Roteiro de Estudo Definitivo: Carregando o Ambiente via `main`

Visão Geral

O Problema a Ser Resolvido:
Nosso Minishell precisa de uma cópia interna de todas as variáveis de ambiente (`PATH`, `USER`, etc.) para funcionar. Precisamos carregar essas variáveis na nossa Hash Table customizada assim que o programa inicia.

As Restrições do Projeto:

1. **Sem variáveis globais:** Isso proíbe o uso de `extern char **environ;`.
2. **Funções permitidas:** A lista de funções é restrita. `getenv` está na lista, mas ela só consegue buscar uma variável se já soubermos seu nome, o que não nos ajuda a obter a lista *completa*.

A Solução Correta (e Permitida):
A solução está na própria assinatura da função que inicia todo programa em C. A forma canônica e completa da `main` é: `int main(int argc, char **argv, char **envp);`. O terceiro parâmetro, `envp`, é um ponteiro para um array de strings, terminado em `NULL`, contendo todo o ambiente. Por ser um parâmetro de função, **não é uma variável global**, e seu uso é totalmente permitido e a única forma correta de resolver este problema dentro das regras do projeto.

Neste roteiro, vamos dominar o uso de `envp` para construir nosso ambiente.

Exercício 1: O Ponto de Entrada - Acessando `envp`

Objetivo de Aprendizagem: Compreender a assinatura completa da função `main` e como acessar o `envp` para ler a lista de variáveis de ambiente que o sistema operacional passa para o nosso programa no momento da execução.

Explicação Conceitual:

`int main(int argc, char **argv, char **envp)`:

- `argc`: Número de argumentos da linha de comando.
- `argv`: Um array de strings contendo os argumentos.

- **`envp`**: O nosso foco. É um `char **`, ou seja, um **array de ponteiros para char** (um array de strings). Cada string está no formato `"CHAVE=VALOR"`. O final do array é marcado por um ponteiro `NULL`, o que nos permite saber quando parar de lê-lo.

Tarefa:
Escreva um programa C simples, `show_envp.c`, que apenas imprime o conteúdo de `envp`. Este exercício prova que você consegue acessar os dados brutos.

1. Declare sua função `main` com a assinatura completa: `int main(int argc, char **argv, char **envp)`.
2. Para evitar avisos de "variável não utilizada", você pode "silenciar" `argc` e `argv` por enquanto com `(void)argc;` e `(void)argv;`.

Crie um loop `for` ou `while` que percorra `envp`. A condição de parada é quando o ponteiro atual for `NULL`.

- Exemplo com `for`: `for (int i = 0; envp[i] != NULL; i++)`
- Exemplo com `while`: `while (*envp)`

4. Dentro do loop, use `printf` para imprimir cada string (`envp[i]` ou `*envp`) seguida de uma quebra de linha.

Ponto de Reflexão:
Por que passar `envp` como um parâmetro para `main` é considerado uma prática de programação superior e mais segura do que acessar uma variável global como `environ`? Pense em termos de escopo, dependências e previsibilidade do código.

Exercício 2: O Processamento - De String a Chave e Valor

Objetivo de Aprendizagem: Isolar e processar uma única string do ambiente, utilizando as funções da sua `libft` para dividi-la em duas novas strings: uma para a chave e outra para o valor.

Explicação Conceitual:
Nossa Hash Table precisa de `key` e `value` como dados separados. Nós receberemos uma única string como `"PATH=/bin:/usr/bin"`. Precisamos de ferramentas para dividi-la de forma segura.

- **`ft_strchr(const char *s, int c)`:** Sua função da `libft` que imita `strchr`. Ela é perfeita para encontrar a primeira ocorrência do caractere `'='`. Ela retornará um ponteiro para esse caractere dentro da string original.
- **`ft_substr(char const *s, unsigned int start, size_t len)`:** Sua função que extrai uma substring. Usaremos isso para criar uma nova string para a chave, alocando a memória necessária. O comprimento (`len`) será a distância do início da string até a posição do `'='`.
- **`ft_strdup(const char *s1)`:** Sua função que duplica uma string. Ideal para criar uma cópia do valor, que começa logo após o `'='`.

Tarefa:
Escreva uma função auxiliar que recebe uma string de `envp`, a processa e imprime o resultado. Isso ajuda a focar na lógica de parsing antes de integrá-la à Hash Table.

```
// Em um arquivo de teste, por exemplo.
void parse_and_print_var(char *var_string)
{
   char *key;
   char *value;
   char *separator_pos;

   separator_pos = ft_strchr(var_string, '=');
   if (!separator_pos) // Se não houver '=', ignora.
       return;

   // 1. Extraia a chave usando ft_substr.
   // O comprimento é a posição do '=' menos a posição inicial da string.
   key = ft_substr(var_string, 0, separator_pos - var_string);

   // 2. Extraia o valor usando ft_strdup.
   // O valor começa um caractere depois do '='.
   value = ft_strdup(separator_pos + 1);

   // 3. Imprima para verificar.
   printf("Chave: [%s] | Valor: [%s]
```

```
", key, value);

    // 4. Libere a memória alocada, pois este é apenas um exercício.
    free(key);
    free(value);
}
```

Teste esta função chamando-a de `main` com strings fixas como `"USER=teste"` e `"PATH=/bin"`.

Ponto de Reflexão:
Considere um caso especial: `EMPTY_VAR=`. O que `ft_strchr` retorna? O que `ft_strdup(separator_pos + 1)` fará? Ele criará uma string vazia `""` ou resultará em um erro? Como sua lógica deve tratar esse caso para ser robusta?

Exercício 3: A Integração - Populando a Sua Hash Table

Objetivo de Aprendizagem: Unir os conceitos anteriores. Criar a função `env_load` que recebe `envp`, itera sobre ele, e usa a lógica de parsing para popular sua Hash Table com todas as variáveis de ambiente.

Tarefa:
Implemente a função `t_hash_table *env_load(char **envp)` em `utils/env_hash.c` e chame-a a partir do `main.c`.

Em `utils/env_hash.c`:

- Defina a função `t_hash_table *env_load(char **envp)`.
- Dentro dela, primeiro chame `ht_create()` para alocar a tabela. Verifique se a criação foi bem-sucedida.
- Inicie um loop para percorrer `envp`.
- Dentro do loop, para cada string, aplique a lógica de parsing do Exercício 2 para criar `key` e `value` em memória alocada.
- Chame `ht_insert(&sua_tabela, key, value)` para adicionar o par.

- **Gerenciamento de Memória Crucial:** Sua função `ht_insert` deve fazer suas próprias cópias internas dos dados. Portanto, imediatamente após a chamada `ht_insert`, você **deve** usar `free(key)` e `free(value)` para liberar a memória temporária que você alocou nesta função. Isso evita vazamentos de memória.
- Ao final, retorne o ponteiro para a tabela preenchida.

Em `src/main.c`:

- Certifique-se de que a `main` tem a assinatura `int main(int argc, char **argv, char **envp)`.
- Declare uma variável `t_hash_table *env_table;`.
- Chame `env_table = env_load(envp);`.
- Verifique se `env_table` não é `NULL`, tratando o erro se a inicialização falhar.
- Agora você tem a sua Hash Table pronta para ser usada pelo resto do seu shell.

Ponto de Reflexão:
A `t_hash_table` é criada em `env_load` e seu ponteiro é retornado para `main`. Isso estabelece um "contrato": `main` agora é a "dona" dessa estrutura de dados. Qual é a responsabilidade que `main` (ou uma função de limpeza chamada por ela) tem em relação a essa tabela antes que o programa termine? Pense em toda a memória que foi alocada: a própria tabela, o array de `items`, e cada `key` e `value` dentro de cada nó.