INSTITUTO FEDERAL CATARINENSE - CAMPUS VIDEIRA

Professora: Angelita Rettore de Araujo Zanella

Disciplina: Sistemas Operacionais

Aluno: Felipe Biava Favarin

Turma: 4° Fase de Ciência da Computação

Videira, 16/10/2024.

Relatório sobre a Atividade Colônia de Bactérias

Introdução

A atividade foi oferecida pela professora Angelita Rettore de Araujo Zanella da disciplina de Sistemas Operacionais para a Quarta fase do curso de Ciência da Computação, este é ofertado pelo Instituto Federal Catarinense Campus Videira.

Sobre a atividade

A atividade consiste no desenvolvimento de um código em linguagem C com o objetivo de simular o crescimento populacional exponencial de certas colônias de bactérias que serão representadas por threads. Primeiramente abordaremos o acontecimento de impasse (deadlock) que o código gera e posteriormente a solução implementada.

Desenvolvimento

Da definição de Impasse
“Grupo de tarefas bloqueadas aguardando umas pelas outras.”

Primeiramente falaremos das condições para se acontecer um impasse, que são:

Exclusão Mútua: visa impedir o entrelaçamento de seções críticas, usando mutexes ou semelhantes.

Posse e Espera: a tarefa tem um recurso e quer acessar outro.

Não-preempção: a tarefa só libera os recursos quando quiser. (No nosso caso quando a tarefa tiver o recurso A (nutrientes) e o recurso B (espaço)).

Espera Circular: ciclo de esperas: a tarefa t1 quer um recurso retido por t2, que quer um recurso retido por t3, que quer um recurso retido por t1. (No nosso caso serão as Colônias (threads) disputando os dois recursos).

Essas quatro condições são necessárias (não suficientes) para a ocorrência de impasses.

Se quisermos que o impasse não aconteça, temos que evitar uma dessas condições.

1. **Código com impasse**

Falarei primeiro dos elementos presentes no código que gera impasse, nome <bacterias_impasse.c>

1.1. Bibliotecas usadas

1° Biblioteca <unistd.h>

A biblioteca <unistd.h> é usada principalmente em sistemas Unix/Linux e contém várias funções úteis para manipular o ambiente do sistema operacional, como o sleep. No Windows, essa biblioteca não está disponível nativamente. Então, usaremos a biblioteca <windows.h>, para a função Sleep.

A função Sleep da biblioteca <windows.h> recebe o valor em milissegundos, então para 1 segundo ficará Sleep(1000);.

2° Biblioteca <math.h>

Para usarmos a função exp(), que será importante para calcular a fórmula de crescimento exponencial dada pelo exercício.

Linha no código usando a função exp():

```
double populacao = colonia->p0 * exp(colonia->r * t);
```

3° Biblioteca <pthread.h>

Para trabalharmos com funções específicas sobre threads.

Partes da biblioteca são derivadas de: “Posix Threads library for Microsoft Windows” (Biblioteca Posix Threads para Microsoft Windows, em tradução).

4° Demais bibliotecas

As demais bibliotecas usadas, como <stdio.h> e <stdlib.h> fazem parte das funções mais básicas da linguagem C.

1.2. Explicações sobre cada parte do código:

Struct para cada Colônia

```
typedef struct {
    int id;
    int p0;
    float r;
    int tempo_total;
    pthread_mutex_t *recursoA;
    pthread_mutex_t *recursoB;
} Colonia;
```

Cada Colônia terá um id para melhor identificação quando for printar (usar printf), assim podemos saber a execução do código passo a passo, e ver quais threads estão sendo executadas.

A variável “int p0”, é a população inicial de cada colônia, definida no momento de compilação. Assim como a variável “float r” para a taxa de crescimento.

1.3. Módulo “simular_crescimento”

Composto pela lógica de crescimento das threads, definido no cálculo da fórmula:

“double populacao = colonia->p0 * exp(colonia->r * t);”

Essa linha retrata a fórmula passada na descrição do exercício, que foi:

$P(t) = Po \cdot e^{(r \cdot t)}$

Onde:

P(t) - é a população no tempo t.

Po - é a população inicial.

r - é a taxa de crescimento.

t - é o tempo.

e - é a base dos logaritmos naturais.

Este módulo também apresenta o laço “for” que irá executar as threads distribuindo os recursos. Estes estão sendo controlados por mutexes.

A condição para o primeiro mutex é simples: se a thread tiver um “id” par ela pode entrar.

Se não entrar, irá para o segundo mutex.

O primeiro mutex representa a distribuição do recurso A (nutrientes), enquanto o segundo mutex representa o recurso B (espaço).

Assim, threads pares irão ter antes o recurso A e threads ímpares o recurso B.

Ao final do módulo há a liberação dos recursos que foram pegos por cada thread.

1.4. Módulo principal (int main)

Na “main” temos a criação dos parâmetros que após execução via comando serão “capturados” e mandados para cada argumento (argv).

Os parâmetros são:

Uso: %s <Populacao inicial> <Taxa de crescimento> <Tempo total> <Numero de colonias> <Numero de recursos de cada tipo>\n", argv[0]

Onde:

argv[0] será o nome do arquivo escolhido na compilação.

argv[1] é a população inicial definida por ”double p0”. (população inicial definida para todas as threads).

argv[2] é a taxa de crescimento definida pelo “double r”. (valor em porcentagem, 0.1 indica 10%).

argv[3] é o tempo total que será definido para a variável “int tempo_total”. (representa os tempos, começando em 0).

argv[4] é o número de colônias que será definido para a variável “int num_colonias”. (representa o número de threads).

argv[5] é o número de recursos de cada tipo que será definido para a variável “int num_recursos”. (representa o número de recursos disponíveis).

Além disso, temos a criação dos mutexes com os “recursosA” e os “recursosB” que serão baseados no número de recursos do argv[5], junto com as threads que serão usadas.

Assim como, a inicialização dos mutexes com o laço “for”, assim como a criação das colônias (threads). Por fim, a destruição dos mutexes (equivalente ao free da alocação de memória).

1.5. Como executar o código (windows) e as diferenças para Linux

Talvez seja necessário mudar a biblioteca <windows.h> (sistemas windows) para <unistd.h> (sistemas linux).

Lembrando que, com isso a função sleep(1); (biblioteca unistd.h) passará a ser Sleep(1000); (biblioteca windows.h)* e vice-versa.

*Pode haver mais diferenças do que apenas essa.

No Git Bash (terminal)

gcc -o <nome arquivo> <nome arquivo.c> -lpthread -lm

./<nome arquivo> 100 0.1 10 4 2

Onde:

100: População inicial para cada colônia
0.1: Taxa de crescimento (10%)
10: Tempo total de simulação
4: Número de colônias (threads)
2: Número de recursos de cada tipo

Para dar deadlock teste a seguinte sequência*:
100 0.1 10 4 1

*Talvez a execução demore para ser executada.

Usando 1 no número de recursos de cada tipo (último parâmetro), diminuiremos o número de recursos, isso é, apenas um mutex para o recurso A e um mutex para o recurso B, para qualquer número de colônia acima de 1 dará deadlock, pois com somente duas colônias, uma esperará pela outra.

Para evitar deadlock abaixando a quantidade de recursos, coloque apenas uma colônia: 100 0.1 5 1 1

Número de recursos de cada tipo = 2 significa que há dois mutexes disponíveis para Recurso A (nutrientes) e dois mutexes para Recurso B (espaço).

2. Código com monitor de impasses

Aqui apresentarei uma atualização para o código anterior, com a implementação de um monitor, para vermos o impasse acontecendo e quais as threads envolvidas. Nome <bacterias_com_monitor.c>

2.1. Mudanças

Teremos uma variável global indicando o número de recursos:
“#define NUM_RECURSOS 1”

Que será usado para definir o tamanho dos mutexes:
pthread_mutex_t recursoA[NUM_RECURSOS];
pthread_mutex_t recursoB[NUM_RECURSOS];

Isso não alterará o argv[5] num_recursos, isso serve apenas para inicialização desses mutexes.

Na struct de cada thread teremos uma variável a mais, que será chamada de “int *status”, um array para monitorar o status da thread.

Além disso, temos um limite máximo para esse status, definido por:

“int thread_status[100]; //máximo de 100 threads”

No módulo “simular_crescimento” temos a inserção da seguinte linha em cada um dos mutexes:

“colonia->status[colonia->id] = 1;”.

Essa linha se refere que uma thread está esperando o próximo recurso, assim ela terá esse status. Quando a thread obter o próximo recurso e liberá-los, temos a linha:

“colonia->status[colonia->id] = 0;”.

2.2. Módulo “monitor_deadlock”

Aqui temos a lógica para monitorar as threads que estão paradas esperando recursos.

Como definido que, quando uma thread “pega” um recurso, ela recebe o status 1.

“colonia->status[colonia->id] = 1;“

Então, no laço while entrarão somente as threads que possuem recurso. Uma verificação é feita a cada 5 segundos para ver se a thread ainda possui o recurso, se a thread ainda ter o recurso e estiver esperando por 3 ciclos de 5 segundos, possivelmente entrou em impasse (deadlock), assim a mensagem é exibida.

2.3. Mudanças no módulo principal (int main)

As linhas:

“pthread_mutex_t recursosA[num_recursos];

pthread_mutex_t recursosB[num_recursos]“

foram substituídas pela criação da thread de monitoramento:
“pthread_t monitor_thread;”
“pthread_create(&monitor_thread, NULL, monitor_deadlock, (void *)thread_status);”

E o seu cancelamento:
“pthread_cancel(monitor_thread);”

2.4. Como executar o código (windows)

No Git Bash (terminal)
gcc -o <nome arquivo> <nome arquivo.c> -lpthread -lm
./<nome arquivo> 100 0.1 3 5 1

*Aguarde alguns instantes para aparecer a mensagem de deadlock.

3. Código com prevenção de impasse

Agora, o código estará completo junto com o monitor e a prevenção do impasse. Nome <bacterias_prevencao_impasse.c>

Dada as seguintes alternativas das técnicas para prevenção de impasses

Prevenção por Ordenação de Recursos: Todas as colônias devem adquirir os recursos na mesma ordem.

Mutex Hierárquico: As colônias devem seguir uma hierarquia de mutexes para evitar ciclos de dependência.

Timeout ou Recurso de Recuperação: Se uma colônia ficar esperando por muito tempo, deve liberar o recurso que já possui e tentar novamente.

Foi escolhida a técnica de Prevenção por Ordenação de Recursos, tirando a verificação se a thread par precisa obter o recursoA antes do recursoB e a thread ímpar o verso.

3.1. Modificações

No módulo “simular_crescimento”, a verificação “if (colonia->id % 2 == 0)” foi tirada, assim como a repetição do mutex_lock de alguns recursos (para não se repetir):

Ficando:
“pthread_mutex_lock(colonia->recursoA); e
“pthread_mutex_lock(colonia->recursoB);”.

3.2. Como executar o código (windows)

No Git Bash (terminal)
gcc -o <nome arquivo> <nome arquivo.c> -lpthread -lm
./<nome arquivo> 100 0.1 3 5 1

*Após aguardar alguns instantes o deadlock não acontecerá

4. Análise de desempenho

O desempenho dos diferentes códigos disponibilizados não se altera tanto quando comparados um em relação ao outro. Portanto, vale destacar que conforme aumenta os valores iniciais definidos na execução, o código pode demorar mais para ser executado. Por exemplo, se você usar o código <bacterias_prevencao_impasse.c>:

Colocando uma população inicial de 2500 junto com uma taxa de crescimento de 0.5 (50%), com um tempo total de 10 (maior) e 20 colônias, a depender do hardware utilizado, recomendo esperar uns bons minutos para tudo ser executado até o fim.

Isso pois, o crescimento dado pela fórmula será exponencial, agregado que a população inicial começou grande, e que ocorrerá uma multiplicação do tempo total pelas colônias, você terá 10 x 20 = 200 “iterações”, por assim dizer.

Assim como, se você testar o impasse no código <bacterias_com_monitor.c> com um certo número dos outro parâmetros aumentados (menos o número de recursos de cada tipo), você terá certa lentidão a depender do seu hardware.

Conclusão

A implementação da prevenção de impasses foi fácil, quando escolhida a técnica certa para isso.

O exercício proposto teve grande importância para a formação do conhecimento, pois foi posto em prática os conhecimentos antes vistos em teoria na sala de aula e questionários na plataforma moodle.

Terminal teste com todos os códigos:

```
User@DESKTOP-85813LT MINGW64 ~/OneDrive/Área de Trabalho
$ gcc -o impasse bacterias_impasse.c -lpthread -lm

User@DESKTOP-85813LT MINGW64 ~/OneDrive/Área de Trabalho
$ ./impasse 100 0.1 2 1 2
Colonia 0 obteve Nutrientes (Recurso A)
Colonia 0 obteve Espaco (Recurso B)
Colonia 0 - Tempo 0: Populacao = 100.00
Colonia 0 obteve Nutrientes (Recurso A)
Colonia 0 obteve Espaco (Recurso B)
Colonia 0 - Tempo 1: Populacao = 110.52

User@DESKTOP-85813LT MINGW64 ~/OneDrive/Área de Trabalho
$ gcc -o monitor bacterias_com_monitor.c -lpthread -lm

User@DESKTOP-85813LT MINGW64 ~/OneDrive/Área de Trabalho
$ ./monitor 100 0.1 2 1 2
```

Colonia 0 obteve Nutrientes (Recurso A)
Colonia 0 obteve Espaco (Recurso B)
Colonia 0 - Tempo 0: Populacao = 100.00
Colonia 0 obteve Nutrientes (Recurso A)
Colonia 0 obteve Espaco (Recurso B)
Colonia 0 - Tempo 1: Populacao = 110.52

User@DESKTOP-85813LT MINGW64 ~/OneDrive/Área de Trabalho
$ gcc -o prevencao bacterias_prevencao_impasse.c -lpthread -lm

User@DESKTOP-85813LT MINGW64 ~/OneDrive/Área de Trabalho
$ ./prevencao 100 0.1 2 1 2
Colonia 0 obteve Nutrientes (Recurso A)
Colonia 0 obteve Espaco (Recurso B)
Colonia 0 - Tempo 0: Populacao = 100.00
Colonia 0 obteve Nutrientes (Recurso A)
Colonia 0 obteve Espaco (Recurso B)
Colonia 0 - Tempo 1: Populacao = 110.52

Link para repositório no Github (repositório público)

https://github.com/felipebfava/S_O_

https://github.com/felipebfava/S_O_/tree/main/S_O_/Simu_Crescimento_Threads