Sistemas Operacionais

Trabalho de Programação

Período: 2021/2-Earte

Data de Entrega: 27/02/2022 ***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 uma shell denominada `psh` (*passaport shell*) para colocar em prática os princípios de manipulação de processos.

Ao iniciar, `psh` 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.

A primeira particularidade da `psh` é que, na linha de comando, o usuário pode solicitar a criação de um ou mais processos:

Exemplo 1 (um processo):

```
psh> comando0
```

Exemplo 2 (três processo):

```
psh> ls -l ; grep batata ; sort parametro1 parametro2
```

Exemplo 3 (cinco processos):

```
psh> comando4 ; comando5 ; comando6 ; comando7 ; comando8
```

Mas ao contrário das shells UNIX tradicionais, quando é passado um ou mais comandos, os processos associados aos respectivos comandos serão criados em *background*. Além disso, haverá uma classificação interna à `psh`:

- processos criados isoladamente (como no Exemplo 1) são os processos "**não vacinados"**, isto é, sem o "passaporte de vacinação" e que, portanto devem sempre estar isolados, cada um em seu *process group*;

- processos criados em conjunto (como no Exemplo 2) são os processos **vacinados** e que apresentam passaporte de vacinação; estes são colocados em um mesmo *process group*, chamado simpaticamente de "grupo dos vacinados".

Assim, no Exemplo 2 acima, a `psh` deverá criar 3 processos – P1 , P2 e P3 – para executar os comandos comando1, comando2 e comando3 respectivamente (*comandoX* trata-se de um "comando externo", como será melhor explicado). Esses três processos deverão pertencer a um mesmo *processo group*, ("grupo dos vacinados"). Supondo que em algum momento o Exemplo 3 seja executado na `psh`, os cinco novos processos serão colocados no mesmo *process group* em que já se encontravam P1 , P2 e P3 (mas calma… mesmo vacinados, todos estarão de máscara!!).

O número de comandos externos passados em uma mesma linha de comando pode variar de 1 a 5… (não dá para criar mais do que isso por vez… dá trabalho para a `psh` verificar o passaporte de cada um :-P)!

Vale lembrar que todos os processos são criados em *background*, com isso:

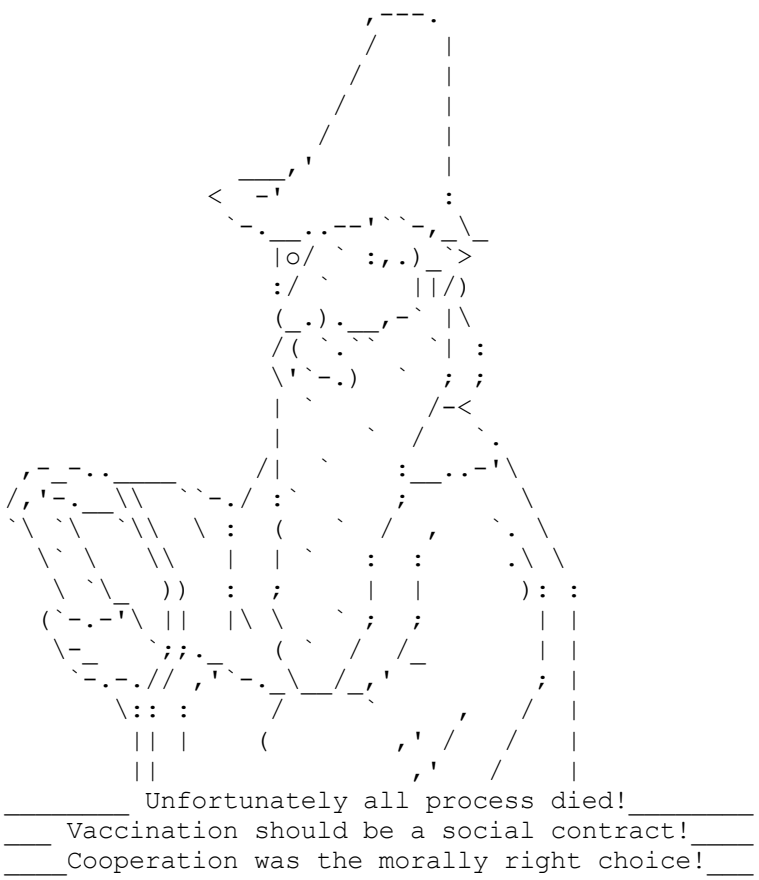
- após o usuário entrar uma linha de comando como no Exemplo 2, a `psh` retorna (exibindo o prompt) imediatamente para receber novos comandos;
- enquanto estão em *background*, os processos não recebem sinais originados no terminal de controle, isto é, nenhum sinal gerado por meio de teclas do terminal, sendo eles:
    - Ctrl+C - SIGINT
    - Ctrl+\ - SIGQUIT
    - Ctrl+Z - SIGTSTP

Mas atenção, os sinais acima representam as cepas conhecidas do vírus. Com isso, os processos do "grupo dos vacinados" na verdade estão todos imunes a esses três sinais (nada deve acontecer se o SO ou outro processo enviar os sinais acima a esses processos). Mas infelizmente o mesmo não pode ser dito sobre os processos não-vacinados… estes, caso recebam um dos três sinais acima, serão finalizados (ou suspenso no caso específico do SIGTSTP). Por fim, a própria `psh`, (que por sua vez estará sempre executando em foreground), quando receber um dos três sinais acima, ela deverá exibir a seguinte mensagem: "Estou vacinada… desista!!".

Mas temos um problema! Apesar de todo o cuidado, como muitos processos ainda estão circulando sem a vacina, uma nova "cepa" do vírus acabou se desenvolvendo e começou a circular. Essa cepa é chamada "SIGUSR1"... e ela é altamente contagiante e letal :-(… Com isso, se qualquer um dos processos (à exceção da própria `psh`) receber um sinal SIGUSR1, esse processo deve finalizar. Além disso, todos os demais processos deverão receber o mesmo sinal SIGUSR1 e se "contaminarão com esse virus" (vacinados e não vacinados), finalizando em seguida. Quanto à própria `psh`, ela é muito cuidadosa, só anda com EPI completo … algo assim:

… e com isso ela não se contaminará como o SIGUSR1!!

Mas, ao “perceber” (isso pode demorar um pouco… não precisa ser imediato!) que há um surto da cepa SIGUSR1, ela deverá exibir a seguinte mensagem, voltando a exibir o prompt em seguida:



Linguagem da psh

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

i. **operações internas da 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 da shell são sequências de caracteres que devem sempre ser executadas pela própria shell e não resultam na criação de novos processos. Na `psh` as operações internas são:

`fg`: faz com que a `psh` coloque todos os processos do grupo de **vacinados** temporariamente em foreground. Eles permanecerão em foreground por 30 seg., retornando para background em seguida. Enquanto o grupo de vacinados está em foreground, a própria `psh` ficará em background esperando (Ela é muito cautelosa mesmo! Não aglomera de jeito nenhum!!)

`term`: termina a operação da `psh`, mas antes disso, ele deve matar todos os processos que foram criados (vacinados e não vacinados).

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).

Exemplo 4:

```
...
psh> comando1 ; comando2 ; comando3
psh> fg
psh>
```

Quanto aos **operadores especiais**, há apenas dois: o símbolo '*;*' e o símbolo '>':

> **O símbolo '*;*'** é utilizado para permitir que diferentes comandos externos possam ser passados mesma linha de comando, resultando na criação de processos no grupo de **vacinados.**
>
> **O símbolo '>'** pode ser utilizado para redirecionar a saída padrão de um processo **não vacinado** para um arquivo, como no exemplo a seguir:
>
> Exemplo 5 :
>
> ```
> psh> comandoX > socialNetowks.txt
>
> //PARA FACILITAR… CONSIDEREM QUE O ARQUIVO JÁ EXISTE!
> ```
>
> … Este operador é bastante utilizado pois muitos processos não vacinados passam a vida jogando informações de forma descontrolada (e sem curadoria nenhuma) em diferentes meios…

Quanto aos **programas a serem executados (comandos externos)**, estes são identificados pelo nome do seu arquivo executável e podem ser seguidos por um número máximo de **dois argumentos** (parâmetros que serão passados ao programa por meio do vetor argv[]). ATENÇÃO! Cada vez que uma linha de comando é processada, e os processos são criados em background, a `psh` deve exibir em seguida o prompt.

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.

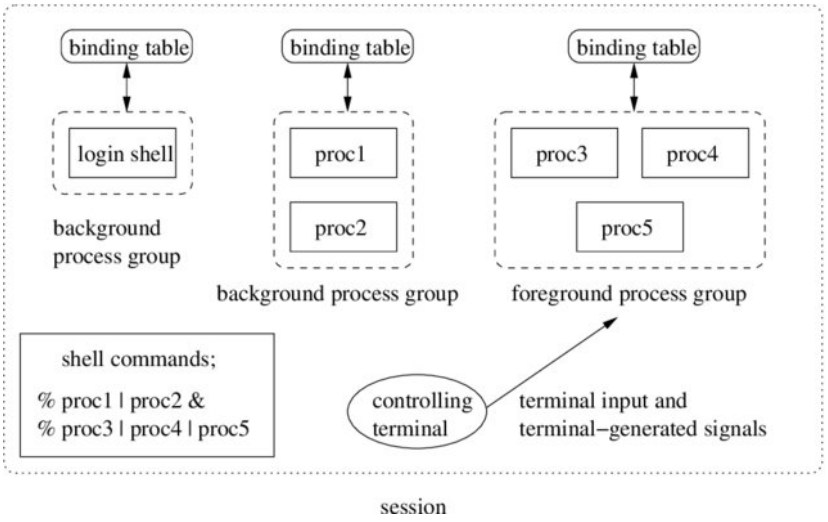


Fig 1: Relação entre processos, grupos, sessões e terminal de controle

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 feita uma chamada a `setpgid()` para alocar um novo *Process Group* para esse processo filho. **Numa shell convencional (como a bash)** 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*.

Mais informações sobre grupos de **foreground e background aqui:**

- https://www.gnu.org/software/libc/manual/html_node/Foreground-and-Background.html
- https://man7.org/linux/man-pages/man3/tcsetpgrp.3.html

Sessões no Linux!

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()` (não será usado neste trabalho), 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.