Lista 2 de Computação Concorrente

Gabriel da Fonseca Ottoboni Pinho - DRE 119043838
Rodrigo Delpreti de Siqueira - DRE 119022353

17/05/2021

Questão 1

A

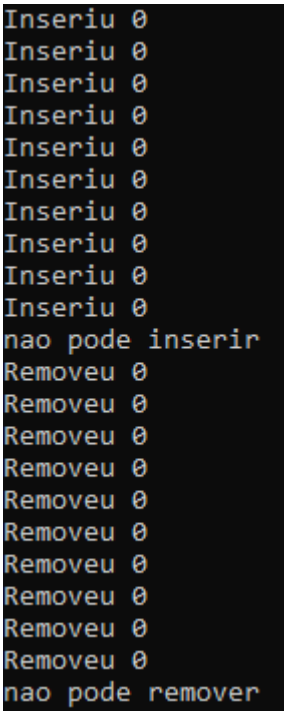
A alternância entre as execuções das threads é garantida. Após o início do programa, as threads `Bar` são imediatamente pausadas pela função `pthread_cond_wait`, enquanto que as de tipo `Foo` seguem livremente até que são pausadas depois de completar suas tarefas. Quando `M` threads `Foo` são pausadas, a função `pthread_cond_broadcast` é chamada e então as threads `Bar` são despausadas. Analogamente às threads `Bar`, as threads `Foo` são pausadas depois que terminam suas tarefas, com a última liberando as threads `Bar`, que estavam pausadas. Dessa forma, os dois tipos de thread são executados de maneira alternada.

B

As variáveis `contaFoo` e `contaBar` não sofrem de condições de corrida, pois todo acesso a essas é controlado por uma mutex. Contanto que nenhuma das threads termine, a alternância continua ocorrendo, sem a presença do *deadlock*. Caso contrário, entretanto, todas as threads restantes ficarão bloqueadas, pois `contaBar` ou `contaFoo` deixará de ser atualizado.

Questão 2

O deadlock irá ocorrer assim que a thread iniciar sua operação de leitura ou escrita. Realizamos alguns testes na execução do código e podemos ver no trecho abaixo como a thread assume o monopólio do recurso para si.



A causa está no uso do método notify(), ao invés do método notifyAll(). Abaixo segue o código com diversas correções, que corrige o problema do deadlock.

```
class Buffer {
    static final int N = 10; //qtde de elementos no buffer

    private int[] buffer; //area de dados compartilhada

    //variaveis de estado
    private int count; //qtde de posicoes ocupadas no buffer
    private int in; //proxima posicao de insercao
    private int out; //proxima posicao de retirada

    // Construtor
    Buffer() {
        this.count = 0;
        this.in = 0;
        this.out = 0;
```

```
        this.buffer = new int[N];
    }

    // Insere um item
    public synchronized void insere (int item) {
        while (count == N) {
            try{ wait(); }
            catch (InterruptedException e) { return; }
        }
        buffer[in % N] = item;
        in++;
        count++;
        notifyAll();
    }

    // Remove um item
    public synchronized int remove () {
        int aux;
        while (count == 0) {
            try{ wait(); }
            catch (InterruptedException e) { return -1; }
        }
        aux = buffer[out % N];
        out++;
        count--;
        notifyAll();
        return aux;
    }
}
```

Questão 3

A

Uma thread `A`, com a mutex adquirida, incrementou `x` para 10 e verificou que 10 é um múltiplo de 10. Com isso, a thread `B`, que estava pausada, foi despausada. Apesar disso, a função `pthread_cond_wait` só poderá de fato retornar depois que a mutex puder ser adquirida por `B`. Temos, então, uma condição de corrida, pois após a mutex ser liberada por `A`, não necessaria-

mente `B` será a próxima adquiri-la.

Para printar 11, após a chamada a `pthread_cond_signal` e a liberação da mutex, alguma outra thread `A` adquiriu a mutex e incrementou `x` mais uma vez. Depois, a mutex foi liberada e só então `B` conseguiu a adquirir, retornando de `pthread_cond_wait`. Por conta da aquisição tardia, o valor de `x` já não era mais 10 e 11 foi impresso.

B

A causa da condição de corrida é a forma como o valor de `x` é passado de `A` para `B`. Sabendo disso, essa troca de informações poderia ser feita usando um modelo de produtores e consumidores. As threads `A` seriam as produtoras e a thread `B` seria a consumidora. Isso resolveria a condição de corrida, pois o valor de `x` seria copiado para um buffer, garantindo a exitência do valor até que `B` o acesse.

Como não há nenhum loop na thread `B`, apenas o primeiro múltiplo será impresso. Tirando vantagem desse comportamento, uma versão simplificada de produtores e consumidores foi implementada. A variável `print_x` faz o papel do buffer, e `A` só escreve no buffer uma única vez. A mutex `print_mutex` não era estritamente necessária, mas foi adicionada a fim de manter o modelo do código original. A função `pthread_cond_wait` exige uma mutex e não faria sentido usar a mutex da outra variável, que nem é utilizada em `B`.

```
int x = 0;
pthread_mutex_t x_mutex;
pthread_cond_t x_cond;

int print_x = 1;
pthread_mutex_t print_mutex;

void *A (void *tid) {
    for (int i=0; i<100; i++) {
        pthread_mutex_lock(&x_mutex);
        pthread_mutex_lock(&print_mutex);
        x++;
        if(!(x%10) && print_x == 1) {
            print_x = x;
            pthread_cond_signal(&x_cond);
        }
        pthread_mutex_unlock(&print_mutex);
```

```
        pthread_mutex_unlock(&x_mutex);
    }
}

void *B (void *tid) {
    pthread_mutex_lock(&print_mutex);
    if(print_x%10)
        pthread_cond_wait(&x_cond, &print_mutex);
    printf("X=%d\n", print_x);
    pthread_mutex_unlock(&print_mutex);
}
```

Questão 4

A

A solução atende aos requisitos do problema. Um escritor não pode escrever enquanto um leitor lê, pois ele será pausado na linha 16. De forma similar, quando uma thread está escrevendo, ela possui a *lock*, o que faz com que leitores fiquem bloqueados ao chamar `EntraLeitor` até que a escrita termine. A *lock* também impede que dois escritores escrevam simultaneamente. Dessa forma, todos os requisitos são atendidos.

B

A substituição poderia ser feita sem problemas. É possível que haja mais de um escritor pausado no `wait` da linha 16, mas, mesmo assim, é garantido que ele será acordado em algum momento; ou pelo `notify` da linha 19, ou pelo `notifyAll` da linha 11.

É possível que o último leitor chame `SaiLeitor` e que uma possível chamada de `EntraLeitor` seja executada antes do leitor que estava esperando, fazendo com que ele seja pausado sem chamar o `notify` da linha 19. Mesmo nesse caso, é garantido que em algum momento o escritor vai conseguir a *lock* logo depois de ser acordado, pois `SaiLeitor` precisará ser chamado.

C

Com o `notifyAll` da linha 11, o `notify` da linha 19 pode ser removido sem problemas. A lógica é similar à da letra B: Se após o `notifyAll` da linha 11 todos os escritores executarem sem que um `EntraLeitor` seja executado entre eles, tudo dará certo. Se houver um `EntraLeitor` entre eles, alguns

escritores podem acordar e depois dormir imediatamente de novo. Mesmo assim, como uma chamada a `SaiLeitor` ocorrerá em algum momento, é garantido que o escritor será acordado novamente.