MINISTÉRIO DA EDUCAÇÃO
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE MINAS GERAIS
IFMG CAMPUS FORMIGA

CURSO: BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA: "**SISTEMAS OPERACIONAIS**"
PROF.º EVERTHON VALADÃO everthon.valadao@ifmg.edu.br

TRABALHO 1 — Interpretador de comandos (`shell`)

VALOR: 10 pontos

DATA (máxima) de entrega**:** 14/10/2022 via https://classroom.google.com

IMPORTANTE, leia atentamente as instruções a seguir:

INSTRUÇÕES

Trabalho prático em grupo, a ser realizado em trio ou dupla de alunos (ou, excepcionalmente, individualmente). Exceções devem ser previamente comunicadas e aprovadas pelo professor.

O trabalho deve ser implementado com programação em ambiente Unix/Linux[1], utilizando a biblioteca C da GNU[2] (libc6-dev) para invocar os recursos POSIX[3]. Evite utilizar desnecessariamente bibliotecas de terceiros, para não sacrificar a portabilidade do código.

Deverá ser submetido através do https://classroom.google.com um arquivo compactado (.zip ou .tgz) contendo:

- o **código-fonte** do programa (.c e .h);
- um **Makefile**[4] (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!

GRUPO

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

ENUNCIADO

Este trabalho tem por objetivo fazer com que os alunos experimentem na prática a programação em ambiente Unix/Linux com o padrão POSIX, para o tratamento de processos e sua comunicação através de **`pipes`** (canais de envio de bytes).

O PROBLEMA

Deverá ser projetado e implementado um interpretador de comandos, conhecido no Linux como **`shell`**. Para tal, será necessário disparar processos e encadear sua comunicação usando pipes. O processos produtores/consumidores poderão trocar dados dentro do **`shell`**, seja para fornecer os dados de entrada para um programa ou para receber os dados de saída de outro programa.

ATENÇÃO

Não será tolerado plágio. A critério do professor, será realizada entrevista com os membros do grupo.

IMPLEMENTAÇÃO

Seu programa deverá ser implementado em C, utilizando a biblioteca GNU libc6-dev (ou posterior). Recursos de outras linguagens (C++, Python, etc) não deverão ser utilizados (na dúvida, pergunte ao professor). O arquivo binário compilado do seu programa interpretador de comandos deverá se chamar **`rush`** ("*rudimentary shell*") e ele deverá ser iniciado sem argumentos de linha de comando.

Quando o programa **`rush`** 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 "`exit`" no início da linha digitada.

Cada linha digitada deve conter um comando ou mais comandos para serem executados. Quando houver mais de um programa a ser executado na linha digitada, eles devem obrigatoriamente ser encadeados por **`pipes`** (indicado pelo símbolo "`|`"), indicando que a saída do programa à esquerda deve ser associada à entrada do programa à direita.

---

[1] The Linux Programming Interface: https://github.com/kevin-leptons/tlpiexer/blob/master/doc/the-linux-programming-interface.pdf

[2] GNU C Library: https://www.gnu.org/software/libc/manual/html_node/index.html

[3] POSIX.1-2017: http://pubs.opengroup.org/onlinepubs/9699919799/toc.htm (SUSv4, ANSI/IEEE Std 1003.1-2017, ISO/IEC 9945)

[4] Makefile tutorial: https://pt.wikibooks.org/wiki/Programar_em_C/Makefiles

Um novo *prompt* só deve ser exibido e um novo comando só deve ser lido quando o comando da linha anterior terminar sua execução. O interpretador deve sempre receber o valor de saída dos programas executados, vide as chamadas `wait()/waitpid()` para esse fim.

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`[5] do programa em C. Por exemplo:

```
$ uname -v
```

Seu interpretador de comandos deverá aceitar os comandos de redirecionamento de entrada e saída normalmente utilizados (no bash): "<" e ">", que indicarão entrada e saída por pipes para processos produtores/consumidores. O símbolo ">" indica que o processo filho deverá abrir (`fopen` ou `open`) o arquivo indicado após o símbolo ">" e conectar (`dup2`) sua saída padrão ao descritor de arquivo (`fileno`), antes de executar (`execvp()`) o comando daquele processo filho. Se o arquivo não existir ele deve ser criado mas, se já existir, ele deve ser sobrescrito ("w" ao invés de "w+") com o novo conteúdo. Por exemplo:

```
$ echo "ola mundo" > arquivo.txt
$ seq 1 9 | wc > arquivo.txt
```

De forma equivalente, um nome de arquivo depois de um comando e separado dele por um "<" indica que o interpretador de comandos deve conectar a entrada padrão do processo ao descritor daquele arquivo antes de executar aquele processo filho. Nesse caso, o arquivo deve existir previamente (ou o interpretador deve indicar um sinal de erro). Por exemplo:

```
$ cat -n < arquivo.txt
$ cat -n < arquivo.txt | sort
```

INFORMAÇÕES ÚTEIS

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.

Execução de comandos

Você deve estruturar o seu interpretador de forma que ele crie pelo menos um novo processo para cada novo comando. Existem duas vantagens nessa forma de operação. Primeiro, ele protege o interpretador de todos os erros que pode ocorrer no novo comando. Além disso, permite expressar concorrência de forma fácil, isto é, vários comandos pode ser disparados para executar simultaneamente (concorrentemente). Isso é importante para se criar os módulos produtor e consumidor que serão necessários para manipular arquivos de entrada e saída e essencial para implementar o comando pipe.

Acesso às páginas de manual

Para encontrar informações sobre as rotinas da biblioteca padrão e as chamadas do sistema operacional, consulte as páginas de manual online do sistema (usando o comando Unix `man`). Você também vai verificar que as páginas do manual são úteis para ver que arquivos de cabeçalho que você deve incluir em seu programa. Em alguns casos, pode haver um comando com o mesmo nome de uma chamada de sistema; quando isso acontece, você deve usar o número da seção do manual que você deseja: por exemplo, `man read` mostra a página de um comando da *shell* do Linux, enquanto `man 2 read` mostra a página da chamada do sistema.

Processamento de entrada

Para ler linhas da entrada, você pode querer olhar a função `fgets()`. Para abrir um arquivo e obter um identificador com o tipo `FILE *`, consulte o manual sobre `fopen()`. Note, entretanto, que funções que manipulam o tipo `FILE *` são da biblioteca padrão, não chamadas de sistema. Estas últimas manipulam um inteiro como descritor de arquivo, o que será importante nos casos de manipulação de pipes. Por exemplo, observe as páginas de manual para a função `fopen()` e a chamada de sistema `open()`.

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).

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

```
meuprograma 205 argum2
```

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>
//...
char* command = "ls";
char* arg_list[] = { "ls", "-a", NULL};
//...
int return_sig = execvp (command, arg_list);
```

---

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

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, recomendamos você use `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.

Uso de pipes

Para os comandos de manipulação de arquivos, você vai ter que criar um pipe que ligue o processo criado ao se disparar o comando indicado com outro processo que execute o código de um produtor ou consumidor, dependendo do controle usado. Verifique a página de manual das primitivas `pipe()` e `dup2()/dup()` para ver os detalhes, inclusive com o exemplo de um código que usa a chamada para criar um canal de comunicação entre dois processos, pai e filho.

Processo de desenvolvimento

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 foque no modo interativo e faça funcionar um único comando em execução (provavelmente primeiro um comando sem argumentos, como `ls`). Em seguida, tentar trabalhar com os comandos de leitura e escrita de arquivos, um por vez, e com os pipes entre comandos. Finalmente, certifique-se que você está tratando corretamente todos os casos em que haja espaço em branco em torno de diversos 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.

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 ou numa máquina virtual com o sistema operacional Linux Ubuntu versão 20.04 LTS. A avaliação do funcionamento do seu código (compilação e execução) será feita neste ambiente.

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, manipulação da entrada e saída padrão de cada processo e de criação de pipes pode exigir algum tempo e esforço.

1. 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).
2. Será valorizado também a clareza, qualidade do código e da documentação e, obviamente, a execução correta com programas de teste.
3. Dúvidas: envie e-mail para everthon.valadao@ifmg.edu.br e agende um horário de atendimento online (via Google Meet). Consulte a agenda online do professor.

Créditos

O enunciado deste trabalho foi baseado no material do Prof.º Dorgival Olavo Guedes Neto (UFMG).