Módulo 3 - Lista de Exercícios 3 (2021/1 REMOTO)

Computação Concorrente (MAB-117)
Prof. Silvana Rossetto

[1]Instituto de Computação/UFRJ — 04 de outubro de 2021

Questão 1 (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*?
(b) Essa solução pode levar as threads a um estado de *starvation* (espera por um tempo muito longo para serem atendidas)?
(c) O que aconteceria se o semáforo rec fosse inicializado com 0 sinais?
(d) O que aconteceria se o semáforo rec fosse inicializado com mais de um sinal?

```
//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 2 (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 está(ão) o(s) erro(s) no código?

(b) Proponha uma maneira de resolvê-lo(s) mantendo as funções `produz_item` e `consome_item` fora da exclusão mútua.

```
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 3 (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).

(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 a aplicação a uma condição de *deadlock*).

Questão 4 (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?

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

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

```
//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.