LISTA 03

13/10/2021

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
CIÊNCIA DA COMPUTAÇÃO
COMPUTAÇÃO CONCORRENTE (2021.1)

PROFESSORA: SILVANA ROSSETTO
ALUNO: IAGO RAFAEL L. MARTINS | DRE: 119181254

Questão 01 (2,5 pontos) Uma aplicação dispõe de um recurso que pode ser acessado por três tipos de *threads* diferentes (`A`, `B` e `C`). As *threads* de mesmo tipo podem acessar o recurso ao mesmo tempo, mas *threads* de tipos diferentes não (i.e., qualquer número de *threads* `A` ou qualquer número de *threads* `B` ou qualquer número de *threads* `C` podem acessar o recurso ao mesmo tempo, o que não é permitido são *threads* de tipos diferentes acessando o recurso ao mesmo tempo). O código abaixo implementa uma solução para esse problema (código que as *threads* `A`, `B` e `C` devem executar antes e depois de acessarem o recurso).

Considerando o que foi exposto, examine o código e responda, **justificando todas as respostas** (as justificativas contam na avaliação da questão):

a) Essa solução garante as condições do problema, ausência de condição de corrida e de *deadlock*?

Sim, essa solução garante as condições do problema perfeitamente. Não há condição de corrida por não existir no código nenhum momento em que mais de um tipo de *thread* acesse o recurso ao mesmo tempo. Da mesma forma, não existem *deadlocks* por não terem situações no código em que uma *thread* é bloqueada, mas não desbloqueada por nenhuma outra *thread*.

b) Essa solução pode levar as *threads* a um estado de *starvation* (espera por um tempo muito longo para serem atendidas)?

Sim, pois pensando na possibilidade de existir um número muito maior de um tipo de *thread*, é extremamente provável que o tempo necessário para que todas as *threads* desse tipo sejam executadas ocasione um estado de *starvation*, tendo em vista a restrição de que temos nas condições do problema: "o que não é permitido são *threads* de tipos diferentes acessando o recurso ao mesmo tempo". Deixando bem claro: essa restrição faz com que o tempo de todas as *threads* desse tipo serem escutadas seja bem maior em relação ao tempo de execução das *threads* de outros tipos.

c) O que aconteceria se o semáforo `rec` fosse inicializado com 0 sinais?

Nesse caso, ao rodar o código, aconteceria uma situação de ***deadlock*** com todos os tipos de *thread*, por cada uma delas ser bloqueada ao entrar no primeiro `if` dentro do `while`, especificamente, no "`sem_wait(&rec)`". Isso acontecerá justamente pelo fato delas não terem

um sinal para consumir e, portanto, não permitirem que o código continue a rodar, prendendo-o nessa situação.

d) O que aconteceria se o semáforo `rec` fosse inicializado com mais de um sinal?

Nesse caso, ao rodar o código, aconteceria uma situação de **condição de corrida** pelo fato de que, com mais de um sinal, *threads* de tipos diferentes conseguiriam acessar o recurso ao mesmo tempo.

```
//globais
int a=0, b=0, c=0; //numero de threads A, B e C usando o recurso,
respectivamente
sem_t emA, emB, emC; //semaforos para exclusao mutua
sem_t rec; //semaforo para sincronizacao logica
//inicializacoes que devem ser feitas na main() antes da criacao das
threads
sem_init(&emA, 0, 1);
sem_init(&emB, 0, 1);
sem_init(&emC, 0, 1);
sem_init(&rec, 0, 1);

//funcao executada pelas As
void *A () {
      while(1) {
            sem_wait(&emA);
            a++;
            if(a==1) {
                  sem_wait(&rec);
            }
            sem_post(&emA);
            //SC: usa o recurso
            sem_wait(&emA);
            a--;
            if(a==0) sem_post(&rec);
            sem_post(&emA);
      }

//funcao executada pelas Bs
void *B () {
      while(1) {
            sem_wait(&emB);
            b++
            if(b==1) {
                  sem_wait(&rec);
            }
            sem_post(&emB);
            //SC: usa o recurso
```

```
            sem_wait(&emB);
            b--;
            if(b==0) sem_post(&rec);
            sem_post(&emB);
      }
}

//funcao executada pelas Cs
void *C () {
      while(1) {
            sem_wait(&emC);
            c++
            if(c==1) {
                  sem_wait(&rec);
            }
            sem_post(&emC);
            //SC: usa o recurso
            sem_wait(&emC);
            c--;
            if(c==0) sem_post(&rec);
            sem_post(&emC);
      }
}
```

Questão 02 (2,5 pontos) O código abaixo implementa uma solução para o problema do produtor/consumidor, considerando um *buffer* de tamanho ilimitado. Em uma execução com apenas um produtor (*thread* que executa a função `prod`) e um consumidor (*thread* que executa a função `cons`), ocorreu do consumidor retirar um item que não existia no *buffer*.

Considerando o que foi exposto, examine o código e responda, **justificando todas as respostas** (as justificativas contam na avaliação da questão):

a) Onde esta(ão) o(s) erro(s) no código?

O erro no código se encontra na linha 11, "`if(n==0) sem_wait(&d)`", da função "`void *cons(void *args)`" por conta da variável `n` ser global e não ser protegida por `mutex`, permitindo que a função "`void *prod(void *args)`" altere o valor dela antes que a função consumidora possa executar e entrar na linha 11, ocasionando, um acúmulo de sinais no semáforo `d`, que fará com que a consumidora possa retirar mesmo não havendo itens no *buffer*.

b) Proponha uma maneira de resolvê-lo(s) mantendo as funções produz item e consome item fora da exclusão mútua.

Acesse o código comentado dessa questão aqui: https://github.com/iagorafaellm/Concurrent_Computing_2021.1/blob/main/lista03/exercise02.c

```
1:int n=0; sem_t s, d; //s inicializado com 1 e d inicializado com 0
2:void *cons(void *args) {                  void *prod(void *args) {
3:   int item;                                 int item;
4:   sem_wait(&d);                             while(1) {
5:   while(1) {                                  produz_item(&item);
6:     sem_wait(&s);                             sem_wait(&s);
7:     retira_item(&item);                       insere_item(item);
8:     n--;                                      n++;
9:     sem_post(&s);                             if(n==1) sem_post(&d);
10:    consome_item(item);                       sem_post(&s);
11:    if(n==0) sem_wait(&d);                  }
12: }}                                       }
```

Questão 03 (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 o problema conhecido como *starvation*).

Considerando o exposto acima, sua tarefa é:

a) Implementar uma solução concorrente em C (apenas o código das *threads*) 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 (use apenas semáforos como mecanismo de sincronização).

o/

b) Escrever argumentos que demonstram a corretude do seu código (primeiro como ele funciona; depois de que forma atende os requisitos colocados, não gera condições de corrida indesejadas e não leva à aplicação a uma condição de *deadlock*).

o/

Questão 04 (2,5 pontos) O código abaixo[1] propõe uma implementação de variáveis de condição e suas operações básicas *wait*, *notify* e *notifyAll* fazendo uso de semáforos. A semântica dessa implementação é similar àquela que vimos em Java: o objeto de *lock* e a variável de condição são os mesmos e estão implícitos, ou seja, não são passados como argumentos para as funções. A operação *wait* deve liberar o *lock* atualmente detido pela *thread* e bloquear essa *thread*. A operação *notify* deve verificar se há alguma *thread* bloqueada na variável de condição implícita e desbloqueá-la. A operação *notifyAll* deve verificar se há *threads* bloqueadas na variável de condição implícita e desbloquear todas elas.

Considerando o que foi exposto, examine esse algoritmo e responda, **justificando todas as respostas** (as justificativas contam na avaliação da questão):

a) Qual é a finalidade dos semáforos `s`, `x` e `h` dentro do código? Esses semáforos foram inicializados corretamente?

Sim. O semáforo `s` tem como finalidade assegurar que novas *threads* não acabem sendo executadas antes das *threads* que já estavam esperando na fila de espera. Já o semáforo `x` tem como finalidade assegurar que duas *threads* não acessem simultaneamente um recurso compartilhado. E, por último, o semáforo `h` assegura que as threads permaneçam na pilha de *threads* a serem executadas.

b) Essa implementação está correta? Ela garante que a semântica das operações *wait*, *notify* e *notifyAll* está sendo atendida plenamente?

Sim, pois a semântica da operação *wait* tem sucesso ao liberar o *lock* da *thread* e a bloquear; a semântica da operação *notify* atende plenamente sua finalidade de assegurar se há *threads* bloqueadas na variável de condição implícita e a desbloquear; a semântica da operação *notifyAll* verifica se há *threads* bloqueadas na variável de condição implícita e desbloqueia todas elas de forma satisfatória.

c) Existe a possibilidade de acúmulo indevido de sinais nos semáforos `s`, `x` e `h`?

Não, pois todos os semáforos dependem da variável `aux` que se encontra dentro da **exclusão mútua**, não havendo possibilidade de duas ou mais *threads* acessarem simultaneamente o recurso compartilhado, não havendo, assim, um acúmulo de sinais.

```
//variaveis internas
sem_t s, x, h;
int aux = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

//inicializacoes feitas na funcao principal
sem_init(&s,0,0);
sem_init(&x,0,1);
sem_init(&h,0,0);
```

```
void wait() {
      //pre-condicao: a thread corrente detem o lock de 'm'
      sem_wait(&x);
      aux++;
      sem_post(&x);
      pthread_mutex_unlock(&m);
      sem_wait(&h);
      sem_post(&s);
      pthread_mutex_lock(&m);
}

void notify() {
      sem_wait(&x);
      if (aux > 0) {
            aux--;
            sem_post(&h);
            sem_wait(&s);
      }
      sem_post(&x)
}

void notifyAll() {
      sem_wait(&x);
      for (int i = 0; i < aux; i++)
            sem_post(&h);
      while (aux > 0) {
            aux--;
            sem_wait(&s);
      }
      sem_post(&x);
}
```

[1]Adaptado de Birrell, Andrew D. "Implementing condition variables with semaphores." Computer Systems. Springer, New York, NY, 2004. 29-37.