Sistemas Operacionais

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

1º Trabalho de Programação

Período: 2018/1

Data de Entrega: 13/05/2018 ***Composição dos Grupos: até 3 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 (.zip OU .rar) com o seguinte nome "**nome-do-grupo.extensão**" (*ex: joao-maria-jose.rar*). Este arquivo deverá conter todos os arquivos (**incluindo o *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 deverão implementar uma nova shell na linguagen C, denominada `csh` (*Crazy Shell*). Ao contrário de uma shell convencional, que é um interpretador tradicional de programas responsável por lançar processos forground ou background a cada comando, a `csh` é um pouquinho "fora da casinha"!

Linguagem da `csh`

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

(i) comandos internos da shell,
(ii) nomes de programas que devem ser executados (e seus argumentos),

- ***Comandos internos 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 `csh` os comandos internos são: `wait` 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).

`wait`: Faz com que a *shell* libere todos os processos filhos que estejam no estado "Zombie" antes de exibir um novo *prompt*. Para cada processo zombie "encontrado" durante um `wait`, deve ser impressa uma mensagem na linha de comando. Esta mensagem deve imprimir o PID do processo "Zombie" e a causa da sua "morte" (EXIT ou SIGNAL). Caso não haja mais processos no estado "Zombie", uma mensagem a respeito deve ser exibida e `csh` deve continuar sua execução exibindo o prompt.

`exit`: Este comando permite terminar propriamente a operação da *shell*. No entanto, como a `csh` é "apegada" a seus filhos, se houver algum deles em execução ou suspenso, ela deve ficar bloqueada aguardando esse processo terminar ... a `csh` só deve morrer de fato após todos os seus filhos terem morrido e 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[]`). E aqui está o maior problema da `csh`: ao receber como entrada o nome de um arquivo executável, ela irá criar um processo filho para executá-lo, mas esse filho criará um outro filho (i.e. que será "neto" da `csh`). Tanto o filho quanto o neto, deverão rodar o mesmo arquivo executável passado como entrada na linha de comando. Mas enquanto o filho executará em *foreground*, o neto deverá executar em *background*. Com isso, somente quando o processo filho finaliza (ou suspende), a `csh` deve re-exibir o seu prompt.

Tratamento de Sinais

- Se usuário digitar "Ctrl-c" uma vez, nenhum processo deve morrer! A `csh` deve capturar esse sinal, imprimindo uma mensagem de aviso ao usuário:

```
csh> Não adianta me enviar um sinal por Ctrl-c, não vou morrer! Você quer
     suspender meu filho que está rodando em foreground? S/n:
```

Se nesse instante houver algum filho rodando em *foreground* (executando um programa), e o usuário digitar `ENTER` ou 's', esse processo filho deve ser suspenso. Caso contrário, nada ocorre e a `csh` re-exibe o prompt. Observe que o processo "neto" que foi criado junto com esse filho nem recebe o SIGINT, já que ele está rodando em *background*.

> IMPORTANTE!!! Enquanto o sinal SIGINT estiver sendo tratado, na csh todos os demais sinais devem permancer bloqueados.

Dicas Técnicas

Este trabalho exercita as principais funções relacionadas ao controle de processo, como `fork`, `execvp`, `waitpid`, e tratamento de s*inais* (**`sigaction`**), 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, mas você deve fazer um tratamento mínimo.

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