UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
SISTEMAS OPERACIONAIS

1º semestre de 2018
Professor: Rafael Sachetto Oliveira

Trabalho Prático 1
Data de Entrega: 1 de Maio 2018. Trabalho em Trio (Máximo)

Este trabalho tem por objetivo fazer com que os alunos experimentem na prática o ambiente de programação Unix/Linux para tratamento de processos e para a comunicação entre processos por pipes (canais de envio de bytes).

1 O problema

Neste trabalho você deve implementar um interpretador de comandos (chamado no Unix/Linux de shell). Para isso, você deverá aprender como disparar processos e como encadear sua comunicação usando pipes. Além disso, você deverá utilizar pipes e manipulação de processos para criar um par produtor/consumidor por envio de mensagens que trabalhará dentro da shell para "alimentar"programas que você disparar com dados, ou para receber os dados de um programa.

2 Implementação

Seu programa deverá ser implementado em C (não serão aceitos recursos de C++, como classes da biblioteca padrão ou de outras fontes).

O programa, que deverá se chamar shellso deverá ser iniciado sem argumentos de linha de comando, ou com apenas um. Caso seja disparado sem argumentos, ele deverá escrever um prompt no início de cada linha na tela (um símbolo como "$"ou uma mensagem como "Sim, mestre?") e depois ler comandos da sua entrada padrão (normalmente, o teclado). Mensagens de erro e o resultado dos programas, salvo quando redirecionados pelos comandos fornecidos, devem ser exibidos na saída padrão (usualmente, a janela do terminal). Essa forma de operação é denominada interativa. Já se um parâmetro for fornecido, ele deve ser interpretado como o nome de um arquivo de onde comandos serão lidos. Nesse caso, apenas o resultado dos comandos deve ser exibido na saída padrão, sem a exibição de prompts nem o nome dos comandos executados.

Em ambos os modos de operação, sua shell termina ao encontrar o comando "fim"no início de uma linha ou ao encontrar o final do fluxo de bytes de entrada (ao fim do arquivo de entrada, ou se o usuário digita Ctrl-D no teclado).

Cada linha deve conter um comando ou mais comandos para ser(em) executado(s). No caso de haver mais de um programa a ser executado na linha, eles devem obrigatoriamente ser encadeados por pipes (|), indicando que a saída do programa à esquerda deve ser associada à entrada do programa à direita. Um novo prompt só deve ser exibido (se necessário) e um novo comando só deve ser lido quando o comando da linha anterior terminar sua execução, exceto caso a linha de comando termine com um "&";, quando o programa deve ser executado em background. Em qualquer caso, o interpretador deve sempre receber o valor de saída dos programas executados – isto é, ele não deve deixar zumbis no sistema (confira as chamadas de sistema wait() e waitpid() para esse fim). Para o caso dos programas executados em background, você deve se informar sobre o tratamento de sinais, em particular o sinal SIGCHLD, para tratar o término daqueles programas.

Cada programa a ser executado deve começar com o nome do arquivo de 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 do programa em C. Como mencionado anteriormente, diversos processos podem ser encadeados usando pipes. O interpretador deve também aceitar

linhas vazias, para as quais nada deve ser executado e simplesmente deve-se iniciar uma nova linha de comando.

O seu interpretador não aceitará os comandos de redirecionamento de entrada e saída normalmente utilizados, "<" e ">". Ao invés disso, ele aceitará os símbolos "=>" e "<=", que indicarão entrada e saída por pipes para processos produtores/consumidores. O sinal "=>" indica que o interpretador de comandos deverá conectar um pipe à saída padrão do processo que vai ser executado e disparar um processo para ler desse pipe para escrever o que for enviado pelo processo para o arquivo indicado (como um processo consumidor do pipe). Se o arquivo não existir, ele deve ser criado; se já existir, ele deve ser sobre-escrito com o novo conteúdo). Um nome de arquivo depois de um comando e separado dele por um "<=" indica que o interpretador de comandos deve conectar um pipe à entrada padrão do processo e disparar um outro processo para ler o arquivo e enviar pelo pipe o conteúdo do arquivo indicado. Nesse caso, o arquivo deve existir (ou o interpretador deve indicar um sinal de erro). Os dois redirecionamentos podem ser usados ao mesmo tempo em uma mesma linha de comando. Caso a linha inclua mais de um comando, obrigatoriamente o "<=" deve ser associado ao primeiro programa na linha e o "=>" deve ser associado ao último.

Com base nessa descrição, são comandos válidos (supondo que o prompt seja "Qual o seu comando?":

```
Qual o seu comando? ls -l
Qual o seu comando? ls -laR => arquivo
Qual o seu comando?
Qual o seu comando? wc -l <= arquivo &
Qual o seu comando? cat -n <= arquivo => arquivonumerado
Qual o seu comando? cat -n <= arquivo | wc -l => numerodelinhas
Qual o seu comando? cat -n <= arquivo | wc -l => numerodelinhas &
Qual o seu comando? fim
```

A terceira linha ilustra o caso da linha de comando vazia. Note que o comando fim não é um programa que será executado, mas um comando embutido (built-in) do interpretador. Cada parte do comando deve ser separada das demais por pelo menos um espaço em branco, ou caracteres de tabulação, inclusive os sinais de uso de arquivos.

3 Programação defensiva

Uma vez que seu programa esteja funcionando, você deve se certificar de que o código seja robusto contra erros do usuário (afinal, um interpretador de comandos não deve parar de executar inesperadamente). Condições que você deve testar e tratar adequadamente (com uma mensagem de erro, quando for o caso) incluem: o interpretador de comandos ser iniciado com um número errado de argumentos ou com um nome de arquivo que não existe (o programa deve exibir uma mensagem de erro e terminar); um comando ou um arquivo de entrada que não existem, ou um arquivo de saída que não pode ser escrito por algum motivo (o programa deve exibir uma mensagem de erro e voltar para o prompt, para o próximo comando). Você pode considerar que linhas de comando nunca serão maiores que 512 caracteres, incluindo o caractere de fim de linha (`"\n"`), e que nenhum componente da linha (nomes de programas, argumentos ou nomes de arquivos) não ultrapassará 64 caracteres.

4 Informações úteis

4.1 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.

4.2 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.

4.3 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.

4.4 Processamento da 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).

4.5 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 umprograma 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". 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!

4.6 Manipulação de processos

Estude as páginas de manual das chamadas do sistema fork(), execvp(), e esperar/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 e argv. As chamadas de sistema wait() e waitpid() permitem que o processo pai espere por seus filhos. Leia as páginas de manual para obter mais detalhes.

4.7 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 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.

4.8 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, adicione a funcionalidade de trabalhar em modo de arquivo de entrada (a maioria dos testes vão usar esse modo, então certifique-se de que ele funciona). 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 correctamente 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. Programadores mais avançados utilizam um sistema de controle de versões, tal como GIT, SVN ou Mercurial. 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.

4.9 O que deve ser entregue

Você deve entregar no moodle um arquivo .zip ou .tgz contendo o(s) arquivo(s) contendo o código fonte do programa (.c e .h), um Makefile e um relatório sobre o seu trabalho, que deve conter:

Um resumo do projeto: alguns parágrafos que descrevam a estrutura geral do seu código e todas as estruturas importantes. Decisões de projeto: descreva como você lidou com quaisquer ambiguidades na especificação. Por exemplo, para este projeto, explicar como seu interpretador lida com linhas que não têm comandos, apenas manipulação de arquivos. Bugs conhecidos ou problemas: uma lista de todos os recursos que você não implementou ou que você sabe que não estão funcionando corretamente Finalmente, embora você possa desenvolver o seu código em qualquer sistema que quiser, certifique-se que ele execute corretamente na máquina virtual com o

sistema operacional Linux que foi distribuída no início do curso. A avaliação do seu funcionamento será feita naquele ambiente.

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

Dúvidas: procurem o professor no gabinete. 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). Vão valer pontos clareza, qualidade do código e da documentação e, obviamente, a execução correta da chamada do sistema com programas de teste.