Instituto Federal de Educação, Ciência e Tecnologia Catarinense - Campus Videira
Bacharelado em Ciência da Computação
Sistemas Operacionais
Aluno: Henrique Pieri de Lima

Relatório Simulação de Crescimento Populacional Exponencial de Colônias de Bactérias com Threads e Prevenção de Impasse

Para simular o crescimento populacional de colônias de bactérias, foi criado um programa em C que utiliza threads (simulando cada colônia) e mutexes (simulando os recursos necessários para as colônias crescerem). Cada colônia de bactérias necessita de recursos compartilhados (nutrientes e espaço) para crescer, os mesmos são escassos e precisam ser alocados simultaneamente para que a colônia complete seu ciclo de crescimento, e sem os dois recursos, a colônia fica bloqueada.

As colônias seguem a fórmula de crescimento exponencial: $P(t) = P_0 \cdot e^{rt}$

Onde: $P(t)$ é a população no tempo t

P_0 é a população inicial

r é a taxa de crescimento

t é o tempo

e é a base dos logaritmos naturais

No programa, são criados dois arrays de **mutexes** (um para cada recurso: nutrientes e espaço) através do tipo *pthread_mutex_t* da biblioteca *pthread.h*, que permite a manipulação de threads e mutexes em C. Também é criado uma estrutura (registro) para conter os dados de cada colônia: ID, população inicial, taxa de crescimento e tempo de simulação.

Na função *main*, é criado um array de **threads** representando as colônias (determinando a quantidade de colônias por uma constante) através do tipo *pthread_t*, também da biblioteca *pthread.h*. Também é criado um array de registros, para armazenar os argumentos (dados) das colônias.

Então, os mutexes de cada tipo de recurso são inicializados através da função *pthread_mutex_init* dentro de uma estrutura de repetição *for* que percorre todos os índices do array. Através de outra estrutura de repetição *for*, são inseridos os dados de cada colônia em seu registro (pode ser feito estático no código para facilitar os testes ou inserido pelo usuário) e depois, por meio de outro *for* (para não começar a execução antes do usuário inserir todos os dados, se for o caso), são criadas suas respectivas threads através da função *pthread_create*, passando por parâmetro o endereço de memória do índice do array que identifica a thread, a função *executarColônia* para ser executada pela thread e os parâmetros desta função, no caso o registro da colônia.

A função *executarColonia* utiliza outra função chamada *adquirirRecurso* para alocar os recursos para as colônias. Esta função *adquirirRecurso* possui uma estrutura de repetição *while* que será executada enquanto não conseguir o recurso, dentro dela há uma estrutura de repetição *for* que percorrerá todos os índices daquele recurso. Como pode haver mais que um recurso do mesmo tipo, precisa percorrer todos os índices do array de mutex do recurso para tentar alocar algum. Então, através da função *pthread_mutex_trylock* (que tenta travar o mutex, mas caso ele já esteja travado ela não espera ele ficar disponível), tenta obter o recurso e, caso consiga, informa o usuário e retorna o índice do recurso alocado. Como o *for* fica executando continuamente até conseguir obter um recurso daquele tipo, o programa pode apresentar um alta taxa de ocupação da CPU (inclusive quando ocorre um impasse), pois continuará percorrendo todos os índices do recurso para cada thread, esperando algum ficar disponível.

Voltando à função *executarColonia*, para **simular o impasse** é feito com que as colônias tentem adquirir os recursos em ordens diferentes, sendo utilizada uma abordagem através do ID da colônia: se for par, tenta obter primeiro o recurso de nutrientes e depois o de espaço; se for ímpar, tenta obter primeiro o recurso de espaço e depois o de nutrientes. Ela informa que está tentando obter o recurso e utiliza a função *adquirirRecurso* citada anteriormente, armazenando o índice do array de cada recurso alocado (usado posteriormente para liberar os recursos). Após alocar os recursos, uma estrutura de repetição *for* percorre o tempo de simulação (passando por cada unidade de tempo), calculando o crescimento da colônia através da função *calcularCrescimento*, que utiliza a fórmula de crescimento exponencial determinada (usando a biblioteca *math.h*). Após simular o crescimento, os recursos alocados são liberados, destravando os mutexes com a função *pthread_mutex_unlock* e, por fim, terminando a execução da thread com a função *pthread_exit*.

Voltando a *main*, uma estrutura de repetição *for* passa por todas as colônias e utiliza a função *pthread_join* para fazer com que espere cada thread terminar para continuar a execução do programa. Após as threads finalizarem, duas estruturas de repetição *for* percorrem todos os índices do array de mutexes dos recursos e os destrói utilizando a função *pthread_mutex_destroy*.

Esta é a parte principal do programa, mas ao mesmo tempo temos a execução do **monitor de impasses** (deadlocks), que é feito monitorando se nenhuma colônia cresceu por algum tempo, o que indica que quase certamente há um impasse. Para isso, é criado um array para monitorar a quantidade de recursos adquiridos por cada colônia, dentro da *main* define-se uma thread para monitoramento de impasses e cria ela após a criação das threads das colônias, executando a função *monitorarImpasse.* Esta função executa um loop infinito (para fazer o monitoramento durante toda a execução do programa), onde é feito uma pausa para permitir que as colônias tentem obter recursos, criada uma variável para identificar se nenhuma colônia cresceu por um tempo e executada uma estrutura de repetição *for*, que percorre todas as colônias e verifica se a colônia adquiriu ambos os recursos, caso sim, marca na variável citada que houve crescimento. Então aguarda mais um tempo e executa o mesmo *for*, garantindo que o impasse identificado é verdadeiro. Por fim, se nenhuma colônia cresceu por um tempo, verificando através da variável utilizada, avisa ao usuário que há um impasse detectado.

Uma das abordagens para **solucionar o problema dos impasses**, que pode ser considerada simples, é fazer com que todas as threads tentem obter os recursos na mesma ordem. Ou seja, para que não haja impasses basta colocar uma ordem estática na aquisição de recursos dentro da função *executarColônia*, assim todas as threads tentarão alocar recursos na mesma ordem.

Para executar as duas versões do programa, com e sem impasses, pode ser manipulado o número de colônias e de recursos, demonstrando que na versão COM impasses, eles sempre ocorrerão (exceto quando o número de recursos for maior que o número de colônias, pela lógica): caso tenha um número alto de recursos comparado ao número de colônias (mas ainda menor que a quantidade de colônias), algumas colônias crescerão, mas logo ocorrerá um impasse e nenhuma mais crescerá; caso tenha um número baixo de recursos comparado ao número de colônias, provavelmente nenhuma colônia crescerá e já ocorrerá um impasse, considerando que tem muito mais threads competindo pelo mesmo recurso, sendo difícil uma mesma thread conseguir alocar ambos recursos necessários. No caso da versão SEM impasses, pode ter qualquer quantidade de colônias e recursos que nunca acontecerão, por mais que algumas colônias demorem, em algum momento serão executadas e terão seu crescimento simulado.