Sistemas Operacionais
Trabalho Prático de Programação – 2017/1

Prof: José Gonçalves

Data de entrega: 25/07/2017 ***Composição dos Grupos: 3 alunos*** ***Linguagem: C***

Material a entregar (por email)

- Relatório comentando a resolução de cada questão (formato: PDF).
- Arquivos .c, executáveis, .txt e makefiles.
- Destinatário: zegonc@inf.ufes.br
- Subject: Trabalho1_SO_ componente1_componente2_componente_3

OBS: Após a entrega, cada grupo fará uma entrevista sobre o trabalho desenvolvido em data a ser marcada.

Parte I. Acesso a uma Lista Encadeada

Três tipos de processos compartilham acesso a uma lista encadeada: processos de busca, inserção e de eliminação. Processos de busca meramente examinam a lista; assim podem executar concorrentemente entre si. Processos de inserção incluem itens no final da lista; inserções devem ser mutuamente exclusivas. No entanto, uma inserção pode ocorrer em paralelo com qualquer número de buscas. Finalmente, eliminações podem ocorrer em qualquer ponto da lista. No máximo um processo de eliminação pode ter acesso à lista em qualquer instante, e a eliminação deve ser mutuamente exclusiva com buscas e inserções. Este problema é uma extensão do problema dos leitores e escritores. Analise a sua solução em relação à possibilidade de starvation para cada um dos tipos de processos.

Você deve fazer este trabalho em C, usando o padrão *pthreads* (POSIX Threads) para Linux 2.x. Você usará funções básicas de gerenciamento de *threads* (*pthread_create*,...), *mutexes* (*pthread_mutex_*...) e variáveis de condição (*pthread_cond_*...) Não é permitido o uso de outras bibliotecas de suporte à programação *multithread* (bibliotecas de semáforos, por exemplo).

Parte II. O Problema dos Macacos

Suponha que haja macacos em ambas as margens de um rio e, de tempos em tempos, os macacos decidem passar para o outro lado à procura de comida. A passagem para a outra margem do rio é feita através de uma ponte de corda. Mais de um macaco pode atravessar a ponte ao mesmo tempo, mas isso só é possível se eles estiverem indo na mesma direção.

Implemente um programa que faça o controle da passagem de macacos pela ponte usando Semáforos. Para testar o sistema, crie 10 threads que representem os macacos, colocando inicialmente metade deles em cada margem do rio. Sempre que um macaco iniciar ou concluir a travessia, imprima uma mensagem na tela identificando o macaco.

Após testar o programa acima, crie agora uma nova verão do programa adicionando dois gorilas, um de cada lado do rio. Como os gorilas são muito pesados, eles só poderão atravessar a ponte sozinhos. Como os outros macacos têm medo dos gorilas, eles terão prioridade para fazer a travessia.

Parte III. Escrevendo um Programa Shell

Suponha que você trabalhe para uma empresa que está desenvolvendo o Super Ultra Sistema Operacional (SUSO). Um dos objetivos do SUSO é fornecer diferentes *shells* adaptados para diferentes usuários. Você está encarregado de desenvolver o *simple shell* (*spsh*), que será programado dentro de "*spsh*.c".

O chefão do SUSO quer que o seu *spsh* seja implementado da seguinte forma:

- Use a variante "*execvp*" da SVC exec.
- O *prompt* do spsh deve ser "*spsh*." concatenado com o path do diretório corrente (use getenv("PWD")). Por exemplo: "spsh./home/jonh/so/>"

Como o chefão do SUSO não estava de bom humor, ele solicitou que o *spsh* fosse capaz de tratar "pipes" (ex: ps –aux | grep john) e redirecionamento de entrada/saída (ex: wc -l < infile). Ele exigiu ainda que o *shell* fosse capaz de executar comandos em *background* (onde o último parâmetro deve ser '&').

Parte IV. Sincronizando threads com semáforos POSIX

Podemos encontrar dois tipos de semáforos em ambientes *Unix- like*: System V and POSIX. Em geral, sistemas mais antigos usam a versão System V e sistemas Li*nux- based* usam a versão POSIX, sendo a curva de aprendizado deste último bem menor.

Nesta parte do trabalho, para esquentar, você deverá resolver os exercícios que são propostos no site abaixo. São três exercícios, que exploram o uso de semáforos POSIX na sincronização de *threads* no sistema operacional Linux. Resolva a sequência de exercícios conforme apresentado no site, com especial atenção para o exercício 3 (problema do produtor-consumidor).

Site: http://www.csc.villanova.edu/~mdamian/threads/posixsem.html.

Após isso, resolva o problema abaixo:

O Problema...

Dizem por aí que durante o último *Rock In Rio*, muitos jovens da UFES tentaram comprar o ingresso no local do show com cambistas, mas que esses ingressos acabaram logo. Então, os jovens acabaram descobrindo que havia um "esquema" muito ilícito para entrar no evento ... em uma comunidade vizinha ao local do show. Lá havia um barqueiro com um pequeno barco no qual cabiam 3 pessoas (além do barqueiro). Entre os fundos do local do show e a comunidade havia um lago (muito sujo...), e o barqueiro se propôs a fazer a travessia do lago pela módica quantia de R$100,00/pessoa. À medida que os jovens iam chegando, eles entravam no barco até que o mesmo ficasse cheio. O barqueiro então recolhia a ajuda de custo, dizendo que era para a comunidade, e saía com o barco para a travessia.

Chegando na outra margem, o barqueiro liberava a saída do barco. No entanto, visto que havia um matagal com uma grande cerca nos fundos do local, o barqueiro esperava que todos tivessem conseguido atravessar o mato e a cerca com sucesso, para então voltar com o barco e fazer uma nova travessia.

Claro que essa história (sim, com h mesmo! Pessoas do DI/UFES juram que aconteceu ...) não teve um final feliz, já que em um dado momento um dos jovens que não era tão jovem assim, ao atravessar o matagal, ficou "agarrado" na cerca... o que acionou a segurança do local e acabou com o esquema do barqueiro!

A Tarefa ...

Vocês terão que modelar essa situação através de "processos/*threads*", escrevendo um programa concorrente na linguagem C, no Linux. Vocês deverão criar dois tipos de processos/*threads*, um representando os jovens e o outro representando o barqueiro. Considere a existência de 15 (quinze) jovens e 1 (um) barqueiro e que a sincronização das ações dos jovens e barqueiro será feita através de *semáforos POSIX.*

O jovem quando chega na comunidade, tenta entrar no barco se houver vaga; se não houver, ele espera em uma fila até que o barco volte. Uma vez no barco, ele espera o barqueiro fazer a travessia e chegar ao outro lado da margem. Do outro lado, ele tenta então atravessar o matagal. Se ele conseguir, "termina" dentro do show. Se ele não conseguir atravessar, ele "termina" preso pelos seguranças e, neste momento, todos os outros processos deverão ser finalizados. Para tanto, o processo jovem terá um atributo para indicar se o mesmo é capaz ou não de atravessar o matagal. Atenção: durante a criação dos 15 jovens, 1 (um) deles, escolhido de forma aleatória, NÃO deve ser capaz de atravessar o matagal.

Algumas regras básicas:

1. A travessia do lago deve durar 3s de ida, e 2s na volta. A travessia do matagal deve durar 1s para cada jovem.

2. Cada atividade de um processo deverá ser reportada através de uma mensagem. Alguns exemplos de mensagens são:

   Jovem 01 vai tentar pegar o barco...
   Jovem 01 entrou no barco e espera que este saia.
   Jovem 01 sai com o barco.
   Jovem 03 vai tentar pegar o barco...
   Jovem 03 entrou na fila, tem 1 jovem na sua frente.
   Jovem 03 saiu da fila e entrou no barco...
   Barqueiro recolhe ajuda para a comunidade e sai com o barco...
   Barqueiro chega na outra margem e libera os 3 passageiros...
   Barqueiro espera todos atravessarem o matagal...
   Tudo certo! Barqueiro retorna para fazer outra viagem...
   Jovem 05 atravessando matagal...
   Jovem 05 entrou no show...
   Jovem 06 preso na cerca... Shii! Os seguranças me pegaram!
   Opa! Problemas! Os seguranças pegaram um! Acabou meu esquema...
   Opa! Os seguranças pegaram um ... não vou poder entrar no show... o jeito vai ser o telão mesmo!
   ... E para fechar: antes de "morrer" o barqueiro deverá criar um *Named Pipe* no qual o mesmo deixará uma mensagem para a prosperidade: "*O caminho mais curto nem sempre é o melhor ...*"

Parte V. IPC

Elaborar um programa onde o processo pai cria uma área de memória compartilhada (*shmem*) e 2 processos filhos (*fork*) e, em seguida, aguarda o recebimento de uma mensagem de cada um dos filhos (*msgrcv*). A área de memória compartilhada deve ser suficiente para conter duas variáveis do tipo *integer*. A primeira das variáveis será iniciada com o valor 300 (trezentos) e outra com 0 (zero).

Um dos processos filho, entra em *loop* e executa 100 (cem) iterações com a seguinte rotina: (a) lê o valor da 1a variável na área de memória compartilhada; (b) exibe no terminal padrão uma *string* contendo o seu PID, um número de série sequencial iniciado em 1, e outras informações que o programador achar relevante; (c) decrementa o valor lido da variável; (d) "dorme" um tempo aleatório (variando entre 20 e 300 ms); (e) armazena o valor decrementado na 1a variável da memória compartilhada; (f) incrementa o valor da 2a variável. Observe que esta 2a variável não é copiada para a memória local do processo filho/*threads*. O outro processo filho, ao invés de entrar em loop, cria duas *threads* filhas. Cada uma entra em *loop* executando a mesma rotina do primeiro processo filho.

A contagem (número de série sequencial) do processo filho e das *threads* (o loop com *printf´s*) deve ser de tamanho suficiente para permitir a observação da execução do programa. Se 300, 100/100/100 não for suficiente aumente este numero proporcionalmente.

Quando a contagem terminar, cada filho/*thread* deve avisar ao processo/*thread* "pai" que já terminou. O pai deve avisar que está ciente de que o filho/*thread* X terminou. No caso dos processos criados através de *fork()* vamos utilizar a troca de mensagens como mecanismo do Unix de comunicação entre processos filhos com o pai. Ou seja, os processos filhos vão enviar uma mensagem para o processo pai (*msgsnd*) - o processo pai já deve estar aguardando a mensagem dos processos filhos.

No caso das *threads*, estas vão avisar o seu término de forma simples. Observe que o processo que cria as *threads* tem que ficar aguardando as mesmas terminarem e o processo "pai de todos" tem que ficar "bloqueado" aguardando o recebimento de uma mensagem de cada filho (ou seja, *msgrcv* deve ser usada de forma bloqueante).

Quando todos os processos filhos terminarem o processo "pai de todos" identifica isso com uma mensagem na tela, exibindo o valor das duas variáveis da memória compartilhada e também termina. A memória compartilhada precisa ser liberada apos o seu uso.

Além das mensagens do *loop*, apresentar mensagens relevantes do que está acontecendo com cada processo (pai e filhos) no terminal padrão e *logar* estas mensagens em um arquivo (por exemplo, o número da contagem, o PID do processo, etc.).

O procedimento executado pelo 1o filho deverá ser programado em um arquivo separado do arquivo principal e link-editado posteriormente. Os filhos devem ser programados de forma que se imponha o intercalamento aleatório dos mesmos (caso contrário eles podem executar em sequencia). Ou seja, o "dormir" de cada processo/thread que ficam em loop deve ser aleatório (usar o *ramdom* e *randomize* corretamente).

Cronometrar o tempo gasto na execução do *fork()* no processo pai e em cada processo filho, e o tempo de criação das *threads* filhas (*logar* estas medidas no arquivo também).

Elaborar um arquivo makefile para compilar e link-editar os arquivos e gerar o arquivo executável. O programa, fontes, Makefile, etc. devem estar em um diretório separado dos outros arquivos do grupo.

O código deve ser acompanhado de uma breve descrição da implementação (um README) se a execução do sistema não for óbvia e/ou os arquivos fontes não estiverem organizados.

Bibliografia

[1] Kay A. Robbins, Steven Robbins, *UNIX Systems Programming: Communication, Concurrency and Threads, 2nd Edition.*

[2] W. Richard Stevens, *Advanced Programming in the UNIX Environment, 1st Edition*.