Sistemas Operacionais

Trabalho de Programação #1

Período: 2024/2

DESCRIÇÃO DO TRABALHO

PARTE 1: A Barbearia de Hilzer

Nesta primeira parte do trabalho você exercitará a programação *multithreading* explorando a biblioteca ***pthreads*** (POSIX Threads) para Linux 2.x. Você deverá usar as funções básicas de gerenciamento de ***threads*** (*pthread_create*,...), ***mutexes*** (*pthread_mutex_lock, …*) e **variáveis de condição** (*pthread_cond_wait,* ...). Não é permitido o uso de outras bibliotecas de suporte à programação *multithread*, como bibliotecas de semáforos, por exemplo.

O problema …

Numa barbearia existem três cadeiras, três barbeiros e um local de espera que pode acomodar quatro pessoas num sofá e que tem uma área em que os clientes podem estar à espera em pé. O número máximo de clientes que pode estar na sala é de 20. Um cliente não pode entrar na loja se esta estiver totalmente cheia com clientes à espera. Se o cliente puder entrar, senta-se no sofá ou aguarda em pé se não há lugar no sofá. Quando um barbeiro está livre o cliente que está à espera há mais tempo no sofá é servido e, se existirem clientes à espera em pé, aquele que está em pé há mais tempo toma o lugar no sofá. Quando o corte de cabelo de um cliente termina, qualquer barbeiro pode aceitar pagamento, mas porque existe apenas uma caixa registadora, o pagamento é aceito para um cliente de cada vez. Os barbeiros dividem o seu tempo entre cortar cabelo, aceitar pagamento e dormir na cadeira à espera que um cliente chegue. Os clientes invocam as funções: *EntrarNaLoja()*, *SentarNoSofa()*, *SentarNaCadeira()*, *Pagar()* e *SairDaLoja()*. Os barbeiros invocam as funções *CortarCabelo()* e *AceitarPagamento()*.

Existem, assim, as seguintes restrições:

- Os clientes não podem invocar *EntrarNaLoja()* se a loja está cheia.
- Se o sofá está cheio, um cliente que entrou na loja não pode invocar *SentarNoSofa()* até que um dos clientes do sofá invoque *SentarNaCadeira()*.
- Se as três cadeiras estão ocupadas, um cliente não pode invocar *SentarNaCadeira()* até que um dos clientes a cortar cabelo invoque *Pagar()*.
- O cliente tem que chamar *Pagar()* antes do barbeiro poder *AceitarPagamento().*
- O barbeiro tem que *AceitarPagamento()* antes que o cliente chame *SairDaLoja().*

Escrever o código do Cliente e do Barbeiro que assegurem este comportamento. Faça testes que mostrem o comportamento adequado do programa.

PARTE 2: Controle de Pedidos de Medicamentos

Nesta segunda parte do trabalho você vai explorar o uso de **semáforos POSIX**. Como visto em sala de aula, 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 L*inux-based* usam a versão POSIX, sendo a curva de aprendizado deste último bem menor. Nesta parte do trabalho você deverá resolver o problema abaixo empregando as primitivas de semáforos POSIX.

O problema …

Neste trabalho, foi contratado pelo Superintendente do Hospital da UFES (HUCAM - Hospital Universitário Cassiano de Morais) para implementar o sistema controle de pedidos de medicamentos do hospital. Para tal, foi decidido utilizar um sistema produtor/consumidor usando *threads*.

Existem N *threads* produtoras a executar, representando cada uma delas um terminal de onde podem ser dadas ordens para a farmácia do hospital. Estas ordens representam pedidos de preparação de medicamentos para um certo doente. Cada ordem é constituída por *{Nome_Paciente, Id_Medicamento, Quantidade}*. Neste trabalho, as ordens deverão estar especificadas em quatro arquivos diferentes, a fim de ser simples simular o sistema. Ou seja, N=4.

Sempre que existe uma ordem de uma das *threads* produtoras, esta é passada à *thread* consumidora (farmácia), que trata de atendê-la. Na prática, o consumidor limita-se a mostrá-la no terminal. A passagem é feita via um conjunto de variáveis globais em memória. No entanto, uma vez que ter apenas um *slot* de memória faz com que todas as *threads* bloquem à espera que este esteja livre, foi decidido que existiriam N *slots* disponíveis, que são ocupados à medida que é necessário. Ou seja, existe uma tabela em memória que vai sendo ocupada. Isso permite aumentar o paralelismo do programa, e a sua velocidade. A Figura 1 abaixo ilustra o princípio de funcionamento do sistema.

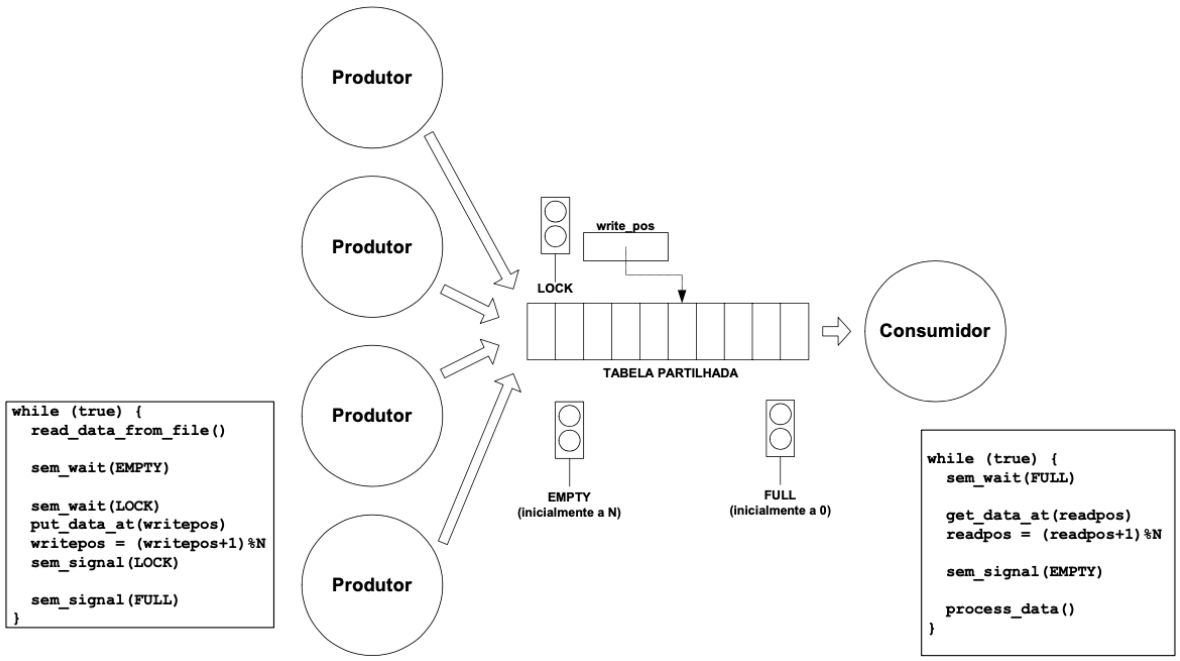

Figura 1 – N Produtores / 1 Consumidor

Existem três semáforos: o semáforo EMPTY, que é inicializado com o número de posições livres (N) e permite aos produtores esperar que existam slots disponíveis; o semáforo FULL, que indica ao consumidor quantos slots ele pode ler neste momento (isto é, quantos estão ocupados, sendo inicialmente igual a 0), e o semáforo LOCK, que permite aos produtores terem atomicidade na obtenção de um slot para escrita (variável *writepos*).

Cada produtor está continuamente executando o seguinte ciclo: Lê uma nova ordem do arquivo (o que no sistema real seria esperar uma ordem do seu terminal). Espera que exista uma posição livre na memória (*sem_wait() no EMPTY*). Ao obter o semáforo, isso quer dizer que existe uma posição livre que pode usar. Assim, irá ver qual é, coloca lá a informação e atualiza a variável que mantém qual a próxima posição livre. Para isso, faz um *sem_wait()* sobre o semáforo LOCK, coloca a informação no *slot* indicado pela *writepos* e atualiza esta variável. É de notar que o buffer é implementado como um buffer circular, o que quer dizer que ao chegar a N, a variável terá de voltar a 0. Finalmente, libera o semáforo LOCK (*sem_signal()* no LOCK), e avisa ao consumidor que existe mais um item pronto a ser tratado (*sem_signal()* no FULL). Quando não existem mais dados no arquivo, escreve uma ordem virtual com uma *flag* que indica que irá terminar.

As *threads* produtoras estão continuamente executando o seguinte ciclo. Espera que exista informação a processar (*sem_wait()* no FULL) e, quando existe, lê essa informação. Como apenas existe um consumidor, a posição a ler é sempre bem conhecida, pois as escritas na memória são feitas em ordem. Para isso, o consumidor utiliza uma variável privada *readpos*. Após ter lido a informação e atualizado a variável de controle de leitura, o consumidor notifica os produtores que já podem utilizar o *slot* acabado de ler (*sem_signal()* em EMPTY).

Finalmente a *thread* consumidora mostra a informação no terminal. Quando souber que todos os produtores já terminaram, deverá também terminar, indicando o número de pedidos que processou.

Tenha atenção aos valores com que inicializa os semáforos. Se não os inicializar corretamente terá um *deadlock* ou então o seu programa não sincronizará de forma correta.

Utilize uma Makefile para automatizar todo o processo de compilação. Apresente testes que comprovem o funcionamento correto do programa.