou leitura**. Uma operação de escrita e leitura pode ser alcançada através de uma operação OR entre as constantes `GENERIC_READ` e `GENERIC_WRITE`, **contudo esse tipo de operação de leitura e escrita em um mesmo arquivo não será necessária neste projeto**;
3. Constante de 4 bytes informando se o acesso ao arquivo será compartilhado ou não. Exemplos dessas constantes são 0 (zero), `FILE_SHARE_WRITE`, `FILE_SHARE_READ`, etc. Como o arquivo desse projeto não precisará de acesso compartilhado com outros programas, **essa constante deverá ser definida como 0 (zero)**;
4. Apontador para uma estrutura do tipo `SECURITY_ATTRIBUTES` (definida em `windows.inc`) contendo atributos de segurança. Esse parâmetro não será necessário nesse projeto, ou seja, **deverá ser informada aqui a constante NULL**;
5. Constante de 4 bytes especificando a necessidade de se criar ou não um novo arquivo. Exemplos dessas constantes são `CREATE_ALWAYS`, `CREATE_NEW`, `OPEN_ALWAYS`, `OPEN_EXISTING`, etc. Neste projeto, **deverá ser utilizada a opção OPEN_EXISTING para abertura do arquivo .bmp original, e CREATE_ALWAYS para a criação do arquivo .bmp de saída**, de modo que o arquivo original só seja aberto e nunca criado, e o arquivo de destino seja sempre um novo arquivo;
6. Constante de 4 bytes especificando os atributos do arquivo a ser aberto, como `FILE_ATTRIBUTE_ARCHIVE`, `FILE_ATTRIBUTE_NORMAL`, etc. Como este projeto não utilizará atributos especiais, **deverá ser utilizada a opção FILE_ATTRIBUTE_NORMAL**;
7. Um handle de 4 bytes para um arquivo que sirva de template quanto a atributos. Como este projeto não utilizará atributos especiais, **deverá ser utilizada a constante NULL**.

Retorno: um handle para o arquivo é retornado através do registrador EAX.

Ex.:

```
invoke CreateFile, addr fileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
```

```
mov fileHandle, eax
```

Criação (com abertura) de Arquivo: Linux

Realizada através da interrupção 80h

Parâmetros:

1. O registrador EAX deve receber o valor 8, referente à chamada de sistema `sys_creat`;
2. O registrador EBX deve conter um apontador (endereço) de string contendo o nome do arquivo a ser aberto (no nosso projeto, **o nome não precisa incluir o caminho de diretórios**, considerando que o arquivo .bmp estará no mesmo diretório do arquivo executável do projeto);
3. O registrador ECX deve conter as permissões do arquivo, de acordo com a convenção de permissões de arquivos utilizada pelo Linux. Por exemplo, a permissão 777 dá acesso total a arquivos (leitura, escrita e execução) para o usuário dono do arquivo, para o grupo do dono e para todos os usuários do sistema.

Retorno: um handle para o arquivo é retornado através do registrador EAX.

Ex.:

```
mov eax, 8          ; sys_creat
mov ebx, filename
mov ecx, 777
int 80h

mov fileHandle, eax
```

Abertura de Arquivo Já Existente: Linux

Realizada através da interrupção 80h

Parâmetros:

4. O registrador EAX deve receber o valor 5, referente à chamada de sistema `sys_open`;
5. O registrador EBX deve conter um apontador (endereço) de string contendo o nome do arquivo a ser aberto (no nosso projeto, **o nome não precisa incluir o caminho de diretórios**, considerando que o arquivo .bmp estará no mesmo diretório do arquivo executável do projeto);
6. O registrador ECX deve conter o modo de acesso do arquivo. Os mais comuns são 0 (read-only), 1 (write-only), e 2 (read-write);
7. O registrador EDX deve conter as permissões do arquivo, de acordo com a convenção de permissões de arquivos utilizada pelo Linux. Por exemplo, a permissão 777 dá acesso total a arquivos (leitura, escrita e execução) para o usuário dono do arquivo, para o grupo do dono e para todos os usuários do sistema.

Retorno: um handle para o arquivo é retornado através do registrador EAX.

Ex.:

```
mov eax, 5          ; sys_open
mov ebx, filename
mov ecx, 0          ; read_only
mov edx, 777
int 80h

mov fileHandle, eax
```

Leitura de Arquivo: Windows

Realizada através da função **ReadFile**

Parâmetros:

1. Handle de 4 bytes do arquivo a ser lido. Esse handle é recebido como retorno da função de abertura do arquivo;
2. Um apontador para um array de bytes onde serão gravados os bytes lidos do arquivo;
3. Um inteiro de 4 bytes indicando a quantidade de bytes máxima a ser lida do arquivo. Observe que essa quantidade máxima de bytes deve ser igual ou inferior à quantidade de bytes do array de bytes utilizado para gravação dos dados;
4. Apontador para um inteiro de 4 bytes onde será gravado a quantidade de bytes efetivamente lidos do arquivo. **Importante: quando a leitura chegar ao final do arquivo, a quantidade de bytes lida será 0, e isso será o indicativo de que você chegou no fim do arquivo;**
5. Apontador para estrutura OVERLAPPED, utilizada para acessos assíncronos ao arquivo. Como neste projeto utilizaremos acessos síncronos, **por simplicidade, esse parâmetro deve conter a constante NULL;**

Retorno: 0 se a leitura falhar, e um número diferente de zero se for bem-sucedida.

Ex.:

```
invoke ReadFile, fileHandle, addr fileBuffer, 10, addr readCount,  
NULL ; Le 10 bytes do arquivo
```

Leitura de Arquivo: Linux

Realizada através da interrupção 80h

Parâmetros:

1. O registrador EAX deve receber o valor 3, referente à chamada de sistema sys_read;
2. O registrador EBX deve conter o handle do arquivo. Esse handle é recebido como retorno da função de abertura/criação do arquivo;
3. O registrador ECX deve conter um apontador para um array de bytes onde serão gravados os bytes lidos do arquivo;
4. O registrador EDX deve conter um inteiro indicando a quantidade de bytes máxima a ser lida do arquivo. Observe que essa quantidade máxima de bytes deve ser igual ou inferior à quantidade de bytes do array de bytes utilizado para gravação dos dados.

Retorno: No registrador EAX terá a quantidade de bytes efetivamente lidos do arquivo. **Importante: quando a leitura chegar ao final do arquivo, a quantidade de bytes lida será 0, e isso será o indicativo de que você chegou no fim do arquivo**

Ex.:

```
mov eax, 3 ; sys_read  
mov ebx, [fileHandle]  
mov ecx, fileBuffer  
mov edx, 10  
int 80h
```

Escrita de Arquivo: Windows

Realizada através da função **WriteFile**

Parâmetros:

1. Handle de 4 bytes do arquivo a ser escrito. Esse handle é recebido como retorno da função de abertura do arquivo;
2. Um apontador para um array de bytes a serem gravados no arquivo;
3. Um inteiro de 4 bytes indicando a quantidade de bytes a ser gravada. Observe que essa quantidade máxima de bytes deve ser igual ou inferior à quantidade de bytes do array de bytes utilizado para gravação dos dados;
4. Apontador para um inteiro de 4 bytes onde será gravado a quantidade de bytes efetivamente escritos no arquivo;
5. Apontador para estrutura OVERLAPPED, utilizada para acessos assíncronos ao arquivo. Como neste projeto utilizaremos acessos síncronos, **por simplicidade, esse parâmetro deve conter a constante NULL**.

Retorno: 0 se a escrita falhar, e um número diferente de zero se for bem-sucedida.

Ex.:

```
invoke WriteFile, fileHandle, addr fileBuffer, 10, addr writeCount,  
NULL ; Escreve 10 bytes do arquivo
```

Escrita de Arquivo: Linux

Realizada através da interrupção 80h

Parâmetros:

6. O registrador EAX deve receber o valor 4, referente à chamada de sistema sys_write;
7. O registrador EBX deve conter o handle do arquivo. Esse handle é recebido como retorno da função de abertura/criação do arquivo;
8. O registrador ECX deve conter um apontador para um array de bytes com o conteúdo a ser gravado no arquivo;
9. O registrador EDX deve conter um inteiro indicando a quantidade de bytes máxima a ser escrita no arquivo.

Retorno: No registrador EAX terá a quantidade de bytes efetivamente escritos no arquivo.

Ex.:

```
mov eax, 4 ; sys_write  
mov ebx, [fileHandle]  
mov ecx, fileBuffer  
mov edx, 10  
int 80h
```

Fechamento de Arquivo: Windows

Realizada através da função **CloseHandle**

Parâmetros:

1. Handle de 4 bytes do arquivo a ser fechado. Esse handle é recebido como retorno da função de abertura do arquivo;

Retorno: 0 se o fechamento falhar, e um número diferente de zero se for bem-sucedido.

Ex.:

```
invoke CloseHandle, fileHandle
```

Fechamento de Arquivo: Linux

Realizada através da interrupção 80h

Parâmetros:

2. O registrador EAX deve receber o valor 6, referente à chamada de sistema `sys_close`;
3. O registrador EBX deve conter o handle do arquivo. Esse handle é recebido como retorno da função de abertura/criação do arquivo

Retorno: No registrador EAX terá um código em caso de erro.

Ex.:

```
mov eax, 6 ; sys_close
mov ebx, [fileHandle]
int 80h
```

Como lidar o formato *bitmap* (.BMP)?

Um arquivo *bitmap* (.BMP) possui uma estrutura bastante simples, composta basicamente de cabeçalhos, uma tabela de cores opcional (tipicamente para casos em que se utilize 8 bits por pixel, ou seja, uma técnica de paleta de cores), e uma lista de valores RGB, sendo um valor para cada pixel. A estrutura de um arquivo *bitmap* pode ser encontrada na imagem abaixo.

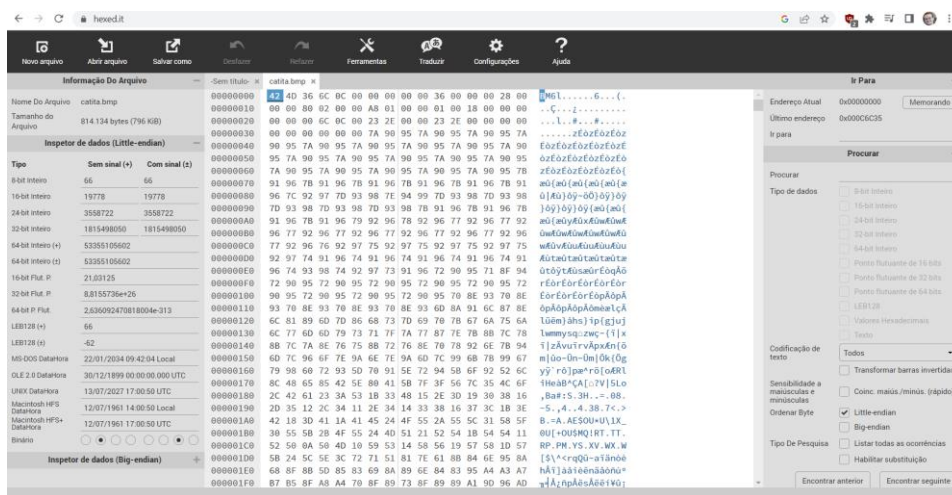
Basic BMP File Format		
Name	Size	Description
Header	14 bytes	Windows Structure: BITMAPFILEHEADER
Signature	2 bytes	'BM'
FileSize	4 bytes	File size in bytes
reserved	4 bytes	unused (=0)
DataOffset	4 bytes	File offset to Raster Data
InfoHeader	40 bytes	Windows Structure: BITMAPINFOHEADER
Size	4 bytes	Size of InfoHeader = 40
Width	4 bytes	Bitmap Width
Height	4 bytes	Bitmap Height
Planes	2 bytes	Number of Planes (=1)
BitCount	2 bytes	Bits per Pixel 1 = monochrome palette. NumColors = 1 4 = 4bit palletized. NumColors = 16 8 = 8bit palletized. NumColors = 256 16 = 16bit RGB. NumColors = 65536 (?) 24 = 24bit RGB. NumColors = 16M
Compression	4 bytes	Type of Compression 0 = BI_RGB no compression 1 = BI_RLE8 8bit RLE encoding 2 = BI_RLE4 4bit RLE encoding
ImageSize	4 bytes	(compressed) Size of Image It is valid to set this =0 if Compression = 0
XpixelsPerM	4 bytes	horizontal resolution: Pixels/meter
YpixelsPerM	4 bytes	vertical resolution: Pixels/meter
ColorsUsed	4 bytes	Number of actually used colors
ColorsImportant	4 bytes	Number of important colors 0 = all
ColorTable	4 * NumColors bytes	present only if Info.BitsPerPixel <= 8 colors should be ordered by importance
	Red	1 byte
	Green	1 byte
	Blue	1 byte
	reserved	1 byte
	repeated NumColors times	unused (=0)
Raster Data	Info.ImageSize bytes	The pixel data

Fonte: http://www.ue.eti.pg.gda.pl/fpgalab/zadania.spartan3/zad_vga_struktura_pliku_bmp_en.html

Os arquivos *bitmap* a serem considerados neste projeto **não devem conter uma tabela de cores**, considerando que essa tabela é opcional. Dessa forma, os cabeçalhos do arquivo *.BMP* ocuparão um número fixo de bytes: 14 bytes do cabeçalho geral (*Header*), e 40 bytes do cabeçalho de informações (*InfoHeader*), totalizando 54 bytes de cabeçalho. Esses 54 bytes deverão apenas ser lidos no arquivo de origem e copiados no arquivo de destino, sem sofrerem nenhuma alteração.

Em seguida, teremos 3 bytes para cada pixel da imagem, considerando a formação da imagem da esquerda para a direita e de cima para baixo. Os bytes seguem a sequência de cores azul, verde e vermelha (ou seja, a ordem inversa de RGB). Uma recomendação que pode ser feita, então, é que após a leituras dos 54 bytes iniciais do cabeçalho, o arquivo de entrada seja lido de 3 em 3 bytes. Nesse array de 3 bytes, estabelece-se que o endereço do array + 0 equivale ao endereço da cor azul, o endereço do array + 1 equivale ao endereço da cor verde, e o endereço do array + 2 equivale ao endereço da cor vermelha. Essa soma do endereço base do array com um índice pode ser realizada em um registrador (ex.: EBX). Uma vez que o endereço da cor desejada esteja em um registrador, pode se fazer um acesso indireto à memória através desse registrador para realizar alterações nesse byte específico.

Por fim, recomenda-se a utilização de um editor de arquivo hexadecimal para facilitar o entendimento do que está acontecendo com o arquivo *bitmap*, considerando que é um arquivo binário. Você poderá utilizar um editor hexadecimal web, como o disponível no link <https://hexed.it>. O arquivo exemplo do projeto, “catita.bmp”, é exibido da seguinte forma nesse editor:



Perceba que podemos identificar com relativa facilidade até onde vão os cabeçalhos, e onde começa a lista de bytes de cores dos pixels. Na imagem abaixo, o cabeçalho está sinalizado em vermelho, e os bytes de cores dos 3 primeiros pixels (na sequência azul, verde e vermelho) estão sinalizados de verde.

