Sistemas Operacionais

Profª. Roberta Lima Gomes - email: soufes@gmail.com

1º Trabalho de Programação

Período: 2015/1

Data de Entrega: 15/05/2015 *(já corrigida)* ***Composição dos Grupos: até 2 pessoas***

Material a entregar

- Por email: enviar um email para **soufes@gmail.com** seguindo o seguinte formato:
  - Subject do email: "**Trabalho 1**"
  - Corpo do email: lista contendo os nomes completos dos componentes do grupo em ordem alfabética
  - Em anexo: um arquivo compactado com o seguinte nome "**nome-do-grupo.extensão**" (*ex: joao-maria-jose.rar*). Este arquivo deverá conter todos os arquivos (incluindo os *makefile*) criados com o código muito bem comentado.

Serão pontuados: clareza, indentação e comentários no programa.
Desconto por atraso: 1 ponto por dia

Descrição do Trabalho

Vocês devem implementar na linguagem C uma shell denominada `fsh` (*family shell*) para colocar em prática os princípios de manipulação de processos.

Ao iniciar, `fsh` exibe seu *prompt* "fsh>"(os símbolos no começo de cada linha indicando que a *shell* está à espera de comandos). Quando ela recebe uma linha de comando do usuário o processamento da mesma começa. 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 diferencial da `fsh` é que, ao contrário das shells UNIX convencionais, ao processar um comando que seja um programa para ser executado ela deverá criar o processo para executar este programa em *background* (a `fsh` retorna imediatamente para receber novos comandos).

Outro diferencial da `fsh` é que ela permite que em uma mesma linha de comandos o usuário solicite a execução de vários programas em paralelo mas seguindo uma hierarquia da seguinte forma:

```
fsh> comando1 @ comando2 @ comando3 ...
```

Neste exemplo a shell deverá primeiramente, criar um processo "gerente" P0. Então, o processo P0, deverá criar os processos P1 (para executar o `comando1`), P2 (para executar o `comando2`), P3 para executar o `comando3`). Desta forma é criada uma família de processos em que o processo "gerente" P0 é pai de P1, P2, P3,... até o processo referente ao último comando da linha (a lista suporta até 10 comandos). Lembre-se que tando P0, quanto os outros Px devem ser processos de background e, portanto, não devem estar associados a nenhum Terminal.

Outra particularidade da `fsh` é que uma família de processos é muito unida! Primeiramente porque todos os processos criados pelo processo "gerente" devem ignorar o sinal SIGTSTP (eles não "gostam" de ser suspensos!!). Então, sempre que o usuário clicar "Ctrl-z", o processo "gerente" deverar exibir a mensagem: "Não adianta tentar suspender... minha família de processos está protegida!". Mas se o usuário clicar "Ctrl-z" e apenas a `fsh` estiver rodando, nada acontece.

Além disso, quando um dos processos morre, todos os demais processos da família morrem juntos, à exceção da `fsh` (que é highlander). Esse genocídio de processos deve ser implementado com o auxílio de sinais (alterando-se a rotina de tratamento de um determinado sinal).

Finalmente, a nossa fsh não quer saber de morte súbita enquanto ela tiver filhos ainda vivos... (muito responsável!). Com isso ela deve BLOQUEAR (não é ignorar) o sinal gerado pelo Ctrl-C enquanto ela tiver filhos. Quando ela não tiver nenhum filho vivo, ela pode ir descansar em paz caso o usuário faça um Ctrl-C.

Linguagem da `fsh`

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

(i) operações internas da shell,
(ii) nomes de programas que devem ser executados (e seus argumentos),
(iii) operadores especiais

- *Operações internas da shell* são as sequências de caracteres que devem sempre ser executadas pela própria shell e não resultam na criação de um novo processo. Na `fsh` as operações internas são: `cd`, `pwd`, `waitz` e `exit`. Essas operações 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).

`cd`: Muda o diretório corrente da *shell*. Isso terá impacto sobre os arquivos visíveis sem um caminho completo (*path*).

`pwd`: Exibe o diretório corrente visto pelo processo.

`waita`: Faz com que a *shell* libere todos os processos filhos que estejam no estado "Zombie" antes de exibir um novo *prompt*. Cada processo que seja "encontrado" durante um `waitz` deve ser informado através de uma mensagem na linha de comando. Caso não haja mais processos no estado "Zombie", uma mensagem a respeito deve ser exibida e `fsh` deve continuar sua execução.

`exit`: Este comando permite terminar propriamente a operação da *shell*. Ele faz com que todos os seus herdeiros vivos (herdeiros diretos e indiretos) morram também ... e a `fsh` só deve morrer após todos eles terem sido "liberados" do estado "Zombie".

- *Programas a serem executados* são identificados pelo nome do seu arquivo executável e podem ser seguidos por um número máximo de cinco argumentos (parâmetros que serão passados ao programa por meio do vetor `argv[]`). Como dito anteriormente, quando o usuário digita um nome de um programa para ser executado (seguido de parâmetros ou não), a fsh deve criar um processo auxiliar P0, e este, por sua vez, deve criar um processo filho que irá executar o programa (supõe-se que o executável do programa esteja em algum diretório do PATH). Os processos devem ser executados em backgroud e a *shell* deve continuar sua execução. Isso significa retornar imediatamente ao *prompt*.

- Existe apenas um tipo de *operador*. Ele é o símbolo '`@`' ao qual vocês já foram apresentados. Os demais operadores conhecidos de shell convencionais, como os símbolos '`|`' e '`>`', NÃO serão tratados neste trabalho. O operador '`@`' só deverá ser usado juntamente com comandos que correspondam a programas executáveis (nunca será usado com operações internas). Por exemplo:

```
fsh> gedit @ ps -l @ firefox www.google.com
```

Dicas Técnicas

Este trabalho exercita as principais funções relacionadas ao controle de processo, como `fork`, `execvp`, `waitpid`, `chdir`, e Sinais, 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.

Verificação de erros

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).