Lista de Exercícios CompConc

Nome: Thalles Nonato DRE: 119058809

Questão 1 (2,5 pts) Responda as questões abaixo, **justificando todas as respostas**:

(a) O que caracteriza que um programa é concorrente e não sequencial?
(b) Para quais tipos de problema a programação concorrente é indicada?
(c) Qual será a *aceleração máxima* de uma aplicação que possui 5 tarefas que consomem o mesmo tempo de processamento, das quais 3 poderão ser executadas de forma concorrente e 2 precisarão continuar sendo executadas de forma sequencial?
(d) O que é *seção crítica* do código em um programa concorrente?
(e) Como funciona a *sincronização por exclusão mútua*?

Respostas:

a) Programa concorrente esta relacionado com programas de computador que incluem linhas de controle diferentes, e podem executar simultaneamente. Um programa concorrente diferencia-se de um programa sequencial por conter mais de um contexto de execução ao mesmo tempo.

b) Programação concorrente é indicada para as aplicações aonde possam ocorrer divisões de tarefas, visando projetar uma solução que terá um maior ganho de desempenho, comparado ao algoritmo sequencial. Também é indicada para algumas aplicações, como a necessidade de capturar a estrutura lógica de um problema, como em servidores web e aplicações gráficas. E também, a necessidade de lidar com dispositivos independentes como sistemas operacionais

c) Lei de Amdahl:

$$S(n) = \frac{1}{(1-p)} = \frac{1}{(1-0.6) + \frac{0.6}{n}}$$

Onde N é o número de threads e P a porcentagem de tarefas feitas concorrentemente. Pensando no caso em que n tende ao infinito, temos:

$$\lim_{x \to \infty} S(n) = \frac{1}{1-p} = \frac{1}{1-0.6} = 2.5$$

d) Seção crítica consiste em uma região do código que compartilha informações entre as threads, podendo assim diferentes threads acessarem e manipularem uma mesma informação.

e) A sincronização por exclusão mútua é uma estratégia com o objetivo de impedir que uma informação seja acessada por mais de uma thread ao mesmo tempo, de forma que, enquanto uma thread estiver acessando um determinado dado, o acesso será bloqueado para qualquer outra thread.

2)

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <math.h>

long int ntermos; // número de termos que iremos escolher para PI
int nthreads; // número de threads
int contador = 0; // contador global para averiguar quantas threads passou pela tarefa

double somatorio = 0.0; // somatório para PI
pthread_t *tid; // tid thread
pthread_mutex_t alock; // mutex para sincronização


void *tarefa( void *arg){
    int id = * (int*) arg;
    for(int i = id ; i <ntermos ; i+=nthreads){ //iterador para calcular o valor de pi
        pthread_mutex_lock(&alock);// com n termos /sincronizar para não ter erro na soma
        somatorio+=  pow(-1, i) * (1.0/(2.0*i+1)); //soma
        pthread_mutex_unlock(&alock); //sincronização
    }


    pthread_mutex_lock(&alock); // sincronização para o contador não ter erro
    contador++;
    pthread_mutex_unlock(&alock);

    if(contador == nthreads){   // quando o contador for igual ao numero de threads
        somatorio = 4* somatorio;//significa  que todas as threads fizeram sua tarefa
    } // então multiplicamos por 4
    pthread_exit(NULL);
}
```

A ideia é que cada thread seja responsável por cuidar dos seus termos, para evitar problemas de sincronização no somatório, usamos o mutex tanto para o somatório ,

quanto para o contador. Quando nosso contador for igual ao número de threads, significa que terminamos nosso processo e podemos multiplicar o somatório por 4 e nosso π esta definido.

3) Analisando as instruções de máquina das operações temos que:

```
t1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    x, %eax
        subl    $1, %eax
        movl    %eax, x
        movl    x, %eax
        addl    $1, %eax
        movl    %eax, x
        movl    x, %eax
        subl    $1, %eax
        movl    %eax, x
        movl    x, %eax
        cmpl    $-1, %eax
        jne     .L3
        movl    x, %eax
        subl    $8, %esp
        pushl   %eax
        pushl   $.LC0
        call    printf
        addl    $16, %esp
```

```
t2:
        pushl   %ebp
        movl    %esp, %ebp
        movl    x, %eax
        addl    $1, %eax
        movl    %eax, x
        movl    x, %eax
        subl    $1, %eax
        movl    %eax, x
        nop
        popl    %ebp
        ret
```

```
t3:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    x, %eax
        addl    $1, %eax
        movl    %eax, x
        movl    x, %eax
        cmpl    $1, %eax
        jne     .L7
        movl    x, %eax
        subl    $8, %esp
        pushl   %eax
        pushl   $.LC1
        call    printf
        addl    $16, %esp
```

Analisando as instruções de máquina, e que elas podem ser entrelaçadas, podemos ter os seguintes casos:

Para o Valor -1: Caso mais básico, começa na thread 1 com x =0, as 3 instruções são feitas sequencialmente, portanto. X = -1 + 1 -1 = -1. Entra no print do if com valor -1. E, após isso são executadas as outras threads.

Para o Valor 0: Começa na thread 1 com x = 0 , as 3 instruções são feitas sequencialmente, portanto. X = -1 + 1 -1 = -1. Ao entrar no if ações de outras threads são feitas. Um exemplo seria entrar na thread 2 , na instrução x = x + 1. Logo , x = 0 , após isso é chamado o printf anterior da thread 1, só que dessa vez com o valor 0 como printado.

Para o Valor 2: Começa na thread 3 com x = 0 , executa a instrução x = x +1. Entra no if da t3, antes do printf da t3, a t2 é executada, faz a operação x = x +1 , e volta para t3 com valor 2.

Para o Valor -2: Começa na thread 1 com x = 0 , executa a primeira operação , x = -1 . E a thread 1 executa sua segunda instrução, junto com a thread 2 executando a primeira, e no caso é exercido somente 1 incremento, portanto x = 0. A thread t3 executa sua instrução e incrementa x, portanto x = 1 e entra em seu if. Ao entrar em seu if, é executada a 3 operação da thread 1 e a segunda operação da thread 2, decrementando duas vezes o valor de x = 0 . Portanto x = -2.

Para o Valor 3: Começa na thread 3 com x = 0 , e executa a primeira operação, logo x = 1. Após isso a thread 2 é executada junto com a thread 1 , todavia o decremento da t1 é descartado, então x = 2. Depois disso, a segunda operação de t1 é chamada e incrementa x, portanto x = 3 , e é chamado o printf de t3 , com o valor 3.

Esses são os valores possíveis.

4)

a) Se T0 executar primeiro que T1, T1 irá ficar esperando a execução da seção crítica de T0, para quando T0 acabar seu processo na seção crítica, trocar a condição de queroEntrar_0 para T1 poder sair de seu loop e entrar na sessão crítica também.

b) Se T1 executar primeiro que T0, T0 irá ficar esperando a execução da seção crítica de T1, para quando T1 acabar seu processo na seção crítica, trocar a condição de queroEntrar_1 para T0 poder sair de seu loop e entrar na sessão crítica também.

c) Caso as threads executem ao mesmo tempo, como elas compartilham de uma mesma variável como condição do loop , a variável TURN. Na hora da execução , TURN vai assumir o valor 0 ou o valor 1, e isso garante que para alguma das threads a espera vai acontecer ao entrar no loop, para a outra thread conseguir acessar a seção crítica.

d) Essa implementação garante exclusão mútua, já que uma das threads fica em espera até que a outra termine a execução da sessão crítica.