Sistemas Operacionais

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

1º Trabalho de Programação

Período: 2016/1

Data de Entrega: 29/05/2016 ***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 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 `nsh` (*Nice Shell*). Ao contrário de uma shell convencional, que é um interpretador de programas responsável por lançar processos forground ou background a cada comando, a `nsh` tem um pequeno "diferencial":

- Todos os processos que ela cria, devem ter suas prioridades minimizadas (reduzidas) de forma serem sempre os últimos processos a serem atendidos dentre os processos prontos.

Linguagem da `nsh`

A linguagem compreendida pela `nsh` 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 `nsh` as operações internas são: `cd`, `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*. Cada processo que seja "encontrado" durante um `wait` deve ser informado por meio de uma mensagem na linha de comando. Caso não

haja mais processos no estado “Zombie”, uma mensagem a respeito deve ser exibida e `nsh` deve continuar sua execução.

`burst`: este comando permite que o usuário “turbine” a prioridade de todos os processos filhos da `nsh`. Com isso, quando ele é digitado, a `nsh` deve aumentar ao máximo (o possível) a prioridade de todos os processos filhos.

`exit`: Este comando permite terminar propriamente a operação da *shell*. Ele faz com que todos os seus herdeiros vivos (herdeiros diretos) morram também ... e a `nsh` 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[]`). Os processos devem ser criados e executados em *foreground.* Para criar processos de *background*, basta que o usuário coloque no final da linha de comando o caracter ‘@’. Nesse caso, a nsh deve criar o processo (em background) e retornar a exibir o prompt para o usuário.

Tratamento de Sinais

Se usuário digitar algum comando especial “Ctrl-...” que gere um sinal, a nsh em si deve capturar esse sinal, imprimindo uma mensagem de aviso ao usuário: “Não adianta me enviar o sinal por Ctrl-... . Não vou fazer nada!!”. Mas se tiver um processo de *foreground* rodando, então esse deve receber qualquer sinal gerado via “Ctrl-...” e executar o tratamento default (observem que nesse caso a nsh não deverá receber o sinal!). Já os processos de *background* não receberão nenhum sinal gerado via “Ctrl-...”. Mas, obviamente, se alguém enviar um sinal para um dos processos de *background* via chamada “kill”, esse sinal deverá ser entregue normalmente ao processo.

IMPORTANTE!!!

Como um “presentinho” que a nsh vai dar para seus filhos, para compensar a baixa prioridade que eles terão, todos eles deverão ignorar o sinal SIGTERM.

Abaixo temos um exemplo de sequência de execução da nsh:

```
nsh> ls -l          //comando externo
...
nsh> firefox www.google.com @     //comando externo, filho criado em background
...
nsh> ps             //comando externo
nsh> wait           //comando interno
processo filho zombie 2351 liberado
processo filho zombie 2353 liberado
nsh>                //usuário digita Ctrl-c...
Não adianta me enviar o sinal por Ctrl-c. Não vou fazer nada!!
nsh> burst          // comando interno … todos os filhos (em background) têm
                    // suas prioridades  aumentadas!
nsh> exit           //FIM DA nsh!!
```

Dicas Técnicas

Este trabalho exercita as principais funções relacionadas ao controle de processo, como `fork`, `execvp`, `waitpid`, `nice`,`renice`, 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, 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).