Especificação Trabalho Prático - Interpretador de Comandos

Eduardo G. R. Miranda

September 19, 2023

O trabalho prático (TP) deve ser realizado em trios e consiste na implementação de um **Interpretador de Comandos**. Tal interpretador deve ser implementado utilizando a linguagem de programação "C".

1 Instruções

Conforme mencionado, o TP proposto consiste em implementar um interpretador de comandos.

1.1 Valor

A implementação correta que atenda a todos os requisitos especificados abaixo tem o valor de **30** pontos.

1.2 Entrega

A **data máxima de entrega** é: 02/10/2023 às 23:55.

As apresentações ocorrerão no dia 04/10/2023 durante a aula.

Deverá ser submetido no Sistema Integrado de Gestão de Atividades Acadêmicas um arquivo compactado (.zip ou .tgz) contendo:

- o **código-fonte** do programa
- um **Makefile**[1] (para automatizar com o make a compilação via gcc);
- um breve relatório contendo um resumo do projeto, descrevendo a estrutura geral do código, estruturas importantes, decisões relevantes (ex.: como lidou com ambiguidades na especificação), bugs conhecidos ou problemas (lista dos recursos que não implementou ou que sabe que não estão funcionando). Não inclua a listagem do seu código-fonte no relatório; afinal, você já vai entregar o código fonte!

[1] `https://pt.wikibooks.org/wiki/Programar_em_C/Makefiles`

1.3 Grupo

Trabalho prático em grupo, a ser realizado em trio (três alunos). Exceções devem ser previamente comunicadas e aprovadas pelo professor.

1.4 Sobre o Interpretador

Seu programa deverá ser implementado em C. Recursos de outras linguagens (C++, Python, etc) não deverão ser utilizados (na dúvida, pergunte ao professor). Seu programa interpretador de comandos deverá se chamar vesh ("*very elementary shell*") e deverá ser iniciado sem argumentos de linha de comando.

Quando o programa **vesh** for executado (sem argumentos) ele deverá solicitar ao usuário a digitação de comandos, escrever um prompt no início de cada linha na tela (utilize algum símbolo como "$" ou frase como "Digite o comando:") e depois ler os comandos digitados na entrada padrão (teclado via stdin). Mensagens de erro e o resultado dos programas deverão ser exibidos na saída padrão (janela do terminal via stdout ou stderr, conforme o caso). Seu shell deve terminar ao encontrar o comando "sair" no início da linha digitada.

Cada linha digitada deve conter um comando para ser executado. Um novo prompt só deve ser exibido e um novo comando só deve ser lido quando o comando da linha anterior terminar sua execução.

Cada comando a ser executado deve começar com o nome do arquivo binário do programa a ser executado e pode ter um número de argumentos de linha de comando, que serão passados para o novo processo através da interface argc/argv[2] do programa em C. Por exemplo:

```
mkdir nova-pasta
```

2 Informações úteis

2.1 Forma de operação

O seu interpretador deve ser basicamente um loop que exibe o prompt (no modo interativo), lê e interpreta a entrada, executa o comando, espera pelo seu término e reinicia a sequência, até que o fluxo de entrada termine ou o usuário digite fim.

2.2 Execução de comandos

Você deve estruturar o seu interpretador de forma que ele crie pelo menos um novo processo para cada novo comando. A grande vantagem nessa forma de operação é que o interpretador fica protegido de todos os erros que pode ocorrer no novo comando.

[2] `https://www.gnu.org/software/libc/manual/html_node/Program-Arguments.html`

2.3 Lista de comandos mínimos

Espera-se que seu programa execute pelo ao menos a seguinte lista de comandos:

- ls
- mkdir
- man
- pwd
- cd
- rm
- touch
- cp
- mv
- date
- cal
- uname

Aviso-os que ao tentar rodar alguns desses comandos no windows não obtive exito, então sugiro realmente que testem em ambientes linux os códigos e melhor ainda se puderem desenvolver no mesmo.

2.4 Processamento de entrada

Para ler linhas da entrada, você pode querer olhar a função fgets().

Certifique-se de verificar o código de retorno de todas as rotinas de bibliotecas e chamadas do sistema para verificar se não ocorreram erros! (Se você ver um erro, a rotina perror() é útil para mostrar o problema.) Você pode achar o strtok() útil para analisar a linha de comando (ou seja, para extrair os argumentos dentro de um comando separado por espaços em branco).

2.5 Manipulação de argumentos de linha de comando

Os argumentos que são passados para um processo na linha de comando são visíveis para o processo através dos parâmetros da função main():

```
int main (int argc, char * argv []);
```

o parâmetro argc contém um a mais que o número de argumentos passados e argv é um vetor de strings, ou de apontadores para caracteres. Por exemplo, se você disparar um programa com

o programa iniciará sua execução com argc valendo 3 e com os seguintes valores em argv:

argv [0] = "meuprograma"

argv [1] = "205"

argv [2] = "argum2"

OBS.: o primeiro argumento, na posição zero, é sempre o arquivo a ser executado.

Esses argumentos são também utilizados na montagem da chamada da função execvp(), usada para disparar um novo processo com os argumentos fornecidos. Nesse caso, é importante notar que a lista de argumentos deve ser terminada com um ponteiro NULL, ou seja, argv [3] = NULL. É extremamente importante que você verifique bem se está construindo esse vetor corretamente! Por exemplo:

```
#include <unistd.h>
int main(){
    int ret;
    char *cmd[] = { "ls", "-a", (char *)0 };
    ret = execvp ("ls", cmd);
    return 0;
}
```

2.6 Manipulação de processos

Estude as páginas de manual das chamadas do sistema fork(), execvp(), e wait()/waitpid(). O fork() cria um novo processo. Após a chamada dessa função, existirão dois processos executando o mesmo código. Você será capaz de diferenciar o processo filho do pai inspecionando o valor de retorno da chamada: o filho vê um valor de retorno igual a 0, enquanto o pai vê o identificador de processo (pid) do filho.

Você vai notar que há uma variedade de comandos na família exec. Para este trabalho, para facilitar, recomenda-se o uso da execvp(). Lembre-se que se essa chamada for bem sucedida, ele não vai voltar, pois aquele programa deixa de executar e o processo passa a executar o código do programa indicado na chamada. Dessa forma, se a chamada voltar, houve um erro (por exemplo, o comando não existe). A parte mais desafiadora está em passar os argumentos corretamente especificados, como discutido anteriormente sobre argc/argv . As chamadas de sistema wait()/waitpid() permitem que o processo pai espere por seus filhos. Leia as páginas de manual para obter mais detalhes.

2.7 Manipulação de processos

Lembre-se de conseguir fazer funcionar a funcionalidade básica do interpretador antes de se preocupar com todas as condições de erro e os casos extremos. Por exemplo, primeiro faça um único comando funcionar (como ls), depois coloque um comando com argumentos, após isso tente executar de forma iterativa recebendo

o comando via terminal. Finalmente, certifique-se que você está tratando corretamente todos os casos em que haja espaço em branco em torno dos comandos ou comandos que faltam.

É altamente recomendável que você verifique os códigos de retorno de todas as chamadas de sistema desde o início do seu trabalho. Isso, muitas vezes, detecta erros na forma como você está usando essas chamadas do sistema. Exercite bem o seu próprio código! Você é o melhor (e neste caso, o único) testador desse código. Forneça todo tipo de entrada mal-comportada para ele e certifique-se de que o interpretador se comporta bem. Código de qualidade vem através de testes – você deve executar todos os tipos de testes diferentes para garantir que as coisas funcionem como desejado. Não seja comportado – outros usuários certamente não serão. Melhor quebrar o programa agora, do que deixar que outros o quebrem mais tarde.

Mantenha versões do seu código. Ao menos, quando você conseguir fazer funcionar uma parte da funcionalidade do trabalho, faça uma cópia de seu arquivo C ou mantenha diretórios com números de versão. Ao manter versões mais antigas, que você sabe que funcionam até um certo ponto, você pode trabalhar confortavelmente na adição de novas funcionalidades, seguro no conhecimento de que você sempre pode voltar para uma versão mais antiga que funcionava, se necessário. Alternativamente pode-se também utilizar um repositório git como o github para manter o versionamento do código.

Finalmente, embora você possa desenvolver o seu código em qualquer sistema que quiser (POSIX), certifique-se que ele execute corretamente no Linux Ubuntu dos laboratórios de informática/biblioteca ou numa máquina virtual com o sistema operacional Linux Ubuntu versão 16.04. A avaliação do funcionamento do seu código (compilação e execução) será feita neste ambiente.

3 Considerações finais

Este trabalho não é tão complexo quanto pode parecer à primeira vista. Talvez o código que você escreva seja mais curto que este enunciado. Escrever o seu interpretador será uma questão de entender o funcionamento das chamadas de sistema envolvidas e utilizá-las da forma correta. **O programa final deve ter apenas algumas (poucas) centenas de linhas de código.** Se você se ver escrevendo código mais longo que isso, provavelmente é uma boa hora para parar um pouco e pensar mais sobre o que você está fazendo. Entretanto, dominar os princípios de funcionamento e utilização das chamadas para criação de processos, pode exigir algum tempo e esforço.

1. Dúvidas: envie e-mail para eduardomiranda@cefetmg.br ou procure o professor fora do horário de aula (vide horário de atendimento).

2. Comece a fazer o trabalho logo, pois apesar do programa final ser relativamente pequeno, o tempo não é muito e o prazo de entrega não vai ficar maior do que ele é hoje (independente de que dia é hoje).

3. Será valorizado também a clareza, qualidade do código e da documentação e, obviamente, a execução correta com programas de teste.

4 Créditos

O enunciado deste trabalho foi baseado no material do Prof.º Everthon Valadão (IFMG).