Sistemas Operacionais

Trabalho de Programação

Período: 2020/2-Earte

Data de Entrega: 02/05/2021 ***Composição dos Grupos: até 3 pessoas***

Material a entregar

Um arquivo compactado com o seguinte nome "**nome_do_grupo.extensão**" (*ex: joao_maria_jose.rar*). Esse arquivo deverá conter todos os arquivos (incluindo *makefile*) criados, com o código muito bem comentado. **Atenção: adicionar nos comentários iniciais do makefile a lista com os nomes completos dos componentes do grupo!**

Valendo ponto: clareza, indentação e comentários no programa.

DESCRIÇÃO DO TRABALHO

Vocês devem implementar na linguagem C um shell denominado `vsh` (*vaccine shell*) para colocar em prática os princípios de manipulação de processos.

Ao iniciar, `vcsh` exibe seu *prompt* (os símbolos no começo de cada linha indicando que o *shell* está à espera de comandos). Quando ele recebe uma linha de comando do usuário, é iniciado seu processamento. Primeiro, a linha deve ser interpretada em termos da linguagem de comandos definida a seguir e cada comando identificado deve ser executado. Essa operação possivelmente levará ao disparo de novos processos.

Um primeiro diferencial do `vsh` é que, na linha de comando, o usuário pode solicitar a criação de um ou mais processos:

Exemplo 1 (um processo):

```
vsh> comando1
```

Exemplo 2 (três processos):

```
vsh> comando1 | comando2 | comando3
```

Quando é passado apenas um comando (como no Exemplo 1 acima), a vsh terá um comportamento similar ao dos shells tradicionais, isto é, ela criará um processo em foreground e ficará esperando esse filho terminar para então exibir o prompt novamente.

Mas ao contrário dos shells UNIX tradicionais, ao processar um conjunto de comandos na mesma linha (como no Exemplo) os processos criados para executar os respectivos comandos deverão todos ser "irmãos" e rodar em background, em uma sessão separada da sessão do `vsh`. Assim, no exemplo 2 acima, a `vsh` deverá criar 3 processos – P1 , P2 e P3 – para executar os comandos comando1, comando2 e comando 3 respectivamente (comando X trata-se de um "comando externo", como será melhor explicado). E esses três processos deverão ser "irmãos" e pertencer a uma mesma sessão, que seja diferente da sessão da `vsh`.

O número de comandos externos passados em uma mesma linha de comando pode variar de 1 a 5… (sem aglomerações)! Com isso em uma mesma sessão só deverá haver processos que tenham sido criados durante uma mesma linha de comando. Notem que um processo de *background* nunca é criado isoladamente. Vejam abaixo:

```
vsh> ls -l //foreground
vsh> xcalc //foreground
vsh> ls -l | grep batata //background
vsh> cat sample | grep -v a | sort - r //background
```

Vale ressaltar que os processos criados em *background* não estarão associados a nenhum terminal de controle, consequentemente:

- os processos criados pela `vsh` não devem receber nenhum sinal gerado por meio de teclas do terminal (ex: Ctrl-c);
- após o usuário entrar uma linha de comando como no exemplo 2 a `vsh` retorna imediatamente para receber novos comandos.

Outra particularidade da `vsh` é que assim como o resto do mundo neste momento de pandemia, ela está muito preocupada com seus processos... é que existem "duas cepas" de sinais "circulando" no sistema que são "letais" para os processos: SIGUSR1 e SIGUSR2. Mas há uma solução para esse problema… uma vacina!! Só que infelizmente ainda não há vacina para todos (incompetência de quem?!? … nem vamos discutir!!). Enfim, nesse sistema apenas os processos "criados para trabalhar na linha de frete", isto é, os processos criados em foreground, serão "vacinados" contra os sinais SIGUSR1 e SIGUSR2. Infelizmente, os processos de background não terão a mesma sorte, e continuarão vulneráveis a esses sinais.

Além de letais, SIGUSR1 e SIGUSR2 são muito contagiosos!! Se algum processo de background morrer devido a um desses sinais, os demais processos irmãos também deverão morrer devido ao mesmo sinal. Mas observem que se existirem outros processos de *background* que tenham sido criados em outras linhas de comando, eles NÃO deverão morrer, uma vez que eles estarão isolados em sessões diferentes (se não tem vacina, mantenham o isolamento!).

Linguagem da vsh

A linguagem compreendida pela `vsh` é bem simples. Cada sequência de caracteres diferentes de espaço é considerada um termo. Termos podem ser:

i. **operações internas do shell,**
ii. **operadores especiais,**
iii. **nomes de programas a serem executados (comandos externos),**
iv. **argumentos a serem passados para operações internas ou comandos externos.**

Operações internas do shell são sequências de caracteres que devem sempre ser executadas pelo próprio shell e não resultam na criação de novos processos. No `vsh` as operações internas são:

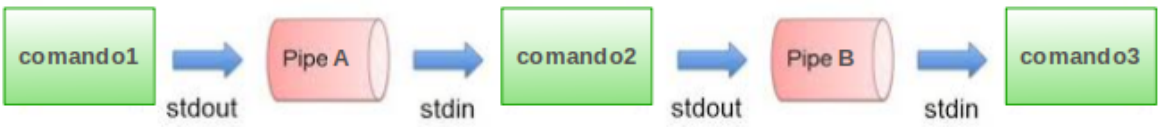
`liberamoita`: faz com que o shell libere todos os seus descendentes (diretos e indiretos, isto é, filhos, netos, bisnetos, etc.) que estejam no estado "Zombie" antes de exibir um novo prompt.

`armageddon`: termina a operação do shell, mas antes disso, ele deve matar todos os seus descendentes (diretos e indiretos, isto é, filhos, netos, bisnetos, etc.) que ainda estejam rodando.

Essas operações internas devem sempre terminar com um sinal de fim de linha (return) e devem ser entradas logo em seguida ao prompt (isto é, devem sempre ser entradas como linhas separadas de quaisquer outros comandos).

Quanto aos **operadores especiais**, há apenas um: o símbolo '`|`'. Ele tem duas finalidades, como mostrado nos exemplos, ele é utilizado para permitir que diferentes comandos externos possam ser passados mesma linha de comando, fazendo com que uma "família" de processos de background sejam criados. A segunda finalidade é que, assim como os shells tradicionais, esse símbolo também funciona como um "pipe" (ou tubo) para redirecionar as entradas e saídas padrões dos processos … vocês ainda não viram o que é isso, mas verão em breve no curso… Mas em resumo teremos o seguinte:

> Considere o Exemplo 3:
>
> ```
> vsh> comando1 | comando2 | comando3
> ```
>
> Aqui, o símbolo '|' entre cada comando indica que será criado um pipe (que é um tipo de arquivo temporário especial). No exemplo acima, a saída padrão do processo que irá rodar o "comando1" deverá ser redirecionada para o primeiro pipe (vamos chamá-lo de "Pipe A"). Assim como a entrada padrão do processo que irá rodar "comando2" deverá ser redirecionada para esse mesmo "Pipe A". Além disso, a saída padrão do processo que irá rodar o "comando2" deverá ser redirecionada para o segundo pipe (vamos chamá-lo de "Pipe B"). Assim como a entrada padrão do processo que irá rodar "comando3" deverá ser redirecionada para esse mesmo "Pipe B". A figura a seguir resume o uso dos pipes para o Exemplo 3:



Programas a serem executados são identificados pelo nome do seu arquivo executável e podem ser seguidos por um número máximo de três argumentos (parâmetros que serão passados ao programa por meio do vetor `argv[]`). Em uma mesma linha de comando, pode haver um conjunto máximo de 5 programas (com seus respectivos argumentos) separados pelo operador '|'.

Com isso cada linha de comando pode conter:

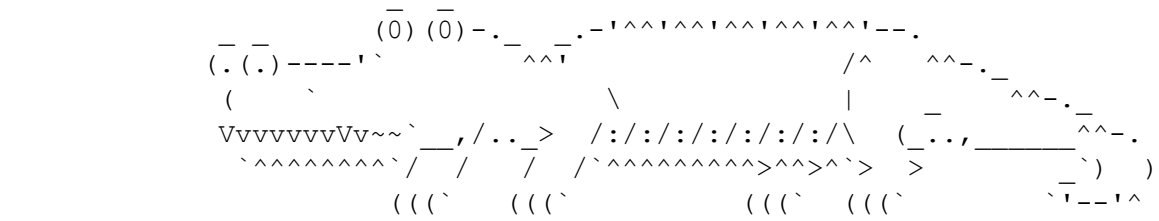
- uma operação interna da `vsh`;
- um comando externo (nome de um programa executável e seus argumentos);
- dois a cinco comandos externos (nomes de programas executáveis e seus argumentos) separados pelo símbolo '|'.

Tratamento de Sinais

Como a vsh também é um processo da "linha de frente", ela também estará vacinada contra o SIGUSR1 e SIGUSR2! Mas ela tomou uma vacina diferente (origem em algum país do oriente) dos demais processos da linha de frente… ficando imune, mas com alguns efeitos colaterais. Com isso,

a vsh deverá **capturar** esses dois sinais e, toda vez que ela receber um desses sinais, ela irá imprimir a seguinte mensagem:

```
vsh>
```



```
I feel weird...
```

Além disso, durante a execução do tratador desses sinais, os seguintes sinais que podem ser gerados via comando especial "Ctrl-..." (isto é, Ctrl-C (SIGINT), Ctrl-\ (SIGQUIT), Ctrl-Z (SIGTSTP)) devem estar bloqueados, para evitar que o tratamento dos sinais SIGUSR1 e SIGUSR2 seja interrompido no meio.

Dicas técnicas

Este trabalho exercita as principais funções relacionadas ao controle de processo, como `fork`, `execvp`, `wait`, entre outras. Certifique-se de consultar as páginas de manual a respeito para obter detalhes sobre os parâmetros usados, valores de retorno, condições de erro, etc (além dos slides da aula sobre SVCs no UNIX).

Outras funções que podem ser úteis são aquelas de manipulação de strings para tratar os comandos lidos da entrada. Há basicamente duas opções principais: pode-se usar `scanf("%s")`, que vai retornar cada sequência de caracteres delimitada por espaços, ou usar `fgets` para ler uma linha de cada vez para a memória e aí fazer o processamento de seu conteúdo, seja manipulando diretamente os caracteres do vetor resultante ou usando funções como `strtok`.

Ao consultar o manual, notem que as páginas de manual do sistema (acessíveis pelo comando `man`) são agrupadas em seções numeradas. A seção 1 corresponde a programas utilitários (comandos), a seção 2 corresponde às chamadas do sistema e a seção 3 às funções da biblioteca padrão. Em alguns casos, pode haver um comando com o mesmo nome da função que você procura e a página errada é exibida. Isso pode ser corrigido colocando-se o número da seção desejada antes da função, por exemplo, "`man 2 fork`". Na dúvida se uma função é da biblioteca ou do sistema, experimente tanto 2 quanto 3. O número da seção que está sendo usada aparece no topo da página do manual.

Ah! Claro! Muitos problemas podem ocorrer a cada chamada de uma função da biblioteca ou do sistema. Certifique-se de testar cada valor de retorno das funções e, em muitos casos, verificar também o valor do erro, caso ele ocorra. Isso é essencial, por exemplo, no uso da chamada `wait`. Além disso, certifique-se de verificar erros no formato dos comandos, no nome dos programas a serem executados, etc. Um tratamento mais detalhado desses elementos da linguagem é normalmente discutido na disciplina de compiladores ou linguagens de programação, mas a linguagem usada neste trabalho foi simplificada a fim de não exigir técnicas mais sofisticadas para seu processamento.

BIBLIOGRAFIA EXTRA

Kay A. Robbins, Steven Robbins, *UNIX Systems Programming: Communication, Concurrency and Threads, 2nd Edition* (Cap 1-3).

ALGUNS CONCEITOS IMPORTANTES

Processos em Background no Linux

No linux, um processo pode estar em *foreground* ou em *background*, ou seja, em primeiro plano ou em segundo plano. A opção de se colocar um processo em *background* permite que o shell execute tarefas em segundo plano sem ficar bloqueada, de forma que o usuário possa passar novos comandos para o ele.

Quando um processo é colocado em *background*, ele ainda permanece associado a um terminal de controle. No entanto, quando um processo tenta ler ou escrever no terminal , o kernel envia um sinal SIGTTIN (no caso de tentativa de leitura) or SIGTTOU (no caso de tentativa de saída). Como resultado, o processo é suspenso.

Por fim, um processo de *background* não recebe sinais gerados por combinações de teclas, como Ctrl-C (SIGINT), Ctrl-\ (SIGQUIT), Ctrl-Z (SIGTSTP). Esses sinais são enviados apenas a processos em foreground criados pelo shell.

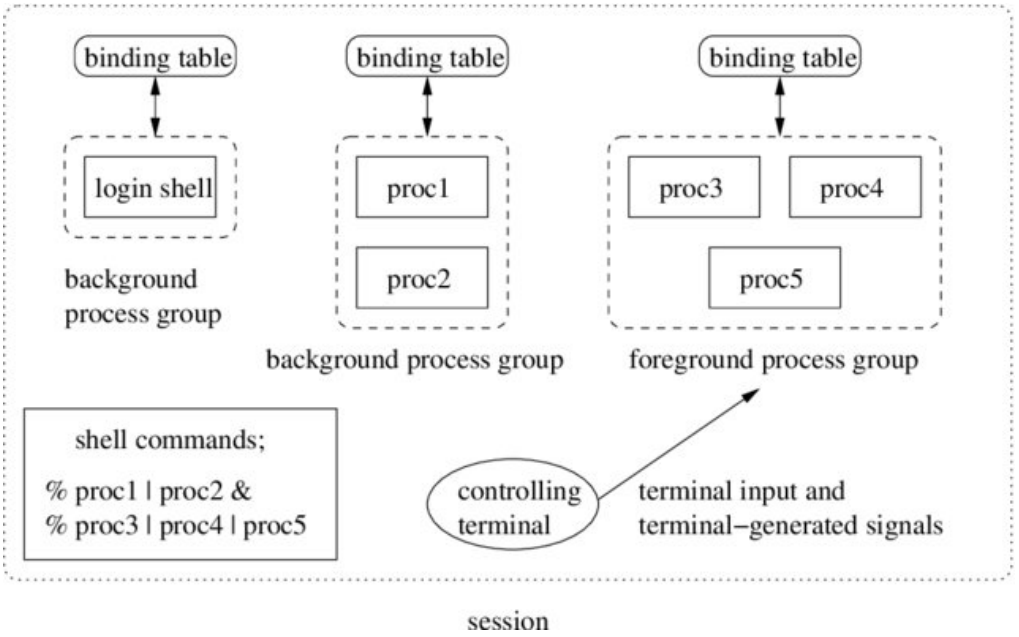


Fig 1: Relação entre processos, grupos, sessões e terminal de controle

Grupos e Sessões no Linux

Como vocês já viram em laboratórios passados, o Unix define o conceito de ***Process Group***, ou Grupo de Processos. Um grupo nada mais é do que um conjunto de processos.

Isso facilita principalmente a vida dos administradores do sistema no envio de sinais para esses grupos. É que usando a chamada `kill()` é possível não somente enviar um sinal para um processo específico, mas também enviar um sinal para todos os processos de um mesmo *Process Group*. Como também já foi visto, quando um processo é criado, automaticamente ele pertence ao mesmo *Process Group* do processo pai (criador), sendo possível alterar o grupo de um processo por meio da chamada `setpgid()`. A bash, por exemplo, quando executa um comando de linha, ela faz `fork()` e logo em seguida é feito uma chamada a `setpgid()` para alocar um novo *Process Group* para esse processo filho. Se o comando for executado sem o sinal '&', esse *process group* é setado para *foreground*, enquanto o grupo da bash vai para *background*. A figura acima ilustra como ficam os grupos após os comandos ilustrados no quadro "shell commands".

- Após a linha de comando "proc1 | proc2 &", a bash cria dois processos em *background* e um *pipe*, e redireciona a saída padrão de proc1 para o *pipe*, e a entrada padrão de proc2 para esse mesmo *pipe*.
- Após a linha de comando "proc1 | proc2 | proc3 ", a bash cria três processos em *foreground* e dois *pipes*, e redireciona a saída padrão de proc1 para o 1o. *pipe*, e a entrada padrão de proc2 para esse mesmo *pipe*; também redireciona a saída padrão de proc2 para o 2o *pipe*, e a entrada padrão de proc3 para esse 2o. *pipe*.

Agora que vocês já estão feras em *Process Groups*, vamos ao conceito de ***Session***, ou Sessão. Uma sessão é uma coleção de grupos. Uma mesma sessão pode conter diferentes grupos de *background*, mas no máximo 1 (um) grupo de *foreground*. Com isso, uma sessão pode estar associada a um terminal de controle que por sua vez interage com os processos do grupo de *foreground* desta sessão. Quando um processo chama `setsid()`, é criada uma nova sessão (sem nenhum terminal de controle associado a ela) e um novo grupo dentro dessa sessão. Esse processo se torna o líder da nova sessão e do novo grupo.