Sistemas Operacionais

Trabalho - Implementar uma Shell

1 Introdução

1.1 Informações práticas

Disciplina: Sistemas Operacionais.

Data de entrega: 27/09/2019.

Forma de Entrega: Relatório, código fonte e apresentação individual. O código fonte deve estar contido no relatório em forma de anexo. A apresentação será feita nos dias 26 e 27 de setembro de 2019.

OBS: ***Trabalho individual.***

1.2 Descrição

Uma *shell* é um programa que facilita o uso do computador pelo usuário. A *bash*, por exemplo, uma *shell* usada no linux, é uma programa executável que pode ser encontrada no diretório `/bin`. O caminho completo da *bash* no sistema de arquivos é `/bin/bash.`

Ao executar `/bin/bash` se percebe que ele se executa normalmente, como qualquer outro programa. Ao digitar `exit` da na execução da *bash* se termina a sessão atual e o controle volta para o programa que chamou a *bash*, nesse caso a *bash* executada durante o *login* no sistema.

Ao se logar em um sistema, o programa de *login*, que pede usuário e senha, após autenticar o usuário executa o programa `/bin/bash`, ou outra *shell* configurada na conta do usuário. No caso do linux a *shell* pode ser configurada pelo administrador do sistema (usuário **root**) no arquivo `/etc/passwd`.

1.3 Objetivo do trabalho

Desenvolver uma *shell* simples, em linguagem ANSI C, versão 2011 (C11 - `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`).

2 Funcionamento básico

A *shell* deve iniciar um laço onde o usuário digita o comando a ser executado e a *shell* processa o comando. Existe dois tipos de comandos para uma *shell*

Interno É um comando executado pela própria *shell*.

Externo É um programa externo. A *shell* procura a existência de um programa (arquivo em disco) com o nome do comando digitado. Caso o comando não contenha uma especificação de diretório no início, a *shell* procura o comando nos diretórios especificados pela variável de ambiente `PATH`. Nesse caso um novo processo é criado com a chamada de sistema `fork()`, substitui o código do novo processo com a chamada de sistema `execve()` e espera o processo terminar.

A *shell* deve executar um laço infinito, mostrando como *prompt* o conteúdo da variável de ambiente `PS1`. Caso `PS1` não esteja definda a *shell* deve mostrar **exatamente** a *string* `sh>`, seguido de um espaço.

Um exemplo de código simples a ser usado para desenvolver a *shell* pode ser visto a seguir

Código 1 - shell.c

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAX_LINE 80 /* Tamanho máximo do comando */

char cmd[MAX_LINE+1];

int main(){
  int should_run = 1; /* Marcador para indicar quando o programa deve terminar */
  while (should_run){
    if (ps1_defined()) printf("%s",PS1);
    else  printf("sh> ");
    fflush(stdout);
    cmd = readCmd();
    /* Processar comando: separar nome dos argumentos */
    if (ComandoInterno(cmd_name)){
      /* Executa comando interno */
    } else {
      /* Comando Externo */
      p = fork(); /* Cria novo processo */
      if (p<0){ /* Processo filho não foi criado */
        /* Mostrar erro */
      } else {
        if (p){ /* processo pai */
          wait(NULL); /* Espera pelo filho */
        } else { /* Processo filho */
          processParams(cmd);
          execve(commando,params,NULL);
        }
      }
    }
  }
  exit(0);
}
```

A *shell* deve esperar o usuário digitar uma linha por vez da entrada padrão (`stdin`). A linha é composta de um comando e um conjunto de argumentos. A *shell* deve processar a linha para separar o comando e os argumentos.

Para simplificar você pode:

- Considerar como separador de palavras apenas o espaço (ASCII 32).
- Não considerar caracteres especiais, tipo TAB, Aspas (simples ou duplas - Há uma exceção com relação as aspas duplas com o comando interno `declare`), *backspace*, etc. Isso implica que a sua *shell* não poderá tratar argumentos com espaços no nome. Por exemplo, arquivos e/ou diretórios com espaços no nome não poderão ser tratados pela sua *shell*.
- Definir um número máximo de caracteres por linha de comando (mínimo de 80), também um número máximo de argumentos (mínimo de 5). Considere um número razoável para ambos. Lembre-se que um nome completo de arquivo/diretório pode ser muito longo.

Caso deseje pode fazer a *shell* sem essas limitações. Será levado em conta para a avaliação.

Depois de processar a linha de comando a *shell* deve executá-lo. O comando pode ser o nome de um arquivo executável (externo) ou um comando interno.

Para facilitar a compilação segue um exemplo de `Makefile` que pode ser usado pelo projeto.

Código 2 - Makefile

```
TARGET = sh
LIBS = -lm
CC = gcc
CFLAGS = -Wall -static -ansi

.PHONY: default all clean

default: $(TARGET)
all: default

OBJECTS = $(patsubst %.c, %.o, $(wildcard *.c))
HEADERS = $(wildcard *.h)

%.o: %.c $(HEADERS)
	$(CC) $(CFLAGS) -c $< -o $@

.PRECIOUS: $(TARGET) $(OBJECTS)

$(TARGET): $(OBJECTS)
	$(CC) $(OBJECTS) -Wall $(LIBS) -o $@

clean:
	-rm -f *.o
	-rm -f $(TARGET)
```

2.1 Uso básico

2.2 Variáveis de ambiente

Variáveis de ambiente são usadas pela *shell* ou programas de usuário e podem ser definidas a partir do comando interno `declare`. As duas variáveis de ambiente que sua *shell* deve, obrigatoriamente, considerar durante a execução são:

PS1 Variável que armazena a *string* de *prompt* a ser mostrada no terminal.

PATH Variável que armazena uma lista de diretórios a serem usados na hora de procurar um comando digitado. Os diretórios devem ser separados por dois pontos.

Outras variáveis com outros tipos de configuração/uso podem ser definidas e será avaliada de forma positiva.

2.3 Comandos internos

Os seguintes comandos internos devem ser implementados pela *shell*

pwd Mostra o caminho completo do diretório atual de trabalho do usuário. PWD = *Print Working Directory.*

cd Muda o diretório de trabalho do usuário.

mkdir Cria novo diretório.

rmdir Remove diretório. A remoção só pode acontecer se o diretório estiver vazio.

echo Mostra mensagem de texto.

exit Sai da *shell*

history Mostra os último comandos executados
(`https://www.gnu.org/software/bash/manual/html_node/Bash-History-Builtins.html`), similar o *history* da *bash*, porém mais simples.

1. *history* sem argumentos deve mostrar os últimos 30 comandos digitados, um por linha. Cada linha deve conter um número a esquerda, chamado *offset*, do comando e um espaço entre o número e o comando. Os comandos são numerados a partir de 0 (zero) até 29 (vinte e nove), sendo 0 (zero) o comando mais antigo.
2. `history -c` deve apagar todos os comandos do histórico.
3. `history [offset]` deve executar o comando de número `[offset]`. A *shell* deve mostrar um erro caso o número não seja válido.
4. Exemplo de uma saída do history:

```
sh> history
0 /bin/ls
1 cd /tmp
2 /bin/ls
3 cat teste.txt
4 cd /home/jorgiano
```

declare Define uma variável de ambiente. Sintaxe: `declare VARNAME=VALUE`.
Exemplo 1: `declare PS1="$ > "`
Exemplo 2: `declare PATH=/bin:/usr/bin:/sbin`

O comando `declare` sem parâmetros deve listar todas as variáveis de ambiente que existem.

unset Remove uma variável de ambiente. Sintaxe: `unset VARNAME`.

2.4 Requisitos adicionais

- Todo o código deve ser desenvolvido em ANSI C.
- A *shell* deve ser compilada com os parâmetros
  `gcc -Wall -static -ansi`
- Códigos compilados com *warnings* perdem pontos. Fique a vontade para usar uma ferramenta de compilação automática (make, ninja, gradle). O uso dessas ferramentas será vista de forma positiva durante a avaliação.
- O programa não deve causar *memory leaks* e deve tratar erros de forma adequada.

3 Erro

A mensagem de erro a ser mostrada deve obedecer a seguinte sintaxe:

```
error: MESSAGE
```

Exemplo de erro ao tentar mudar de diretório:

```
error: Diretório XXX não existe!
```

Para tal considere a seguinte linha, em ANSI C, para mostrar a mensagem:

```
printf("error: %s\n",errormsg)
```

onde `errormsg` é uma *string* contendo a mensagem de erro. Para erros obtidos a partir de execução de programas externos considere a função `strerror(errno)`, que retorna uma *string* com a mensagem de erro, se houver. A variável global `errno` contém o código do erro se o mesmo acontecer ao executar uma chamada de sistema. Verifique o retorno das funções que utilizam recursos do sistema. Não considere que toda requisição de memória será bem sucedida (`malloc` pode não conseguir alocar a quantidade de memória solicitada). O seu programa deve verificar e informa ocorrência de erro.

4 Dicas

- Use as *man pages*para acessar a documentação das funções. As *pages* 2 possuem documentação das chamadas de sistema e as *pages* 3 as das bibliotecas. Por exemplo, para ler a documentação da chamada de sistema `fork()` digite `man 2 fork`. Para a documentação da função `printf()` digite `man 3 printf`.
- Para dúvidas de utilização do linux acesse verifique o tutorial de linux disponível no SUAP ou use a Internet (Google/Yahoo/etc).
- A maioria dos problemas que vocês encontrarão possivelmente outras pessoas também encontraram :) Use o google para procurar respostas. O site *stackoverflow* (http://pt.stackoverflow.com) possui perguntas e respostas sobre programação e pode ser útil para suas dúvidas. A versão em inglês é mais completa.

5 Entrega

O trabalho deve ser entregue em forma de relatório e códigos fonte. Os códigos fontes também devem estar no relatório como anexo.

O relatório deve conter uma explicação do que é uma *shell* e como ela faz para executar programas digitados pelo usuário (comandos externos), detalhando e explicando a parte do código que realiza o processamento. Todos os comandos internos devem ser explicados, tanto o seu uso como a implementação.

Para as variáveis de ambiente o relatório deve explicar as estruturas usadas. Também explicar como usá-las. É necessário explicar as variáveis `PS1` e `PATH`. Exemplo de organização do relatório:

```
1 - Introdução
2 - O interpretador de comandos
  2.1 - Interação com usuário
  2.2 - Comandos internos
        (uma subseção para explicar cada comando)
  2.3 - Comandos externos
  2.4 - Variáveis de ambiente
        (Detalhar aqui as variáveis que a sua shell usa - PS1, PATH, etc)
3 - Implementação
  3.1 - Estruturação do código
        Explicar os arquivos fontes e as funções existentes em cada um
  3.2 - Interpretação de comandos
      3.2.1 - Decomposição do comando
      3.2.2 - Execução de comando externo
              EXPLICAR:
              Onde e como a shell procura o comando
              Chamada de sistema fork()
              Chamada de sistema execve()
      3.2.4 - Execução de comando interno
  3.3 - Estruturas de variáveis de ambiente
4 - Conclusão
```

Esta organização é apenas uma sugestão, fique a vontade para modificar.