Lista de Exercícios - 3

Programação Concorrente (ICP-361) - 2023-2
Prof. Silvana Rossetto

[1]IC/CCMN/UFRJ
27 de setembro de 2023

Questão 1 A função abaixo **void barreira(int nthreads)** implementa uma solução de sincronização coletiva (barreira). O parâmetro `nthreads` informa o número total de threads que devem participar da barreira. Assume-se que todas as threads invocarão essa função no ponto do código onde a sincronização por barreira é requerida. Ocorreu em uma execução do programa que usa essa função para implementar mais de um ponto de sincronização por barreira, de uma das threads passar pela segunda barreira antes de todas as threads terem chegado até ela. Responda, justificando: (a) Por que isso aconteceu? Onde está o erro na função? (b) Como ele pode ser corrigido?

```
1 sem_t mutex; //exclusao mutua (iniciado com 1)
2 sem_t cond; //condicional (iniciado com 0)
3 int chegaram=0; //variavel de estado global
4 void barreira(int numThreads) {
5    sem_wait(&mutex);
6    chegaram++;
7    if (chegaram < numThreads) {
8      sem_post(&mutex);
9      sem_wait(&cond);
10   } else {
11     for(int i=1; i<numThreads; i++)
12        { sem_post(&cond); }
13     chegaram = 0;
14     sem_post(&mutex);
15    }
16}
```

Questão 2 O código abaixo implementa uma solução para o padrão **leitores/escritores** (código das threads leitoras e escritoras). Lembrando os requisitos do padrão: (i) os leitores podem ler simultaneamente uma região de dados compartilhada, mas apenas um escritor pode escrever a cada instante; (ii) se um escritor está escrevendo, nenhum leitor pode ler a mesma região de dados compartilhada. Responda, justificando: (a) Essa solução está correta? Ela atende a todos os requisitos do problema?

```
1 int leitores=0; //leitores lendo
2 sem_t mutex, escrita; //inicializados com 1 sinal cada
3 void leitor(){
4   sem_wait(&mutex);
5   leitores++;
6   if(leitores==1) //primeiro leitor
7      { sem_wait(&escrita); }
8   sem_post(&mutex);
9   /* faz a leitura .... */
10  sem_wait(&mutex);
11  leitores--;
12  if(leitores==0) //ultimo leitor
13     { sem_post(&escrita); }
14  sem_post(&mutex);
15 }

16 void escritor(){
17   sem_wait(&escrita);
18   /* faz a escrita .... */
19   sem_post(&escrita);
20 }
```

Questão 3 Implemente uma solução para o padrão **produtor/consumidor** (implementar as funções $insere$ e $retira$ e parâmetros globais), usando **variáveis de condição e locks em C**. Lembrando, os requisitos do padrão são: (i) os produtores não podem inserir novos elementos quando a área de dados já está cheia; (ii) os consumidores não podem retirar elementos quando a área de dados já está vazia; (iii) os elementos devem ser retirados na mesma ordem em que foram inseridos; (iv) os elementos inseridos não podem ser perdidos (sobreescritos por novos elementos); (v) um elemento só pode ser retirado uma vez por um consumidor.

Questão 4 Implemente uma solução para o padrão **produtor/consumidor** (implementar as funções $insere$ e $retira$ e parâmetros globais) com a seguinte variação do problema: a cada execução de um **consumidor**, ele deve consumir o buffer inteiro, e não apenas um único item (para isso ele deve esperar o buffer ficar completamente cheio). O produtor segue a lógica convencional, isto é, insere um item de cada vez. A aplicação poderá ter mais de um produtor e mais de um consumidor. O uso eficiente dos mecanismos de sincronização será considerado na avaliação. **Use a linguagem C com semáforos.**

Questão 5 O código abaixo implementa a lógica central de operação de dois tipos de threads: *foo* e *bar*. Descubra qual é a condição para elas executarem a linha `//SC: usa o recurso`. Justifique sua resposta.

```
//globais
int a=0, b=0; //numero de threads foo e bar usando o recurso, respectivamente
sem_t emA, emB; //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(&rec, 0, 1);

void *foo () {
  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);
}
void *bar () {
  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);
  }
}
```

Questão 6 Considere uma **aplicação concorrente em C** com N threads que faz acesso de leitura e escrita em uma base de dados compartilhada pelas threads. O programa abaixo implementa as funções que as threads deverão executar ANTES e DEPOIS de fazerem as operações de leitura e escrita nessa base de dados, de forma a garantir os seguintes requisitos: (i) mais de uma operação de leitura pode ocorrer ao mesmo tempo; (ii) apenas uma operação de escrita pode ocorrer de cada vez; (iii) nenhuma operação de leitura pode ocorrer enquanto uma operação de escrita está sendo realizada; (iv) as operações de escrita devem ser priorizadas (novas operações de leitura só podem ser iniciadas quando não houver mais operações de escrita em espera). Avalie o programa apresentado e responda as questões abaixo justificando suas respostas: **(a)** Os requisitos (i) e (ii) são atendidos no programa? **(b)** O requisito (iv) é atendido no programa? **(c)** Há possibilidade de ocorrência de *starvation* (inanição) em alguma execução deste programa?

```
sem_t em_e, em_l, escr, leit; //semaforos iniciados com 1 sinal
int e=0, l=0;

void AntesLeitura() {
   sem_wait(&leit);
   sem_wait(&em_l);
   l++;
   if(l==1) sem_wait(&escr);
   sem_post(&em_l);
   sem_post(&leit);
}

void DepoisLeitura() {
   sem_wait(&em_l);
   l--;
   if(l==0) sem_post(&escr);
   sem_post(&em_l);
}

void AntesEscrita() {
   sem_wait(&em_e);
   e++;
   if(e==1) sem_wait(&leit);
   sem_post(&em_e);
   sem_wait(&escr);
}

void DepoisEscrita() {
   sem_post(&escr);
   sem_wait(&em_e);
   e--;
   if(e==0) sem_post(&leit);
   sem_post(&em_e);
}
```

Questão 7 O código abaixo implementa uma aplicação concorrente com duas threads (T0 e T1) as quais executam um trecho de código que requer exclusão mútua (linha 7). As linhas 3 a 6 implementam a lógica para entrada na seção crítica do código sem fazer uso de mecanismos de bloqueio. A linha 8 implementa o código de saída da seção crítica. Responda: (a) O que irá acontecer se as duas threads tentarem acessar a seção crítica ao mesmo tempo? (b) O que irá acontecer se uma das threads tentar acessar sozinha a seção crítica por várias vezes seguidas? Ela sofrerá alguma forma de contenção nesse acesso? (c) O que acontecerá se uma thread tentar acessar a seção crítica quando a outra thread já estiver acessando e esta mesma thread (a que está na seção crítica), quando sair da seção crítica, tentar acessá-la novamente antes da thread que está esperando ganhar a CPU novamente? (d) O código proposto garante exclusão mútua no acesso à seção crítica? **Justifique suas respostas**.

```
//variaveis globais
int querEntrar0 = 0; querEntrar1 = 0; int turno;

1:void *T0 (void *args) {        | void *T1 (void *args) {
2:  while(1) {                   |   while(1) {
3:    querEntrar0 = 1;           |     querEntrar1 = 1;
4:    turno = 1;                 |     turno = 0;
5:    while((querEntrar1==1) &&  |     while((querEntrar0==1) &&
6:       (turno==1))  { ; }      |        (turno==0)) { ; }
7:    //..seção crítica          |     //..seção crítica
8:    querEntrar0 = 0;           |     querEntrar1 = 0;
9:    //..fora da seção crítica  |     //..fora da seção crítica
10: }}                           |   }}
```

Questão 8 No código mostrado abaixo, a thread B foi programada para imprimir o valor de X quando ele for divisível por 10. Em uma execução com *dez* threads A e *uma* thread B, o valor *11* foi incorretamente impresso. (a) O que aconteceu? (b) Como o código pode ser corrigido para que essa situação não se repita? Justifique suas respostas.

```
1 int x = 0; pthread_mutex_t x_mutex; pthread_cond_t x_cond;
2 void *A (void *tid {
3    for (int i=0; i<100; i++) {
4      pthread_mutex_lock(&x_mutex);
5      x++;
6      if ((x%10) == 0) { pthread_cond_signal(&x_cond); }
7      pthread_mutex_unlock(&x_mutex);
8 } }
9 void *B (void *tid) {
10   pthread_mutex_lock(&x_mutex);
11   if ((x%10) != 0) { pthread_cond_wait(&x_cond, &x_mutex); }
12   printf("X=%d\n", x);
13   pthread_mutex_unlock(&x_mutex);
14 }
```

Questão 9 Considere uma aplicação onde três (3) threads são disparadas para executar a função abaixo. A variável $saldo$ é definida no escopo global com valor inicial igual a 0. Depois de todas as threads concluirem sua execução, o valor de $saldo$ é impresso na saída padrão pelo fluxo principal da aplicação. Responda as questões abaixo, justificando suas respostas: **(a)** Pode ocorrer de um valor acima de 600 ser impresso na saída padrão? **(b)** Pode ocorrer do valor 100 ser impresso na saída padrão? **(c)** Pode ocorrer do valor 200 ser impresso na saída padrão? **(d)** Pode ocorrer do valor 500 ser impresso na saída padrão?

```
float saldo = 0;
void* tarefa (void* arg) {
   for(int i=0; i<2; i++)
      saldo = saldo + 100;
}
```

Questão 10 Considere um programa concorrente com N threads que executam a função $tarefa$ mostrada abaixo e responda, **justificando**: **(a)** Pode ocorrer do valor de $saldo$ ficar negativo? **(b)** Se sim, corrija a implementação para que isso não ocorra. Se não, acrescente uma função de incremento do $saldo$.

```
float saldo=100.0; pthread_mutex_t l;
void retira (float val){       float le() {                  void* tarefa(void* arg) {
  pthread_mutex_lock(&l);        pthread_mutex_lock(&l);       float val = *(float *) arg;
  saldo = saldo-val;             float s = saldo;              float meuSaldo = le();
  pthread_mutex_unlock(&l;       pthread_mutex_unlock(&l);     if(meuSaldo >= val) retira(val);
}                                return s; }                   }
```