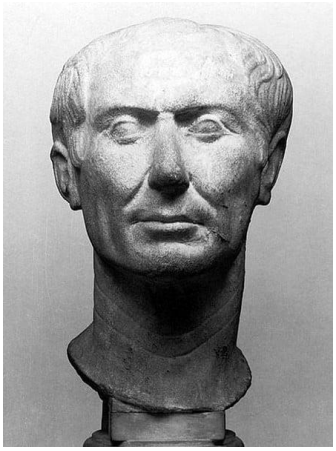
	UNIVERSIDADE FEDERAL DA PARAÍBA **CENTRO DE INFORMÁTICA** **DISCIPLINA: Arquitetura de Computadores** **PROFESSOR: Ewerton Monteiro Salvador**

TRABALHO COM LINGUAGEM ASSEMBLY
"Cifra de César"

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 individual e deverá ser enviado pelo SIGAA até as **23:59h do dia 26/05/2023**.

ESPECIFICAÇÃO



Busto de Júlio César

A Cifra de César é um dos algoritmos de criptografia mais simples de serem implementados, e também um dos mais simples de serem "quebrados". O nome do algoritmo é uma homenagem ao imperador romano Júlio César, que utilizava essa técnica para proteger mensagens que continham segredos militares. A técnica original consistia em se trocar cada letra da mensagem original por uma outra letra que fosse listada no alfabeto em um número fixo de posições após a letra substituída. Por exemplo, se considerarmos o uso da Cifra de César usando um deslocamento de 3 posições (neste exemplo, a "chave" seria a constante 3), teríamos os seguintes caracteres originais e cifrados:

ORIGINAL: ABCDE
CIFRADO: DEFGH

Neste trabalho, iremos considerar uma variação da Cifra de César onde **todos os bytes de um arquivo texto** são deslocados para frente (deslocamento positivo) em uma constante que pode ser de 1 a 20 (essa constante será chamada de "chave"). Ou seja, o processo de criptografia consiste em somar o valor da chave a cada um dos bytes de um arquivo, e o processo de decifração consiste em subtrair o valor da chave de cada um dos bytes de um arquivo.

Escreva um programa que apresente um menu de opções para o usuário contendo ao menos 3 itens: "Criptografar", "Descriptografar" e "Sair". As opções "Criptografar" e "Descriptografar" devem

solicitar 3 entradas: o nome de um arquivo de entrada, o nome de um arquivo de saída e uma chave (de 1 a 20). “Criptografar” deve produzir como arquivo de saída uma versão do arquivo de entrada em que todos os bytes foram somados com o valor fornecido para a chave. Já a opção “Descriptografar” deve produzir como arquivo de saída uma versão do arquivo de entrada em que o valor da chave foi subtraído de todos os bytes. O menu de opções deve ser reapresentado para o usuário após a execução de “Criptografar” ou “Descriptografar”, até que o usuário selecione a opção “Sair” para encerrar a execução do programa.

Observações importantes:

- O(a) aluno(a) não deve se preocupar em validar as entradas de dados. Assuma que o usuário sempre digitará valores válidos como entradas de dados;
- 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 processo de leitura e escrita de arquivos deve utilizar um buffer de 512 bytes (tamanho típico de setores em HD/SSD). Ou seja, tanto a leitura quando a escrita de arquivos deve acontecer em porções de 512 bytes;
- O programa deve implementar uma função para criptografar e outra para descriptografar o buffer de leitura do arquivo (512 bytes). Essas funções devem receber três parâmetros: 1) o endereço do buffer a ser criptografado/descriptografado (ou seja, passar o buffer de leitura do arquivo **por referência**), 2) o tamanho do buffer (para os casos em que o buffer não é totalmente preenchido pela leitura de um arquivo), e 3) um número inteiro de 1 a 20 a ser utilizado como chave para criptografia/descriptografia. Tanto o tamanho do buffer (parâmetro 2) como a chave (parâmetro 3) podem ser números inteiros de 4 bytes. A função não deve retornar nenhum valor. A não utilização de função acarretará em redução da nota do trabalho;
- Não é permitida a utilização de pseudoinstruções Assembly além daquelas utilizadas na disciplina até o momento (por exemplo, “invoke” do MASM32). Se você possui dúvida se pode ou não utilizar um determinado recurso do MASM ou do NASM, por favor, consulte o professor da disciplina;
- A entrada e saída de console no Windows deve utilizar as funções ReadConsole e WriteConsole da biblioteca kernel32. A utilização da macro “printf” no MASM32 acarretará em redução da nota do trabalho. No Linux podem ser utilizadas as funções printf e scanf da biblioteca padrão da linguagem C, utilizando o gcc para “linkagem” do programa;
- Cada aluno deverá tomar medidas para garantir que o código-fonte possua boa legibilidade (comentários são cruciais nesse sentido) e que o programa seja minimamente eficiente, ou seja, o programa não deverá realizar ações claramente desnecessárias para a solução do problema.

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 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! ---

DESAFIO (2 pontos extra no trabalho de Assembly)

Adicione uma opção ao menu do programa chamada “Criptoanálise”. Essa opção deve receber como entrada apenas o nome de um arquivo, e deve produzir como saída um valor de chave que o programa julgue ser a correta para descriptografar o arquivo. A criptoanálise deverá considerar que a mensagem original estará na língua portuguesa. Uma possível técnica de criptoanálise é a utilização de todas as chaves possíveis (21 no caso deste projeto, considerando as chaves de 1 a 20 e também a chave 0, representando a possibilidade do arquivo informado já estar descriptografado) seguida de uma forma de avaliação da distribuição das letras obtida a partir de cada chave utilizada, em busca de uma distribuição esperada na língua portuguesa (existem várias fontes bibliográficas possíveis para distribuição de letras em textos de língua portuguesa, sendo uma dessas fontes a disponível no seguinte link - https://www.gta.ufrj.br/grad/06_2/alexandre/criptoanalise.html). Como exatamente será feita essa avaliação da distribuição das letras é algo que ficará a critério de cada aluno(a). Considere que o arquivo criptografado fornecido para essa opção possua **pelo menos 100 caracteres** (sem contar espaços em branco). A opção deverá encontrar a chave correta em um conjunto de pelo menos 2 testes que o professor da disciplina realizará.

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 escrito 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_ATRIBUTES (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
```

Reposicionamento do Apontador de Arquivo: Windows

Realizado através da função **SetFilePointer**

Parâmetros:

1. Handle de 4 bytes do arquivo cujo apontador será reposicionado. Esse handle é recebido como retorno da função de abertura do arquivo;
2. Um número dos 4 bytes menos significativos do deslocamento sobre o apontador do arquivo a ser realizado a partir de um ponto de partida;
3. Um apontador para os 4 bytes mais significativos do deslocamento sobre o apontador do arquivo a ser realizado a partir de um ponto de partida. Se os 4 bytes mais significativos não forem necessários, deverá ser utilizada a constante NULL;
4. Uma constante definindo o ponto de partida para a movimentação do apontador de arquivo. As constantes possíveis são FILE_BEGIN (início do arquivo), FILE_CURRENT (posição atual do apontador de arquivo) e FILE_END (final do arquivo).

Retorno: os 4 bytes menos significativos da quantidade de posições em que o apontador do arquivo foi movimentado.

Ex.:

```
invoke SetFilePointer, fileHandle, 0, NULL, FILE_BEGIN
```

Reposicionamento do Apontador de Arquivo: Linux

Realizada através da interrupção 80h

Parâmetros:

1. O registrador EAX deve receber o valor 19, referente à chamada de sistema sys_lseek;
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 o deslocamento sobre o apontador do arquivo a ser realizado a partir de um ponto de partida;
4. O registrador EDX deve conter o ponto de partida para a movimentação do apontador de arquivo. As opções possíveis são 0 (SEEK_SET, ou seja, início do arquivo), 1 (SEEK_CUR, ou seja, posição atual do apontador de arquivo) e 2 (SEEK_END, ou seja, final do arquivo).

Retorno: No registrador EAX terá a quantidade posições efetivamente deslocadas no apontador de arquivo. Um retorno “-1” indica a ocorrência de erro.

Ex.:

```
   mov eax, 19           ; sys_lseek
   mov ebx, [fileHandle]
   xor ecx, ecx
   xor edx, edx
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
```