Sistemas Operacionais

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

1º Trabalho de Programação

Período: 2019/2

Data de Entrega: 22/10/2019 (até meia-noite)
Composição dos Grupos: até 3 pessoas

Material a Enviar

- 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.zip**" (*ex: joao-maria-jose.zip*). Este arquivo deverá conter todos os arquivos (incluindo os *makefile*) criados com o código muito bem comentado.
    Valendo ponto: clareza, endentação e comentários no programa.
    Desconto por atraso: 1 ponto por dia

Motivação

Insatisfeitos com todos os tipos de shells que vocês já usaram, vocês decidiram criar a sua própria shell, chamada bgsh (background shell). Essa shell deve tratar: execução de programas (em background e em foreground), comandos internos e tratamento de sinais.

Objetivos

Se familiarizar com chamadas básicas de sistemas, sinais, grupos de foreground/background.

Descrição do Trabalho

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

Ao iniciar, `gsh` exibe seu *prompt* "`gsh>`" (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, é iniciado o processamento desse comando. Primeiramente, 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 `gsh` é que, ao contrário das shells UNIX convencionais, é que na linha de comando o usuário pode solicitar a criação de um conjunto de processos:

```
gsh> comando1 # comando2 # comando3
```

A *shell* poderá receber até 5 comandos na mesma linha. Neste exemplo, o programa deverá criar 3 processos – P1 , P2 e P3 – para executar os comandos `comando1`, `comando2` e `comando3` respectivamente (comando*X* corresponde a um arquivo executável do sistema, tratando-se de um "comando externo" que eventualmente pode receber parâmetros, como "ls -l"). Outra

particularidade dessa *shell* é que quando a gsh recebe apenas um comando como abaixo, ela cria o processo em foreground:

```
gsh> ls -l
```

Mas quando ela recebe vários comandos (separados pelo caractere '#') os novos processos (no exemplo acima, P1, P2 e P3) serão criados como processos de *background*. Mas lembrem-se que todos os processos do conjunto devem ser "irmãos" e pertencer o mesmo *Group*. Observem que apenas comandos passados na mesma linha resultarão em processos irmãos.

Essa shell tem um pequeno problema… de forma aleatória, novos processos criados podem criar filhos "fantasmas" de forma espontânea. No primeiro exemplo, o processo P1 pode (com uma chance de 50%) criar um filho P1' que irá executar o mesmo programa a ser executado por P1 (i.e., comando 1). O mesmo é válido para P2 ou P3. Percebam que como consequência, eventualmente, P1 poderia criar um filho ghost P1' e P2 poderia não criar nenhum filho ghost.

Outra particularidade da `gsh` é que quando um dos "irmãos" morre devido ao um sinal, os demais processos de *background* do mesmo grupo também devem morrer (incluindo os processos *ghosts* que podem ter sido criados). No primeiro exemplo, se P2 terminar porque recebeu um sinal, P1 e P3 (e se existirem P1' e P3') também devem ser finalizados. Mas se um dos processos *ghosts* morrer devido a um sinal, nada acontece com os demais processos do grupo. Por fim, se os processos morrem normalmente, também nada acontece com os demais processos do grupo.

ATENÇÃO: se existirem outros processos de *background* criados em outras linhas de comando, eles NÃO devem morrer).

SOBRE O TRATAMENTO SINAIS...

Nossa gsh não quer saber de morte súbita enquanto ela tiver descendentes ainda vivos... (muito responsável!). Com isso quando o usário digitar Ctrl-C (SIGINT), caso ela ainda tenha descendentes vivos (mas observem que ela NÃO vai considerar nessa conta seus descentes *ghosts*, isto é, aqueles processos que foram criados por um surto sobrenatural!), ela deve imprimir uma mensagem perguntando ao usuário se ele tem certeza que ele deseja finalizar a shell. Caso o usuário confirme, a shell é finalizada. Mas se a shell não tiver nenhum descendente vivo, ela pode ir descansar em paz caso o usuário faça um Ctrl-C.

> IMPORTANTE: durante a execução do tratador do sinal SIGINT, todos os demais sinais deve ser BLOQUEADOS. Dica: pesquise a chamada de sistema sigaction()...

Quanto aos descendentes da shell, TODOS devem IGNORAR o SIGINT… sejam eles processos de *foreground*, *background* ou *ghosts*.

Por fim, caso o usuário digite Ctrl-Z (SIGTSTP), a shell em si não será suspensa, mas ela deverá suspender todos os seus descendentes (incluindo processos de *foreground*, *background* e *ghosts*).

LINGUAGEM DA SHELL

A linguagem compreendida pela `gsh` 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 que devem ser executados,
(iv) argumentos a serem passados para os comandos ou programas.

- *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 `gsh` as operações internas são:

  - `mywait`: faz com que a shell libere todos os seus descendentes (diretos e indiretos) que estejam no estado "Zombie" antes de exibir um novo prompt Aqui vocês podem desconsiderar os processos *ghosts*… afinal o cara já era *ghost* mesmo… que mal tem ele se tornar um Zombie e ficar perambulando por ahi !?
  - `clean&die`: deve terminar a operação da shell, mas antes, esta deve garantir que todos os seus descendentes vivos morram também (background e ghosts!)... Ela só deve morrer após todos eles (herdeiros diretos e indiretos) terem morrido.

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).

- *Operadores especiais:* Existe apenas um tipo de *operador:* o símbulo '#' ao qual vocês já foram apresentados. Os demais operadores conhecidos de shell convencionais, como os símbolos '`|`', '`&`', etc., não serão tratados neste trabalho.

- *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[]`). Cada comando do tipo (i) ou (iii) seguido ou não de argumentos deve terminar com um fim de linha. No caso (i) o comando é executado diretamente pela gsh. No caso (iii) o processo (ou processos) devem ser criados conforme explicado anteriormente. ATENÇÃO! Cada vez que um processo Px criado em foreground retorna, a gsh deve exibir imediatamente o prompt.

Dicas Técnicas

Este trabalho exercita as principais funções relacionadas ao controle de processo, como `fork`, `execvp`, `waitpid`, 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). Esse site a seguir também apresenta algumas dicas sobre como manipular grupos e sessões, processos de foreground e background:

https://www.win.tue.nl/~aeb/linux/lk/lk-10.html

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).