

Os arquivos originais estarão no github, assim como esse pdf

[Questão 1](#)

[Questão 2](#)

[Questão 3](#)

[Questão 4](#)

## Questão 1

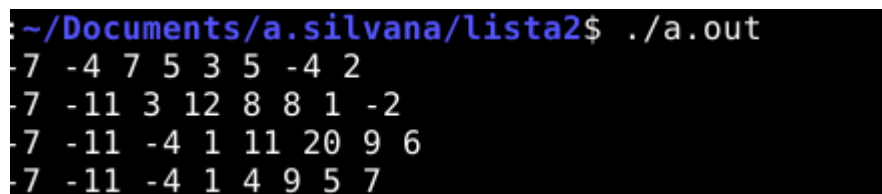
- a. As principais etapas são a tarefa() e a barreira(). A barreira() é a mais fácil de explicar porque apenas faz as threads esperarem até todas terem chegado nela. É necessário saber quantas estão ativas, pois vão diminuindo com a primeira condição `if(id>=salto)`. O argumento da função serve exatamente para obter esse valor (`nthreads - salto`)

A tarefa() é bem diferente do algoritmo sequencial:

Cada valor do vetor final precisa ser a soma dos elementos anteriores mais ele mesmo. Percebe-se que o algoritmo dobra o salto a cada iteração, isso porque cada salto dobra o número de elementos originais contidos em uma das posições.

Os valores mais próximos do início precisam somar menos, logo suas threads saem do loop (`break;`) mais cedo.

Observe a imagem do vetor em cada passo:



```
~/Documents/a.silvana/lista2$ ./a.out
7 -4 7 5 3 5 -4 2
7 -11 3 12 8 8 1 -2
7 -11 -4 1 11 20 9 6
7 -11 -4 1 4 9 5 7
```

posição 0 já possui valor final, break no início.

posição 1:  $-4 > -11 > \text{break};$

como podemos observar, a posição 1 já está completa no primeiro salto

posição 2:  $7 > 3 > -4 > \text{break};$

posição 3:  $5 > 12 > 1 > \text{break};$

estão^ completas a partir do salto 2

posição 4:  $3 > 8 > 11 > 4 > \text{break};$

posição 5:  $5 > 8 > 20 > 9 > \text{break};$

posição 6:  $-4 > 1 > 9 > 5 > \text{break};$

posição 7:  $2 > -2 > 6 > 7 > \text{break};$

estão completas a partir do salto 3

Assim por diante, as 2^salto posições estão completas no salto de mesmo valor

Esse algoritmo possui somas redundantes, como podemos observar mais facilmente com as posições 1 e 2:

posição 2 =  $(-4 + 7) - 7 = -3 + 7 = -4$

posição 1 =  $-7 - 4 = -11$

Observe que, duas vezes, o -7 foi somado, totalizando 3 somas no total

Num algoritmo sequencial teríamos

soma 1 =  $-7 - 4 = -11$

e soma 2 =  $-11 + 7 = -4$

totalizando 2 somas apenas, menos do que o algoritmo concorrente.

- b. Como visto, está correta, basta olhar a primeira e última linha e checar cada elemento
- c. A segunda barreira serve para impedir que o aux seja lido antes da linha:

```
vetor[id] = aux + vetor[id];
```

ser executada para todos os threads. Retirar isso fará o resultado variar:

```
:~/Documents/a.silvana/lista2$ ./a.out
resultado: -7 -11 -4 1 4 2 5 7
~/Documents/a.silvana/lista2$ ./a.out
resultado: -7 -11 -4 1 8 2 12 4
~/Documents/a.silvana/lista2$ ./a.out
resultado: -7 -11 -4 1 8 2 0 7
~/Documents/a.silvana/lista2$ ./a.out
resultado: -7 -11 -4 1 4 2 7 15
~/Documents/a.silvana/lista2$ ./a.out
resultado: -7 -11 -4 1 4 2 7 4
~/Documents/a.silvana/lista2$ ./a.out
resultado: -7 -11 -4 1 8 2 0 4
~/Documents/a.silvana/lista2$ ./a.out
resultado: -7 -11 -4 8 4 9 7 11
```

## Questão 2

```
34
33 /* Variaveis globais */
32 int nthreads = 2;
31 long long int contador=0;
30 int imp = 1;
29
28 pthread_cond_t x_cond;
27 pthread_mutex_t x_mutex;
26
25 /* Funcao das threads */
24 void FazAlgo(int c){
23     usleep(50); // dorme pro 0.05 segundos
22     if(contador%100==0){
21         imp = 1; // mostra que quer entrar no loop
20         pthread_mutex_lock(&x_mutex);
19         pthread_cond_broadcast(&x_cond); // autoriza o T2 a imprimir
18         while(imp) // entra no loop caso imp = 1
17             pthread_cond_wait(&x_cond,&x_mutex); // sai apenas com o broadcast do T2
16         pthread_mutex_unlock(&x_mutex);
15     }
14 }
13 void *T1 (void *v) {
12     while(1) {
11         FazAlgo(contador);
10         contador++;
9     }
8 }
7
6 void *T2 (void *v) {
5     while(1) {
4         pthread_mutex_lock(&x_mutex);
3         pthread_cond_wait(&x_cond,&x_mutex);
2         printf("contador = %lld\n",contador); // imprime
1         imp = 0; // impede T1 de reentrar no loop
39        pthread_cond_broadcast(&x_cond); // retira T1 da condição de espera
1         pthread_mutex_unlock(&x_mutex);
2     }
3 }
4 }
```

O programa completo está no git...

```
:~/Documents/a.silvana/lista2$ ./a.out
contador = 0
contador = 100
contador = 200
contador = 300
contador = 400
contador = 500
contador = 600
contador = 700
contador = 800
contador = 900
contador = 1000
contador = 1100
contador = 1200
contador = 1300
contador = 1400
contador = 1500
contador = 1600
contador = 1700
contador = 1800
contador = 1900
contador = 2000
contador = 2100
contador = 2200
contador = 2300
contador = 2400
contador = 2500
contador = 2600
contador = 2700
contador = 2800
contador = 2900
contador = 3000
contador = 3100
contador = 3200
contador = 3300
contador = 3400
contador = 3500
contador = 3600
contador = 3700
contador = 3800
contador = 3900
^C
:~/Documents/a.silvana/lista2$
```

### Questão 3

- Como funciona? O construtor de FilaTarefas inicia vários MyPoolThreads, que permite uma fila de nThreads até que a função shutdown seja lançada. Novas threads são colocadas na fila com a função execute(), e são iniciadas tão rápido quanto o while(1) dentro de cada um dos MyPoolThreads consegue rodar. Depois disso, usar shutdown() marca o MyPoolThreads para parar e a impede de receber novos fluxos. Quando todas as threads pararem, ele retorna.

- b. O código funciona no meu computador sem nenhum erro... Duas coisas que podem ser erros:

O programa original cria 10 pools diferentes e cada um executa simultaneamente um loop while até parar, isso poderia ser reduzido para 1 loop while. Ter 10 pools não é um erro, só é muito estranho, e pode ocasionar erros se o programa sustentar-se por alguma probabilidade dentro de MyPoolsThreads.

Outro possível erro, mais provável de ser o que se pede, é o shutdown não parar as threads sendo executadas, fazendo um loop infinito ser possível. Acredito que o objeto Runnable não possua uma maneira de parar externamente, então o único jeito é criar um mecanismo para parar quando shutdown for iniciado.

```
}
static class Tarefa{
    static boolean mustRun=true;
    public static void shut(){
        mustRun=false;
    };
}
static class TarefaFinita extends Tarefa implements Runnable{
    int id;
    TarefaFinita(int id){
        this.id=id;
    }
    public void run(){
        int bob1=1000,bob2=1000000;
        for(;bob1<bob2;bob1++){
            System.out.println("Tarefa "+id+" parando");
        }
    }
}
static class TarefaInfinita extends Tarefa implements Runnable{
    int id;
    TarefaInfinita(int id){
        this.id=id;
    }
    public void run(){
        while(mustRun){
            try{
                System.out.println("Tarefa "+id+" continua rodando");
                Thread.sleep(3000);//dorme por 3 segundos
            }catch (InterruptedException e){}
        }
        System.out.println("Tarefa "+id+" parando");
    }
}
}
```

Agora basta chamar uma função que finaliza as threads depois de chamar o shutdown().

```

15     public void shutdown() {
14         synchronized(queue){
13             this.shutdown=true;
12         }
11         new WaitAndForce().run();
10         for (int i=0; i<nThreads; i++)
9             try {
8                 threads[i].join();
7             } catch (InterruptedException e) {
6                 return;
5             }
4         }
3     class WaitAndForce implements Runnable{
2         public void run(){
1             try { Thread.sleep(5000); } catch (InterruptedException e){}
43         tarefa.shut();
1             }
2     }

```

Agora qualquer tarefa para

```

Tarefa 70 parando
Tarefa 69 parando
Tarefa 67 parando
Tarefa 80 parando
Tarefa 81 parando
Tarefa 82 parando
Tarefa 83 parando
Tarefa 84 parando
Tarefa 85 parando
Tarefa 86 parando
Tarefa 87 parando
Tarefa 88 parando
Tarefa 89 parando
Tarefa 90 parando
Tarefa 91 parando
Tarefa 92 parando
Tarefa 93 parando
Tarefa 94 parando
Tarefa 95 parando
Tarefa 96 parando
Tarefa 97 parando
Tarefa 66 parando
Tarefa 65 parando
Tarefa 64 parando
Tarefa 98 parando
Tarefa 79 parando
Tarefa 78 parando
Tarefa 77 parando
Tarefa 76 parando
Tarefa 75 parando
Tarefa 74 parando
Tarefa 99 continua rodando
Tarefa 99 continua rodando
Tarefa 99 parando
~/Documents/a.silvana/lista2$

```

#### Questão 4

- a. Como exemplo, basta olhar no laboratório anterior, quando o número de leitores é muito alto:

```
class LeitorEscritor {  
    static final int L = 1000;  
    static final int E = 3;  
}
```

le.leitorSaindo(261)  
le.leitorSaindo(306)  
le.leitorSaindo(284)  
le.leitorLendo(47)  
le.leitorLendo(48)  
le.leitorLendo(2)  
le.leitorLendo(49)  
le.leitorSaindo(318)  
le.leitorLendo(50)  
le.leitorLendo(51)  
le.leitorSaindo(568)  
le.leitorLendo(52)  
le.leitorLendo(53)  
le.leitorLendo(54)  
le.leitorSaindo(897)  
le.leitorLendo(55)  
le.leitorSaindo(270)  
le.leitorLendo(56)  
le.leitorSaindo(973)  
le.leitorLendo(57)  
le.leitorSaindo(860)  
le.leitorSaindo(996)  
le.leitorSaindo(972)  
le.leitorLendo(58)  
le.leitorSaindo(917)  
le.leitorLendo(59)  
le.leitorLendo(60)  
le.leitorLendo(61)  
le.leitorSaindo(998)  
le.leitorSaindo(969)  
le.leitorSaindo(271)  
le.leitorSaindo(841)  
le.leitorSaindo(916)  
le.leitorLendo(62)  
le.leitorSaindo(981)  
le.leitorLendo(63)  
le.leitorSaindo(963)  
le.leitorSaindo(957)  
le.leitorSaindo(989)  
le.leitorSaindo(983)  
le.leitorSaindo(984)  
le.leitorSaindo(997)  
le.leitorSaindo(910)  
le.leitorSaindo(987)  
le.leitorSaindo(877)  
le.leitorSaindo(975)  
le.leitorLendo(64)  
le.leitorSaindo(992)  
le.leitorSaindo(977)  
le.leitorSaindo(971)  
le.leitorLendo(65)  
le.leitorSaindo(964)  
le.leitorSaindo(960)



Preenche meu terminal, sem 1 escritor entrando.

Uma maneira de resolver isso seria colocar um limite de tempo ou uma variável de prioridade:

```
// Entrada para escritores
public synchronized void EntraEscritor (int id) {
    try {
        while ((this.leit > 0) || (this.escr > 0)) {
            System.out.println ("le.escritorBloqueado("+id+")");
            wait(200); //bloqueia pela condicao logica da aplicacao
            this.forceWriter=true;
            while(this.leit>0 || this.escr > 0){
                wait();
            }
        }
        this.escr++; //registra que ha um escritor escrevendo
        System.out.println ("le.escritorEscrevendo("+id+")");
    } catch (InterruptedException e) { }
}

// Saida para escritores
public synchronized void SaiEscritor (int id) {
    this.escr--; //registra que o escritor saiu
    this.forceWriter = false;
    notifyAll(); //libera leitores e escritores (caso existam leitores ou escritores bloqueados)
    System.out.println ("le.escritorSaindo("+id+")");
}
```

Agora basta pedir para os leitores esperar se for o caso:

```
// Entrada para leitores
public synchronized void EntraLeitor (int id) {
    try {
        while (this.escr > 0 | forceWriter == true) {
            System.out.println ("le.leitorBloqueado("+id+")");
            wait(); //bloqueia pela condicao logica da aplicacao
        }
        this.leit++; //registra que ha mais um leitor lendo
        System.out.println ("le.leitorLendo("+id+")");
    } catch (InterruptedException e) { }
}
```

Agora colocando um bando de leitor:

```
// Classe principal
class LeitorEscritor {
    static final int L = 1000;
    static final int E = 3;
```

O forceWriter pode ser verdadeiro em diferentes situações, não necessariamente depois de 0,2 segundos. O resultado é perfeitamente funcional, mas força muitos leitores a esperar, como observado:

```
le.escriptorEscrevendo(1)
le.escriptorSaindo(1)
le.leitorLendo(1)
le.leitorSaindo(1)
le.leitorLendo(2)
le.leitorSaindo(2)
le.escriptorEscrevendo(2)
le.escriptorSaindo(2)
le.escriptorEscrevendo(1)
le.escriptorSaindo(1)
le.leitorLendo(1)
le.leitorSaindo(1)
le.leitorLendo(3)
le.leitorSaindo(3)
le.escriptorEscrevendo(3)
le.escriptorSaindo(3)
le.escriptorEscrevendo(1)
le.escriptorSaindo(1)
le.leitorLendo(1)
le.leitorSaindo(1)
le.leitorLendo(4)
le.leitorSaindo(4)
le.leitorLendo(2)
le.leitorSaindo(2)
le.escriptorEscrevendo(2)
le.escriptorSaindo(2)
le.escriptorEscrevendo(1)
le.escriptorSaindo(1)
le.leitorLendo(1)
le.leitorSaindo(1)
le.leitorLendo(5)
le.leitorSaindo(5)
le.escriptorEscrevendo(1)
le.escriptorSaindo(1)
le.leitorLendo(1)
le.leitorSaindo(1)
le.leitorLendo(6)
le.leitorSaindo(6)
le.leitorLendo(3)
le.leitorSaindo(3)
le.leitorLendo(2)
le.leitorSaindo(2)
le.escriptorEscrevendo(3)
le.escriptorSaindo(3)
le.escriptorEscrevendo(2)
le.escriptorSaindo(2)
le.escriptorEscrevendo(1)
le.escriptorSaindo(1)
le.leitorLendo(1)
le.leitorSaindo(1)
```