



UNIVERSIDADE FEDERAL DA PARAÍBA CENTRO DE INFORMÁTICA

DISCIPLINA: Arquitetura de Computadores

PROFESSOR: Ewerton Monteiro Salvador

TRABALHO COM LINGUAGEM ASSEMBLY

“Censurador de Arquivo BMP”

O programa especificado abaixo deverá ser implementado utilizando-se a linguagem Assembly, no Windows (MASM 32 bits) ou no Linux (NASM 32 bits ou 64 bits). O trabalho será **em dupla** e deverá ser enviado pelo SIGAA até as **23:59h do dia 31/10/2023**.

ESPECIFICAÇÃO

Escreva um programa que receba como entrada o nome do arquivo de uma imagem no formato *bitmap* (extensão .bmp) com cabeçalho de 54 bytes e número de pixels por linha (largura da imagem) que seja múltiplo de 4 (mais detalhes sobre isso no final da especificação, na seção que apresenta características de arquivos BMP). Além disso, o programa também deve receber como entrada mais 4 números de 4 bytes: uma coordenada inicial x e y , uma largura e uma altura. Por fim, o programa também deve solicitar o nome de um arquivo de saída. O programa deve produzir como saída uma cópia da imagem recebida contendo um retângulo de cor preta censurando uma determinada área, desenhado a partir da coordenada inicial informada (x,y), largura e altura especificadas.

Por exemplo, suponha que um usuário forneça a imagem “fotoanonima.bmp”, a qual precisa ser censurada de modo a impossibilitar a identificação da pessoa fotografada:



Suponha agora que um usuário utilizou o seu programa para censurar a possível fonte de mal-entendido. Ele executou o programa e informou o nome do arquivo (fotoanonima.bmp), informou o ponto X 250 e o ponto Y 310 como coordenadas iniciais (em arquivos BMP as coordenadas crescem da esquerda para a direita, e **de baixo para cima**), e informou a largura 230 e altura 30. O programa deve produzir como resultado a imagem abaixo, a qual insere a censura na área indicada e claramente impossibilita a identificação da pessoa fotografada!



Recomenda-se que os(as) alunos(as) utilizem a seguinte metodologia de desenvolvimento:

- Primeiramente implementar um programa que recebe as entradas solicitadas, abra o arquivo indicado e realize a seguinte sequência de operações:
 - Ler 18 bytes (14 + 4) do arquivo recém-aberto e escrever esses bytes no arquivo de saída;
 - Ler 4 bytes referentes ao tamanho da largura da imagem de entrada, salvar o valor em uma variável inteira e escrever esses bytes no arquivo de saída;
 - Ler os 32 bytes restantes do cabeçalho da imagem (o cabeçalho tem um total de 54 bytes) e escrever esses bytes no arquivo de saída;
 - Por fim, implemente um loop que leia a quantidade de bytes equivalente ao tamanho da largura da imagem multiplicada por 3 (considerando que para cada pixel de largura existem 3 bytes referentes aos componentes RGB), e simplesmente escreva os bytes lidos no arquivo de destino, sem nenhuma alteração. Para essa leitura, recomenda-se que os bytes de uma linha da imagem sejam salvos em um array de 6480 bytes (3 bytes/pixel multiplicados por 2160 pixels, que é a largura de uma imagem com resolução 4K – o maior tamanho de imagem possível no contexto deste projeto);
 - Ao final desse processo você terá um programa que salva em uma variável a largura de uma imagem e simplesmente faz uma cópia inalterada da imagem informada na entrada. **Essa primeira parte do trabalho vale 5 pontos**
- Uma vez que o tópico anterior tenha sido solucionado, altere o programa para desenhar o retângulo preto na área indicada. Recomenda-se o uso da seguinte estratégia:
 - Crie uma função que receba 3 parâmetros, na seguinte ordem: 1. O endereço do array que contém os bytes da linha da imagem; 2. A coordenada X inicial; 3. A largura da censura a ser aplicada. Essa função deve preencher os pixels a partir da posição X com três bytes 0, referentes à cor preta no padrão RGB. Esse preenchimento deve acontecer até a posição “X inicial” + largura. A inclusão dessa função conforme especificado é considerada **obrigatória** nesta parte do projeto;
 - Uma vez que essa função esteja pronta, altere o loop de leitura do arquivo, para que essa função altere apenas as linhas que estejam no conjunto que vai desde linha da coordenada Y inicial até a linha “Y inicial” + altura. As linhas que não estiverem contidas neste conjunto devem ser apenas copiadas de forma inalterada para o arquivo de destino.

Algumas 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;
- 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 em 32/64 bits para Linux (NASM). O trabalho deve ser desenvolvido **em dupla**. O código implementado deve ser original e gerado exclusivamente pelos alunos da dupla, não sendo permitidas cópias de códigos inteiros ou trechos de códigos de outras fontes (exceto quando expressamente autorizado pelo professor da disciplina). Também não é permitido o uso de código gerado por sistemas de inteligência artificial generativa, como o ChatGPT, o qual deverá ser utilizado apenas para tirar possíveis dúvidas dos alunos, e não para gerar o código que de fato irá para o trabalho a ser desenvolvido. Por fim, **recomenda-se enfaticamente que não haja compartilhamento de código entre os(as) alunos(as) da disciplina**. Os debates entre alunos(as) devem estar restritos a ideias e estratégias, e nunca envolver códigos, para evitarem penalidades na nota relacionadas à plágio.

--- Boa sorte! ---

INFOMAÇÕES COMPLEMENTARES PARA O PROJETO (Considerando Windows 32 bits e Linux 32 bits)

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 estará no mesmo diretório do arquivo executável do projeto);

Observação importante: strings recebidas através de funções de entrada de dados (como `ReadConsole` do `MASM32`) costumam ser terminadas com os caracteres ASCII “Carriage Return” (CR, decimal 13), seguido de “Line Feed” (LF, decimal 10), seguido finalmente do terminador de string (decimal 0). Contudo, um nome de arquivo **não deve conter** os caracteres CR ou LF, portanto, a string recebida por esse tipo de função precisa ser tratada para remover esses caracteres problemáticos. O trecho de código abaixo (trecho de código autorizado para ser utilizado no projeto) percorre uma string procurando o caractere CR (ASCII 13), e quando encontra esse caractere, o substitui pelo valor 0 (terminador de string). Dessa forma, a string resultante desse tratamento pode ser utilizada na função para abertura de arquivo.

```
mov esi, offset uma_string ; Armazenar apontador da string em esi
proximo:
mov al, [esi] ; Mover caractere atual para al
inc esi ; Apontar para o proximo caractere
cmp al, 13 ; Verificar se eh o caractere ASCII CR - FINALIZAR
jne proximo
dec esi ; Apontar para caractere anterior
xor al, al ; ASCII 0
mov [esi], al ; Inserir ASCII 0 no lugar do ASCII CR
```

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 de entrada, e `CREATE_ALWAYS` para a criação do arquivo 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 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 (essa convenção utiliza números na base **octal**). 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. Um retorno “-1” indica a ocorrência de erro.

Ex.:

```
mov eax, 8 ; sys_creat
mov ebx, filename
mov ecx, 0o777
int 80h
```

```
mov fileHandle, eax
```

Abertura de Arquivo Já Existente: Linux

Realizada através da interrupção 80h

Parâmetros:

1. O registrador EAX deve receber o valor 5, referente à chamada de sistema `sys_open`;
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 estará no mesmo diretório do arquivo executável do projeto);
3. 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);
4. O registrador EDX deve conter as permissões do arquivo, de acordo com a convenção de permissões de arquivos utilizada pelo Linux (essa convenção utiliza números na base **octal**). 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. Um retorno “-1” indica a ocorrência de erro.

Ex.:

```
mov eax, 5           ; sys_open
mov ebx, filename
mov ecx, 0           ; read_only
mov edx, 0o777
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. Retorno “-1” indica a ocorrência de erro. **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. Um retorno “-1” indica a ocorrência de erro.

```
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:

1. O registrador EAX deve receber o valor 4, referente à chamada de sistema `sys_write`;
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 com o conteúdo a ser gravado no arquivo;
4. 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. Um retorno “-1” indica a ocorrência de erro.

```
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. Um retorno “-1” indica a ocorrência de erro.

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

Verificando códigos de erro: Windows

Realizado através da função **GetLastError**

Retorno: um código de erro de 4 bytes, de acordo com as listagens disponíveis no seguinte link - <https://learn.microsoft.com/en-us/windows/win32/debug/system-error-codes>

```
Ex.:
invoke GetLastError
```

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 a seguir.

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	Red intensity
		Green 1 byte	Green intensity
		Blue 1 byte	Blue intensity
		reserved 1 byte	unused (=0)
repeated NumColors times			
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. Uma última observação é que os pixels da imagem são codificados de tal forma que a quantidade de bytes em uma linha da imagem precisa ser múltipla de 4 bytes. Caso o número de pixels em uma linha multiplicado por 3 (1 byte para cada cor do pixel) não seja múltiplo de 4, são acrescentados bytes 0 (zero) ao final da linha até que o número total de bytes se torne múltiplo de 4. Contudo, para este projeto, iremos aceitar como entrada **apenas imagens que tenha um número de pixels em uma linha que seja múltiplo de 4**. Por exemplo, a imagem “catita.bmp” fornecida como exemplo, possui 640 pixels por linha, que é múltiplo de 4.

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>. Um arquivo BMP é exibido da seguinte

