

Módulo 2 - Lista de Exercícios 2 (2021/1 REMOTO)

Computação Concorrente (MAB-117)
Prof. Silvana Rossetto

¹Instituto de Computação/UFRJ
Julho de 2021

Questão 1 (2,5 pontos) Considere o seguinte problema. Dado um vetor de inteiros, precisamos calcular as somas parciais em cada posição desse vetor (somatório de todos os elementos que antecedem essas posições, incluindo o elemento da própria posição). Por exemplo, dado o vetor $[1, 4, -1, 7]$ como entrada, o vetor resultante com as somas parciais em todas as posições é: $[1, 5, 4, 11]$. O primeiro elemento se mantém, o segundo elemento é a soma de $1 + 4$, o terceiro elemento é a soma de $1 + 4 + (-1)$, e o quarto elemento é a soma de $1 + 4 + (-1) + 7$. O algoritmo sequencial para resolver esse problema é bastante simples e mostrado abaixo. Repare que o próprio vetor de entrada é modificado, isto é, usamos apenas um vetor que é alterado como saída do programa.

```
for(int i=1; i<n; i++)
    vetor[i] = vetor[i] + vetor[i-1];
```

O programa a seguir implementa uma solução concorrente para esse problema. Considera-se que o número de elementos do vetor de entrada (n) é sempre uma potência de 2 e que o número de threads será sempre igual ao número de elementos do vetor.

```
/* Variaveis globais */
int bloqueadas = 0;
pthread_mutex_t x_mutex;
pthread_cond_t x_cond;
int nthreads;
int *vetor;

/* Funcao barreira */
void barreira(int nthreads) {
    pthread_mutex_lock(&x_mutex);
    if (bloqueadas == (nthreads-1)) {
        //ultima thread a chegar na barreira
        pthread_cond_broadcast(&x_cond);
        bloqueadas=0;
    } else {
        bloqueadas++;
        pthread_cond_wait(&x_cond, &x_mutex);
    }
    pthread_mutex_unlock(&x_mutex);
}

/* Funcao das threads */
void *tarefa (void *arg) {
    int id = *(int*)arg;
    int salto;
    int aux;
    for(salto=1; salto<nthreads; salto*=2) {
        if(id >= salto) {
            aux = vetor[id-salto];
            barreira(nthreads-salto);
            vetor[id] = aux + vetor[id];
            barreira(nthreads-salto);
        } else break;
    }
    pthread_exit(NULL);
}

/* Funcao principal */
int main(int argc, char *argv[]) {
    pthread_t threads[nthreads];
```

```

int id[nthreads];
/* Inicializa o mutex (lock de exclusao mutua) e a variavel de condicao */
...
/* Recebe os parametros de entrada (tamanho do vetor == número de threads) */
...
/* Inicia as variaveis globais e carrega o vetor de entrada */
...
/* Cria as threads */
for(int i=0;i<nthreads;i++) {
    id[i]=i;
    pthread_create(&threads[i], NULL, tarefa, (void *) &id[i]);
}
/* Espera todas as threads completarem */
for (int i = 0; i < nthreads; i++) {
    pthread_join(threads[i], NULL);
}
/* Armazena o vetor de saida, libera variáveis e encerra */
...
return 0;
}

```

Tarefa: Considerando a solução concorrente apresentada no programa acima, responda as perguntas abaixo, **justificando todas as respostas:**

- Quais são as etapas principais do programa? Como ele funciona em detalhe?
- A solução apresentada está correta? O resultado dela equivale ao resultado que seria obtido com algoritmo sequencial apresentado?
- Por que são necessárias duas chamadas de sincronização coletiva (implementada pela função *barreira()*)? Elas poderiam ser substituídas por apenas uma?

Questão 2 (2,0 pontos) Uma aplicação em C dispara uma thread T1 para execução (código mostrado abaixo).

```

long long int contador=0;
void *T1 (void *) {
    while(1) {
        FazAlgo(contador);
        contador++;
    }
}

```

Tarefa: Implemente uma thread adicional T2 para necessariamente imprimir na tela o valor da variável *contador* sempre que ele for múltiplo de 100 (indicando que a função *FazAlgo* foi executada por mais 100 vezes). Crie outras variáveis globais e altere o código de T1, caso necessário. **Comente seu código.**

Questão 3 (3,0 pontos) O código Java abaixo implementa um *pool de threads* com dois métodos públicos: *execute* (escalona uma tarefa para execução pelo pool) e *shutdown* (encerra o pool depois que todas as tarefas já escalonadas foram finalizadas). Um *pool de threads* permite construir aplicações com alocação dinâmica de tarefas para as threads. As threads são criadas e disparadas e no seu método *run* elas executam um loop central onde elas esperam por novas tarefas e as executam, enquanto houver tarefas e a aplicação não for encerrada.

Em uma aplicação foi criado um pool de threads com 10 threads e foram disparadas 100 tarefas para execução pelo pool. Em seguida o pool de threads foi encerrado. A aplicação foi executada várias vezes. Ocorreu que em uma das execuções a aplicação não finalizou.

Tarefa: (a) Descreva como essa implementação do *pool de threads* funciona. (b) Identifique o erro no código e mostre como corrigi-lo. **Justifique suas respostas.**

```

1: class FilaTarefas {
2:     private int nThreads; private MyPoolThreads[] threads;
3:     //similar a um vetor de objetos Runnable
4:     private LinkedList<Runnable> queue;
5:     private boolean shutdown;

6:     public FilaTarefas(int nThreads) {
7:         this.shutdown=false; this.nThreads=nThreads;
8:         queue=new LinkedList<Runnable>();
9:         threads = new MyPoolThreads[nThreads];
10:        for (int i=0; i<nThreads; i++) {
11:            threads[i] = new MyPoolThreads();
12:            threads[i].start(); }
13:    }
14:    public void execute(Runnable r) {
15:        synchronized(queue) {
16:            if (this.shutdown) return;
17:            queue.addLast(r); //inclui um novo elemento na lista 'queue'
18:            queue.notify();
19:        }
20:    }
21:    public void shutdown() {
22:        synchronized(queue) { this.shutdown=true; }
23:        for (int i=0; i<nThreads; i++)
24:            try { threads[i].join(); }
25:            catch (InterruptedException e) {return;}
26:    }
27:    private class MyPoolThreads extends Thread {
28:        public void run() {
29:            Runnable r;
30:            while (true) {
31:                synchronized(queue) {
32:                    //verifica se a lista está vazia...
33:                    while (queue.isEmpty() && (!shutdown)) {
34:                        try { queue.wait(); }
35:                        catch (InterruptedException ignored){}
36:                    }
37:                    if(queue.isEmpty() && shutdown) return;
38:                    //retira o primeiro elemento da lista e o retorna
39:                    r = (Runnable) queue.removeFirst();
40:                }
41:                try { r.run(); } catch (RuntimeException e) {}
42:            } } } }

```

Questão 4 (2,5 pontos) Considere o padrão leitores e escritores com seus requisitos básicos como foram apresentados: (i) mais de um leitor pode ler ao mesmo tempo; (ii) apenas um escritor pode escrever de cada vez; (iii) nenhum leitor pode ler enquanto um escritor escreve. Implementando os requisitos básicos do problema, podemos ter situações em que os escritores são impedidos de executar a escrita por um longo intervalo de tempo (causando um problema conhecido como *inanição*).

Tarefa: (a) Discuta (mostre exemplos) em que situações isso pode acontecer. (b) Implemente uma solução concorrente em **Java** para o padrão leitores e escritores que minimize o tempo de espera dos escritores, fazendo com que, todas as vezes que um escritor tentar escrever e existir leitores lendo, a entrada de novos leitores fique impedida até que todos os escritores em espera sejam atendidos. Garanta que seu código esteja correto e não bloqueie indefinidamente nenhuma thread. Use *notifyAll* apenas no caso de *notify* não atender. **Comente seu código.**