SEMINÁRIO DE COMPILADORES

Loaders & Arquivos Executáveis: .exe, elf, etc

ARQUIVO EXECUTÁVEL

O que é?

- É um arquivo que contém instruções codificadas (código binário de máquina) para serem executadas diretamente pelo processador
- Contém chamadas para rotinas no SO, ou seja, chamadas sys. Chamadas para a API do **Win32** no Windows (ou MSIL - Microsoft Intermediate Language) e para o **posix** no Linux/Unix.
- O arquivo exe pode ser compilado para que tenha como alvo qualquer CPU que o CLR suporte
- Contém dados de programa, recursos, informações de realocação, informações de importação e etc.

- No **Windows**, as extensões mais comuns são: exe, bat, com, cmd, inf, ipa, osx, pif, run e wsh
- No **Mac**, o formato app
- No **Linux**, arquivo out, deb, rpm

Bibliotecas dinâmicas x Arquivo objeto

(arquivo .so e .dll e mostrar exemplo ou desenho, e como são usados pelo loader durante a execução)

Apesar de as **DLLs (Dynamic-Link Libraries)** utilizarem a mesma estrutura de um arquivo executável, elas não podem ser rodadas diretamente. Ela possui dados e funções (exportadas ou internas) que podem ser usados por outros módulos, e metadados. Elas podem exportar dados, mas são geralmente utilizados apenas pelas funções internas. As DLLs ajudam a reduzir a sobrecarga de memória quando vários aplicativos usam a mesma funcionalidade em paralelo. Embora cada aplicativo receba sua própria cópia dos dados da DLL, os aplicativos compartilham o código da DLL.

Algumas vantagens incluem a redução de códigos que dependem da dll e melhorar o desempenho dos programas, atualizações independentes. O que implica na desvantagem de problemas de compatibilidade de executáveis com novas versões da DLL. Além disso, como as DLLs são carregadas na memória em tempo de execução, elas ficam abertas a malwares e outras ameaças de segurança.

Os **arquivos objetos** possuem código de máquina que ainda não está completamente ligado, ou seja, não está pronto para executar independentemente. Possui código de máquina, tabela de símbolos, informações de realocação, e metadados. Podem

ser descritos como blocos de construção intermediários usados durante o processo de compilação para criar executáveis ou bibliotecas.

Por que existem vários tipos de executáveis?

Os programas executáveis possuem códigos opcode em bytes para que o processador entenda. Mas, além disso, é necessário a ajuda de um sistema operacional. O SO é responsável por configurar um contexto para o programa (fornecer memória, carregar os dados, bibliotecas, configurar uma pilha), mas para isso, ele precisa saber algumas informações sobre o programa (dados que utilizará, tamanho dos dados, valores iniciais armazenados na pilha, lista de opcodes que compõem o programa). Esses dados serão armazenados no arquivo executável, e cada SO entende um formato diferente de executável.

Formatos de arquivos executáveis

(arquivos exe, out, elf… explicar a diferença entre os principais, colocar um desenho para mostrar)

Existem muitos formatos de arquivos executáveis, como o PE, ELF, Mach-O, A.OUT, WASM, PEF, MZ/DOS EXE, COFF, FAT Binary, JAR, APK, BIN. Mas aprofundaremos em apenas alguns.

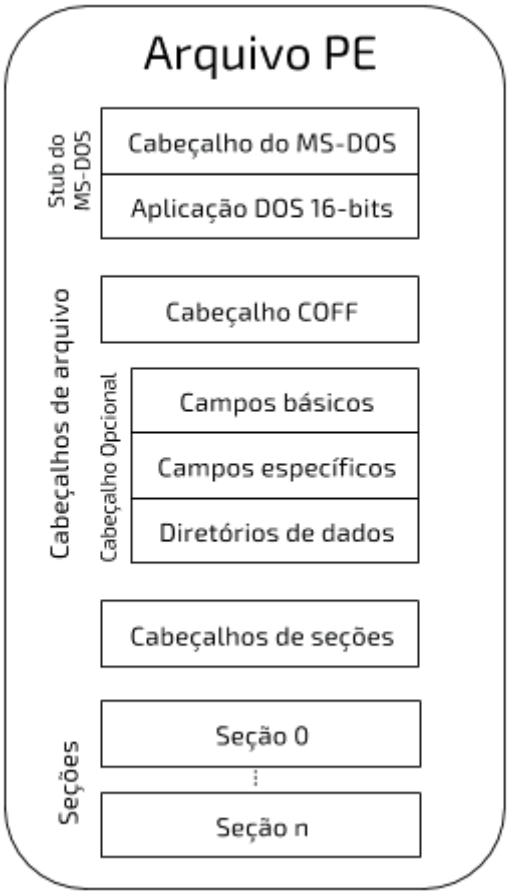
O Windows utiliza um formato de executável **PE (Portable Executable) e COFF.** No PE, há uma seção (.text) que tem instruções de máquina. Antigamente, o MSDOS utilizava um formato chamado 16-bit DOS MZ, que mais tarde, foi se tornando o padrão utilizado hoje.

O UNIX utiliza os formatos **ELF(Executable and Linkable Format)** e **COFF (Common Object File Format)**.

Mas nem todos os executáveis precisam de formatos, como por exemplo, os **bootloaders** de sistemas, que ocupam o primeiro setor de uma partição bootavel do disco e são usado para carregar o SO.

Formato PE

O nome desse formato refere-se ao fato de que o formato não é específico da arquitetura da máquina. Inclui os arquivos exe, dll, ocx, cpl e sys.



O arquivo de **cabeçalho do MS-DOS** é composto pela estrutura:

```
typedef struct {
    uint16_t e_magic;
    uint16_t e_cblp;
    uint16_t e_cp;
    uint16_t e_crlc;
    uint16_t e_cparhdr;
    uint16_t e_minalloc;
    uint16_t e_maxalloc;
    uint16_t e_ss;
    uint16_t e_sp;
    uint16_t e_csum;
    uint16_t e_ip;
    uint16_t e_cs;
    uint16_t e_lfarlc;
    uint16_t e_ovno;
    uint16_t e_res[4];
    uint16_t e_oemid;
    uint16_t e_oeminfo;
    uint16_t e_res2[10];
    uint32_t e_lfanew;
} IMAGE_DOS_HEADER;
```

- O campo **e_magic** sempre terá os caracteres ‘MZ’.
- O campo **e_lfanew** é conhecido como assinatura PE e tem uma sequência de bytes que representam “PE\0\0”.

Na **aplicação DOS 16-bits** está codificado uma função que imprime na tela o erro “This program cannot be run in DOS mode.” caso o usuário tente rodar no MS-DOS.

O **cabeçalho COFF** é especificado para o sistema VAX/VMS, reutilizado no Windows. A razão pelo o qual ainda está lá, é porque boa parte dos engenheiros do time que desenvolveu o Windows NT trabalhavam na empresa responsável por esse sistema.

```
typedef struct {
    uint16_t Machine;
    uint16_t NumberOfSections;
    uint32_t TimeDateStamp;
    uint32_t PointerToSymbolTable;
    uint32_t NumberOfSymbols;
    uint16_t SizeOfOptionalHeader;
    uint16_t Characteristics;
} IMAGE_FILE_HEADER, IMAGE_COFF_HEADER;
```

- **Machine**: indica a arquitetura de máquina para o qual o programa foi construído
- **NumberOfSections:** número de seções que o arquivo PE possui
- **TimeDateStamp:** data da criação, mas pode ser alterado. Ele não é utilizado pelo loader.
- **SizeOfOptionalHeader:** tamanho do Cabeçalho Opcional
- **Characteristics:** define alguns atributos do arquivo usando uma máscara de bits. Ele guarda características como se o arquivo é dll, se é 32 bits, se é executável, entre outros.

Apesar do seu nome, o **cabeçalho Opcional** é obrigatório para arquivos executáveis, apenas opcional para arquivos objeto. O tamanho desse cabeçalho não é fixo, e definido pelo **SizeOfOptionalHeader** do COFF. A estrutura para arquivos PE de 32 bits é a seguinte:

```
typedef struct {
    uint16_t Magic;
    uint8_t MajorLinkerVersion;
    uint8_t MinorLinkerVersion;
    uint32_t SizeOfCode;
    uint32_t SizeOfInitializedData;
    uint32_t SizeOfUninitializedData;
    uint32_t AddressOfEntryPoint;
    uint32_t BaseOfCode;
    uint32_t BaseOfData;
    uint32_t ImageBase;
    uint32_t SectionAlignment;
    uint32_t FileAlignment;
    uint16_t MajorOperatingSystemVersion;
    uint16_t MinorOperatingSystemVersion;
    uint16_t MajorImageVersion;
    uint16_t MinorImageVersion;
    uint16_t MajorSubsystemVersion;
    uint16_t MinorSubsystemVersion;
    uint32_t Reserved1;
    uint32_t SizeOfImage;
    uint32_t SizeOfHeaders;
    uint32_t CheckSum;
    uint16_t Subsystem;
    uint16_t DllCharacteristics;
    uint32_t SizeOfStackReserve;
    uint32_t SizeOfStackCommit;
    uint32_t SizeOfHeapReserve;
    uint32_t SizeOfHeapCommit;
    uint32_t LoaderFlags;
    uint32_t NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[MAX_DIRECTORIES];
} IMAGE_OPTIONAL_HEADER_32;
```

- **Magic:** identifica o tipo de executável (32/64bits), representado por um número mágico 🙂
- **AddressOfEntryPoint**: contém o endereço do ponto de entrada do programa. Nos executáveis, ele é um endereço relativo à base da imagem (como a Microsoft chama o arquivo executável). Para bibliotecas, não é necessário.
- **ImageBase:** endereço de memória que representa onde estará a base do arquivo executável carregado em memória. Por padrão, executáveis é 0x400000 e dlls são 0x10000000.
- **SectionAlignment:** define qual o fator de alinhamento deve ser utilizado para todas as seções do binário quando mapeadas em memória. O padrão é o valor do tamanho da página de memória do sistema.
- **SubSystem:** define o tipo de subsistema necessário para rodar o programa. Por exemplo: 0x002 - Windows GUI (Graphical User Interface) e 0x003 - Windows CUI (Character User Interface)
- **DllCharacteristics:** usa uma máscara de bits para descrever características do arquivo. Por exemplo: se haverá randomização de endereços de memória.

Os **diretórios de dados** são 16 no total, cada um especificando uma função. A estrutura de cada diretório é a seguinte:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    uint32_t   VirtualAddress;
    uint32_t   Size;
} IMAGE_DATA_DIRECTORY;
```

Os diretórios mais importantes são:

- **Export Table:** tabela de funções exportadas pela aplicação. VirtualAddress aponta para uma estrutura chamada EDT(export Directory Table), que contém os nomes e endereços das funções exportadas e seus endereços. Também aponta para uma estrutura EAT (Export Address Table).
- **Import Table:** tabela de funções importadas pela aplicação. O VirtualAddress aponta para a IDT(Import Directory Table), que é um array de estruturas que apontam para a IAT(Import Address Table).
- **Resource Table:** aponta para uma estrutura de árvore binária que guarda todos os resources(ícones, janelas, strings…)

A **Import Directory Table(IDT)** é um array de estruturas do tipo abaixo. O número de estruturas é o número de bibliotecas/dll que o executável depende, e finaliza com um elemento vazio pra indicar o fim do array.

```
typedef struct {
    union {
        uint32_t Characteristics;
        uint32_t OriginalFirstThunk; // Endereço da ILT
    } u1;
    uint32_t TimeDateStamp;
    uint32_t ForwarderChain;
    uint32_t Name; // Endereço do nome da DLL
    uint32_t FirstThunk; // Endereço da IAT
} IMAGE_IMPORT_DESCRIPTOR;
```

- **OriginalFirstThunk:** aponta para a **Import Lookup Table(ILT)** que possui o endereço para uma estrutura que contém o nome da função. O número de elementos do array ILT é o número de funções importadas por uma DLL
- **FirstThunk:** aponta para a **Import Address Table (IAT)**, muito semelhante a ILT, porém, ela possui os endereços reais das funções importadas (executáveis dinamicamente linkados)

Os **cabeçalhos de seções** definem cada uma das NumOfSections seções do COFF. Cada cabeçalho é composto pela estrutura:

```
typedef struct {
    uint8_t Name[SECTION_NAME_SIZE];
    uint32_t VirtualSize;
    uint32_t VirtualAddress;
    uint32_t SizeOfRawData;
    uint32_t PointerToRawData;
    uint32_t PointerToRelocations;
    uint32_t PointerToLinenumbers; // descontinuado
    uint16_t NumberOfRelocations;
    uint16_t NumberOfLinenumbers;  // descontinuado
    uint32_t Characteristics;
} IMAGE_SECTION_HEADER;
```

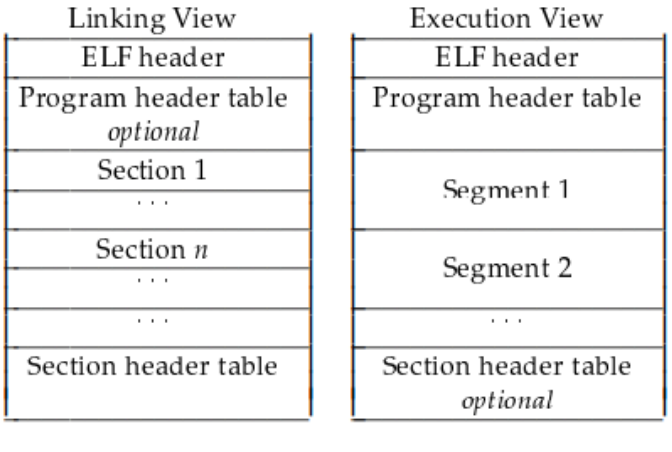
- **Name:** nome da seção. Por exemplo: .text
- **VirtualSize:** tamanho em bytes da seção depois de mapeada na memória pelo loader.
- **VirtualAddress:** endereço relativo à base da imagem quando carregada na memória
- **SizeOfRawData:** tamanho em bytes da seção do arquivo PE, antes de ser mapeada na memória
- **PointerToRawData:** aponta para o primeiro byte da seção
- **Characteristics:** define algumas flags para a seção (usando máscara de bits), além das permissões em memória que ela deve ter quando for mapeada. Por exemplo: Se a seção possui código executável, dados inicializados, dados não inicializados, entre outros.

As seções são geralmente separadas em relação ao seu conteúdo, entre código e dados. Algumas padronizadas são:

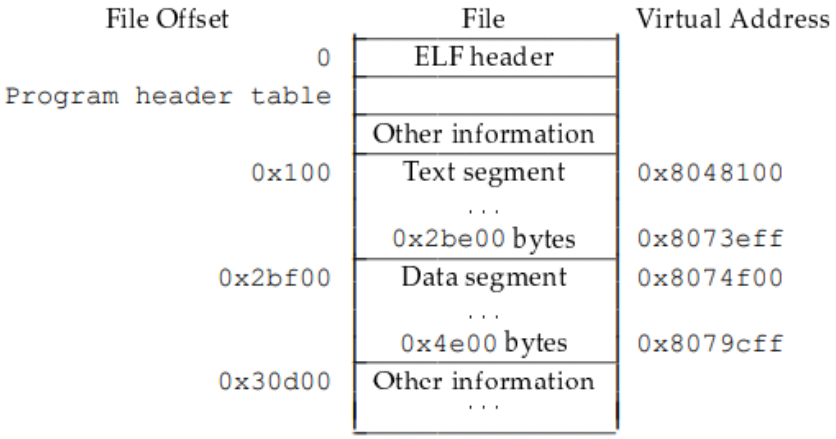
- **.text/.code**: além do código, geralmente possui permissões de leitura e execução
- **.data**: dados inicializados/declarados com permissão de leitura e escrita
- **.rdata:** dados inicializados com permissão somente de leitura (constantes)
- .**idata:** tabela de imports, possui permissão de leitura e escrita

Formato ELF

Um formato que abrange arquivos executáveis, bibliotecas compartilhadas (.so) e arquivos objetos (.o). Diferente do formato PE, esse formato utiliza seções e segmentos.
Os **segmentos** são usados pelo loader no processo de carregamento do programa na memória. Cada segmento é mapeado para várias seções, mas nem todas as seções vão ser mapeadas para um segmento. Nem todas as seções vão ser carregadas na memória e nem todos os arquivos objetos têm segmentos.
Cada seção possui dados ou códigos de um tipo/uso específico, e podem variar de tamanho. As seções são contidas em arquivos em memória.



Executable File



O **cabeçalho de identificação ELF** (16Bytes) contém informações básicas do programa:

- **Magic:** Número mágico
- **Class:** 32/64bits
- little/big endian
- versions

O **header ELF** possui informações sobre o layout do executável:

- Versão do ELF
- executável/biblioteca/outro
- arquitetura alvo
- endereço de entrada
- outros

O **header do programa** possui informações para o loader de quais segmentos do arquivo colocar na memória e onde coloca-los:

- tipo
- flags de permissão
- endereço de memória do alvo

- outros

O resto do arquivo possui instruções de máquina e dados que queremos utilizar, guardadas em sessões, semelhante ao formato PE. As sessões mais comuns do ELF:

- **.data:** dados inicializados com permissão de leitura e escrita
- **.rodata:** dados inicializados com permissão apenas de leitura
- **.bss:** dados não inicializados com permissão de leitura e escrita
- **.got:** uma tabela contendo os endereços de memória das funções
- **.plt:** aponta para nomes de funções para as entradas na .got

Programas para verificar o conteúdo de um arquivo executável

(curiosidade)

- Microhex
- wxHexEditor
- dumpbin (verificar os atributos de arquivos PE)
- DIE
- x64dbg

Editores Hexadecimais

- hexDump
- bvi
- fhex
- GNUpoke
- hexflend
- hexpatch
- rehex

- xxd (Linux)
- readelf (lê arquivos elf no Linux)
- ghex (Linux)
- bless (Linux)

Disassemblers

- IDA Pro Freeware

LOADER

O que é

(explicar o que é e para que serve)

- Geralmente o Loader é considerado um componente do sistema operacional que carrega códigos binários na memória e sendo distinto, mas complementar, ao Linker.
- Componente essencial de um sistema operacional, encarregado de carregar programas e bibliotecas na memória para prepará-los para execução pela CPU
- Ou seja, ele atua como uma ponte entre o armazenamento e a CPU

Como funciona

(explicar como funciona e se possível, mostrar um esquema/desenho ou passo a passo de como funciona)

O Loader aceita o código objeto gerado pelo compilador e o deixa pronto para a execução. Esse procedimento possui quatro partes:

Pré-fase: Um objeto de processo é criado. O Loader lê o programa do armazenamento

1. **Alocação (allocation):** aloca memória para o programa na memória principal.

Para alocar memória no programa, o carregador aloca a memória com base no tamanho do programa. O loader fornece o espaço na memória onde o programa objeto será carregado para a execução.

2. **Vinculação (linking):** combina dois ou mais programas ou módulos de objetos separados e fornece as informações necessárias.

Resolve o código de referências simbólico ou dados entre os módulos de objeto alocando todos os endereços de sub-rotina do usuário e da biblioteca. O linker vincula uma função a linha do módulo onde a implementação está.

3. **Realocação (reallocation):** Modifica o programa objeto para que ele possa ser carregado em um endereço diferente do local.

Existem alguns locais dependentes de endereço no programa, e essas constantes de endereço devem ser modificadas para se ajustarem ao espaço disponível. O loader modifica o programa objeto (instruções específicas) para que o programa objeto seja carregado em um endereço diferente do fornecido inicialmente.

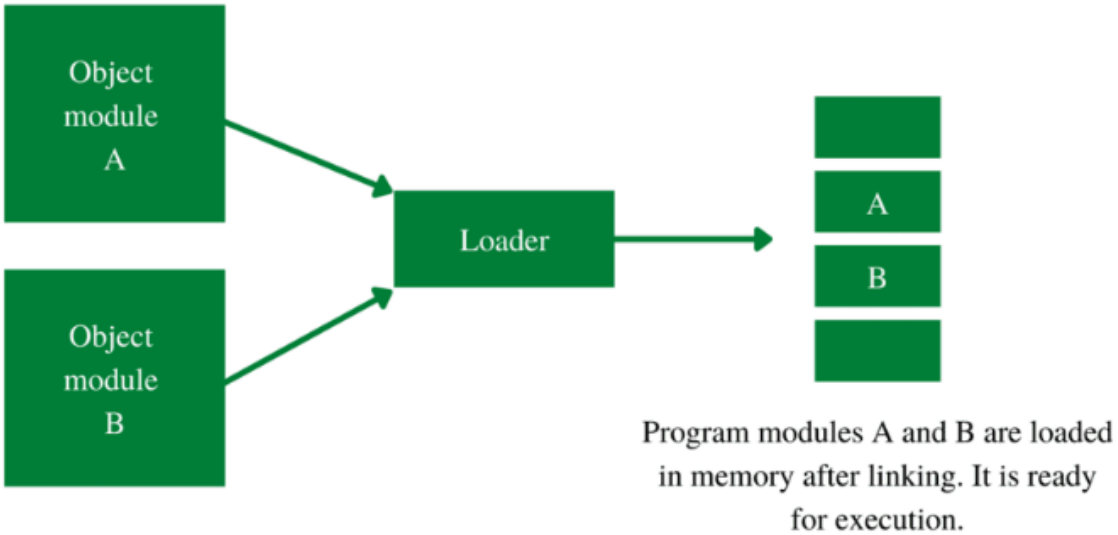
4. **Carregamento (loading):** traz o programa objeto para a memória principal para a execução.

O loader carrega as instruções de máquina e dados de programas e subrotinas relacionados na memória principal. Ele executa o carregamento depois que o assembler fornece o arquivo objeto.

OBS: Depois que os módulos estão na memória, ele ajusta o código e os dados do programa para se adequarem aos endereços de memória reais. Prepara a pilha e define os parâmetros de execução inicial.

Exemplo de execução do .exe

1. Um objeto de processo é criado
2. O arquivo exe é lido na memória do processo. As seções do exe são mapeadas separadamente e recebem diferentes permissões (executar, ler/escrever, apenas escrever)
3. As realocações ocorrem no exe (caso ele não tenha sido carregado no endereço preferido)
4. A tabela de importação é percorrida e as DLLs necessárias são carregadas
5. As DLLs são mapeadas, e caso necessário, ocorrem realocações também. Depois suas DLLs dependentes também são carregadas
6. O processo inicia a execução em um stub inicial em NTDLL
7. O stub do carregador inicial executa os pontos de entrada de cada DLL e depois salta para o ponto de início do exe



General loading scheme

Loading Estático x Loading Dinâmico

O **estático** carrega o programa inteiro na memória antes da execução, garantindo que todos os componentes estejam disponíveis na inicialização. Ele é mais estável, mas consome mais memória.

O **dinâmico** carrega os módulos do programa sob demanda durante a execução, o que otimiza o uso de memória. Ele permite que os programas iniciem mais rápido e usem os recursos de forma eficiente, pois os módulos são carregados apenas quando necessário. Assim, ele melhora o desempenho, principalmente para aplicativos grandes, minimizando o consumo de memória e o tempo de inicialização.

Loaders x Linkers

Possuem propósitos diferentes, mas complementares.
O **linker** combina os módulos do programa, resolvendo referências simbólicas para formar um único arquivo executável. Ele cria uma unidade coesa que pode ser carregada na memória. Ou seja, responsável por preparar o programa.
O **loader** começa o trabalho quando o executável está pronto. Ele carrega o programa na memória, realiza realocações e inicia a execução. Ou seja, garante a execução do programa configurando o ambiente de memória.

Tipos de loaders

(pincelado)

Os loaders podem ser classificados em diferentes tipos, baseados no comportamento:

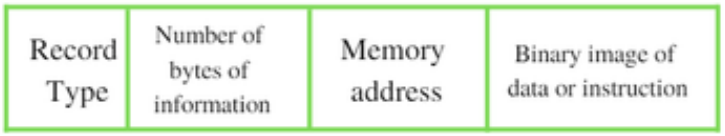
Loader Absoluto

Colocam programas em um local de memória fixo, ideal para sistemas simples.
Transfere o texto do programa para a memória no endereço fornecido pelo assembler após ler o programa objeto linha por linha. O **programa objeto** deve informar ao loader:

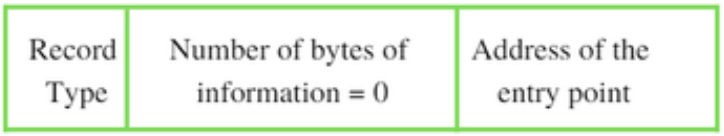
- as instruções de máquina que o assembler criou junto com o endereço de memória
- O início da execução do programa, que iniciará assim que o programa carregar

O programa objeto é a sequência de registros de objeto, e em cada registro está algum aspecto específico do programa no módulo de objeto. Há dois **tipos de registros**:

- **Texto**: contem uma imagem binária do programa assembly
- **transferência**: ponto de início ou entrada da execução



Text record



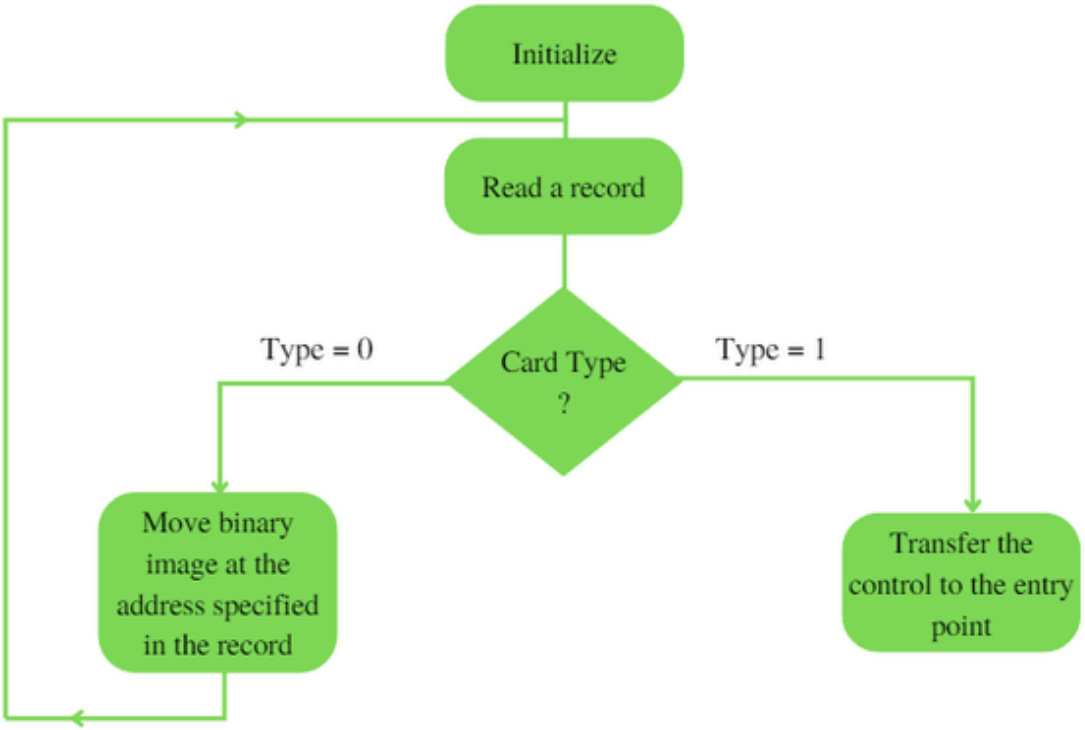
Transfer record

Record type = 0 for Text record
Record type = 1 for Transfer record

Formats of object records

O **algoritmo** do loader absoluto é o seguinte:

a) O arquivo objeto é lido registro por registro
b) A imagem binária é movida para os locais especificados no registro
c) O registro final é um registro de transferência
d) Quando o controle atinge o registro de transferência, ele é transferido para o ponto de entrada para execução



Loader Relocável

Mais flexiveis que os absolutos, eles ajustam os endereços no programa para se ajustarem a diferentes locais de memória, suportando multitarefa.

Loader Dinâmico

Carregam módulos de programas sob demanda, permitindo uso eficiente de memória e inicialização mais rápida do programa.

Gerenciamento de memória

(Importante! Colocar exemplo prático ou esquema/desenho do funcionamento)

Loader e multitarefas

- Podem manipular vários programas simultaneamente, ao alocar e gerenciar a memória, garantindo que cada programa opere dentro de seu espaço alocado
- O ambiente multitarefa é desafiador para os loaders, pois precisam alocar a memória de forma eficiente para não causar sobreposições ou conflitos
- Também precisam gerenciar as dependências, e garantir que todos estejam na memória quando for necessário
- Precisam fazer um tratamento de contenção de recursos, ou seja, quando vários programas competem por recursos limitados do sistema. Nesse tratamento, otimizam a distribuição de recursos fazendo um gerenciamento dos processos de alocação, vinculação e realocação de memória.

Benefícios

- Permite multitarefas ao gerenciar o espaço de memória para vários programas e bibliotecas
- A preparação do loader minimiza erros de execução, otimiza o uso de recursos e melhora o desempenho do sistema
- Sem ele, os programas enfrentariam atrasos de execução, erros ou falhas devido a ambientes de memória despreparados, prejudicando a confiabilidade do sistema

REFERÊNCIAS

Executáveis

[1] https://www-techtarget-com.translate.goog/whatis/definition/executable-file-exe-file?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=sge#:~:text=Hera%20Wigmore-,O%20que%20%C3%A9%20um%20arquivo%20execut%C3%A1vel%20(arquivo%20EXE)?,%E2%80%8B%E2%80%8Bs%C3%A3o%20arquivos%20EXE.

[2] https://stackoverflow.com/questions/1616772/how-exactly-do-executables-work

[3] https://stackoverflow.com/questions/2048052/what-does-executable-file-actually-contain?rq=3

[4] https://stackoverflow.com/questions/1495638/whats-in-a-exe-file?rq=3

[5] https://learn.microsoft.com/pt-br/windows/win32/debug/pe-format

[6] https://www.youtube.com/watch?v=-ojciptvVtY

[7] https://mentebinaria.gitbook.io/engenharia-reversa/o-formato-pe

[8] https://www.youtube.com/watch?v=cX5tQJhuNeY

[9] https://alacerda.github.io/posts/o-formato-elf/

[10] https://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf

[11] https://www.youtube.com/watch?v=1VnnbpHDBBA&t=1s

[12] https://man7.org/linux/man-pages/man5/elf.5.html

[13] https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries

[14] https://www.spiceworks.com/tech/tech-general/articles/what-is-dll/#:~:text=Advantages%20of%20DLL-,What%20Is%20a%20Dynamic%20Link%20Library%20(DLL)?,applications%2C%20to%20access%20UI%20components.

[15] https://pt.wikipedia.org/wiki/Arquivo_objeto

[16] https://www.linuxjournal.com/article/1059

[17] https://maskray.me/blog/2024-01-14-exploring-object-file-formats

[18]

Loaders

[1] https://www.geeksforgeeks.org/basic-functions-of-loader/

[2] https://www.lenovo.com/us/en/glossary/loader/?orgRef=https%253A%252F%252Fwww.google.com%252F&srsltid=AfmBOoqT8r4xGN_OIkofisQatXsLn72G1F697A7PvKeHokfZrpfQZ8iT