Sistemas Operacionais

1º Trabalho de Programação

Período: 2020/Especial

Data de Entrega: 30/11/2020 ***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 um shell denominado `acsh` (*anti-crowd shell*) para colocar em prática os princípios de manipulação de processos.

Ao iniciar, `acsh` 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.

Um primeiro diferencial do `acsh` é que, na linha de comando, o usuário pode solicitar a criação de um ou mais processos:

```
acsh> comando1 <3 comando2 <3 comando3
```

Mas ao contrário dos shells UNIX tradicionais, ao processar um comando, ou um conjunto de comandos, ele deverá criar o(s) processo(s) para executar o(s) respectivo(s) comando(s) em uma sessão separada da sessão do `acsh`. Isso é justamente para evitarmos aglomerações de processos… apenas pequenos grupos dentro de cada sessão! Assim, no exemplo dado acima, o `acsh` deverá criar 3 processos – P1 , P2 e P3 – para executar os comandos comando1, comando2 e comando 3 respectivamente (comando X trata-se de um "comando externo", como será melhor explicado). E esses três processos deverão ser "irmãos" e pertencerem a uma mesma sessão, que seja diferente da sessão do `acsh`. O número de comandos externos passados em uma mesma linha de comando pode variar de 1 a 5… lembrando… sem aglomerações! Com isso em uma mesma sessão só deverá haver processos que tenho sido criados durante uma mesma linha de comando.

Além disso, os processos criados serão executados em *background* e não estarão associados a nenhum terminal de controle, consequentemente:

- os processos criados pelo `acsh` não devem receber nenhum sinal gerado por meio de teclas do terminal (ex: Ctrl-c);
- o `acsh` retorna imediatamente para receber novos comandos.

Outra particularidade do `acsh` é que o sinal SIGUSR1 é um sinal muito perigoso e contagioso! Se um dos processos de *background* morre devido ao SIGUSR1, os demais processos "irmãos" que se encontram na mesma sessão (ou seja, que foram criados na mesma linha de comando) devem morrer de forma coletiva, devido ao mesmo sinal. No exemplo anterior, se, por exemplo, P2 terminar devido a um sinal SIGUSR1, P1 e P3 também devem ser finalizados por meio do mesmo sinal. Observem que se existirem outros processos de *background* que tenham sido criados em outras linhas de comando, eles NÃO deverão morrer, uma vez que eles estavam isolados em sessões diferentes.

Ah sim! Tem mais… se um processo de *background* for criado isoladamente, como neste exemplo:

```
acsh> comando1
```

... ele não poderá morrer devido ao sinal SIGUSR1… nunca! Afinal, ele está isolado e bem comportado em uma sessão exclusiva para ele… :-) … isolamento social tem que ter alguma vantagem!!

Bom… apenas em algumas situações não serão criados processos em *background*. A primeira delas é quando o usuário entrar com uma operação interna do shell. Com isso não é criado nenhum novo processo e o próprio shell executará a operação (detalhes mais à frente). A segunda situação ocorre quando for utilizado um operador especial '%' no final da linha. Por exemplo:

```
acsh> comando1 %
```

Neste exemplo acima, o `acsh` deverá criar um processo em *foreground* e pertencendo à mesma sessão do `acsh`. Neste caso, o *prompt* só será novamente exibido ao final da execução do processo de *foreground* criado.

Linguagem do shell

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

i. operações internas do 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 do shell são sequências de caracteres que devem sempre ser executadas pelo próprio shell e não resultam na criação de novos processos. No `acsh` as operações internas são:

`cd`: Muda o diretório corrente do shell. Recebe um argumento contendo o nome do diretório.

`exit`: termina a operação do shell, mas antes disso, ele deve finalizar todos os processos de *background* que ainda estejam rodando.

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).

Quanto aos **operadores especiais**, há apenas dois. O primeiro deles é o símbolo '`<3`' utilizado para se entrar com diferentes comandos na mesma linha de comando. O segundo operador é o símbolo '%' que indica que o comando ao qual ele sucede deve ser executado em *foreground*.

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[]`). Em uma mesma linha de comando, pode haver um conjunto máximo de 5 programas (com seus argumentos) separados pelo operador '<3'

Com isso cada linha de comando pode conter:

- uma operação interna do shell
- um comando externo (nome de um programa executável e seus argumentos), terminado ou não com o símbolo '%'
- dois a cinco comandos externos (nomes de programas executáveis e seus argumentos) separados pelo símbolo '<3' (nesse caso não pode haver o símbolo '%' no final).

Tratamento de Sinais

Se O usuário digitar algum comando especial "Ctrl-..." que gere um sinal, isto é, Ctrl-C (SIGINT), Ctrl-\ (SIGQUIT), Ctrl-Z (SIGTSTP):

- O `acsh` em si deve **capturar** esse sinal, imprimindo uma mensagem de aviso ao usuário: "Não adianta me enviar o sinal por Ctrl-... . Estou vacinado!!". Atenção! Durante a execução do tratador desses sinais, esses três sinais deverão estar bloqueados (medida de precaução contra usuários histéricos!).
- Mas se houver um processo de *foreground* rodando, então este deve receber qualquer sinal gerado via "Ctrl-..." e executar o tratamento default. Mas e o `acsh` ? Enquanto houver um processo em foreground rodando o `acsh` deverá ignorar esses sinais gerados via teclado!.

Quanto aos processos de *background*, estes não receberão nenhum sinal gerado via "Ctrl-...", como é esperado. Mas, obviamente, se alguém enviar um sinal para um dos processos de *background* via chamada "kill", esse sinal deverá ser entregue normalmente ao processo, o qual deverá executar o tratamento default (lembrando daquela pequena exceção do SIGUSR1 ... no caso de um processo em *background* que tenha sido criado isoladamente, ele se encontra protegido em sua respectiva sessão, então ele não morre devido ao SIGUSR1 …).

Dicas técnicas

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

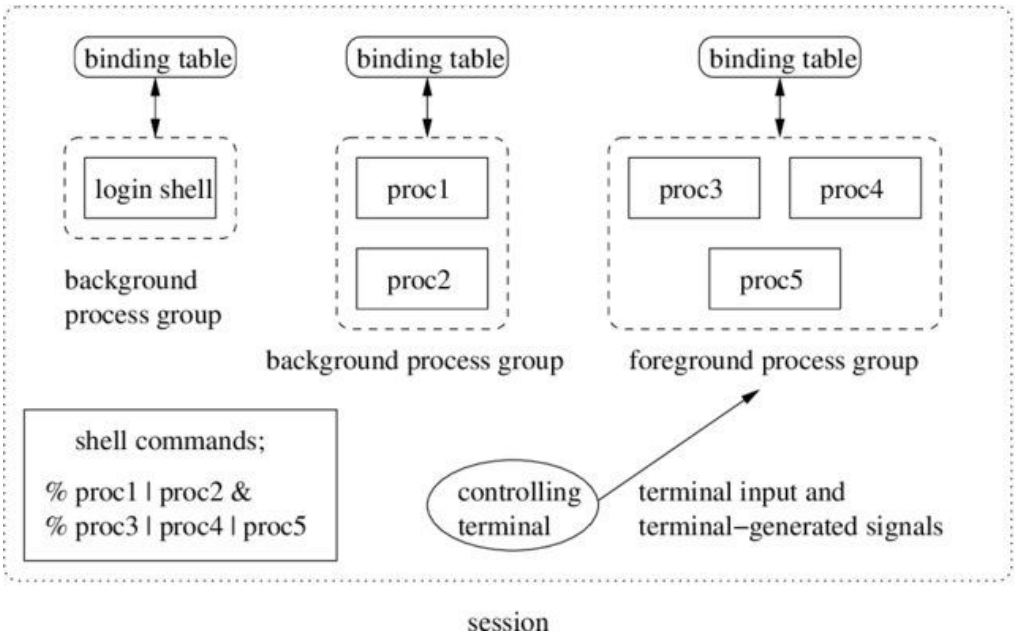

Fig 1: Relação entre processos, grupos, sessões e terminal de controle

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 feito uma chamada a `setpgid()` para alocar um novo *Process Group* para esse processo filho. 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*.

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()`, 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.