Primeira Lista de Exercícios

Programação Concorrente (ICP-361) - 2024-2
Prof. Silvana Rossetto

[1]IC/CCMN/UFRJ
18 de setembro de 2024

Questão 1 (a) Escreva uma função em C para calcular o valor de pi usando a fórmula de Bailey-Borwein-Plouffe mostrada abaixo. A função deve receber como entrada o valor de n, indicando que os n primeiros termos da série deverão ser considerados.

$$\pi = \sum_{k=0}^{\infty} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \frac{1}{16^k}$$

(b) Agora escreva uma versão concorrente dessa função (que será executada por M threads, dividindo a tarefa em subtarefas), com balanceamento de carga entre as threads.

Questão 2 Responda as questões abaixo:

(a) O que é *seção crítica* do código em um programa concorrente?
(b) O que é *corrida de dados* em um programa concorrente?
(c) O que é *violação de atomicidade* em um programa concorrente?
(d) O que é *violação de ordem* em um programa concorrente?
(e) Como funciona a sincronização por exclusão mútua com bloqueio (que usa *locks*)?
(f) Como funciona a sincronização condicional com bloqueio (que usa as funções *wait, signal e broadcast*)?
(g) Por que mecanismos de comunicação e sincronização são necessários para a programação concorrente?

Questão 3 Em um trabalho de Shan Lu et al. [1] são apresentados *bugs* de concorrência encontrados em aplicações reais (MySQL, Apache, Mozilla and OpenOffice). Dois deles estão transcritos abaixo. Proponha uma solução para cada um deles.

Caso 1 (bug de violação de atomicidade no MySQL): Nesse caso temos duas threads (Thread 1 e Thread 2). Como nós programadores estamos mais acostumados a pensar de forma sequencial, temos a tendência de assumir que pequenos trechos de código serão executados de forma atômica. Os programadores assumiram nesse caso que se o valor avaliado na sentença 1 (S1) é diferente de NULL, então esse mesmo valor será usado na sentença 2 (S2). Entretanto, pode ocorrer em uma execução qualquer que a sentença 3 (S3) quebre essa premissa de atomicidade, causando um erro na aplicação. (a) Mostre qual ordem de execução das sentenças vai gerar o erro. (b) Proponha uma correção no código para evitar esse erro.

```
Thread 1:                          Thread 2:
S1: if (thd->proc_info) {          S3: thd->proc_info=NULL;
  S2: fputs(thd->proc_info,...);   ...
}
```

[1]LU, Shan et al. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. 2008. p. 329-339.

Caso 2 (bug de violação de ordem no Mozilla): Nesse caso também temos duas threads (Thread 1 e Thread 2). A thread 2 só deveria acessar a variável `mThread` depois dela ser devidamente inicializada. **(c)** Proponha uma correção no código para garantir que essa condição seja sempre satisfeita.

```
Thread 1:                                  Thread 2:
void init (...) {                          void mMain(...) {
  mThread=PR_CreateThread(mMain,...);        mState=mThread->State;
   ...                                       ...
}                                          }
```

Questão 4 Uma aplicação dispara três threads (T1, T2 e T3) para execução (códigos mostrados abaixo). Verifique se os valores $1, -1, 0, 2, -2, 3, -3, 4, -4$ podem ser impressos na saída padrão quando essa aplicação é executada. Em caso afirmativo, mostre uma sequência de execução das threads que gere o valor correspondente.

```
int x=0; //variavel global
(0)      T1:                  T2:             T3:
(1)      x = x-1;             x = x+1;        x = x+1;
(2)      x = x+1;             x = x-1;        if(x == 1)
(3)      x = x-1;                                printf("%d",x);
(4)      if (x == -1)
(5)         printf("%d",x);
(6)
```

Questão 5 Considere um programa concorrente com N threads que executam a função $tarefa$ mostrada abaixo. Pode ocorrer do valor de $saldo$ ficar negativo? Justifique sua resposta.

```
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; }                   }
```

Questão 6 O código abaixo implementa o padrão **leitores/escritores** usando variáveis de condição em C. **Responda as questões abaixo, justificando suas respostas**: **(a)** Quais requisitos lógicos do padrão estão sendo atendidos e de que forma? **(b)** Os blocos `while` poderiam ser substituídos por blocos `if`?

```
int leit=0; //contador de threads lendo
int escr=0; //contador de threads escrevendo
pthread_mutex_t mutex;    pthread_cond_t cond_leit, cond_escr;

//entrada leitura                                 !  //saida leitura
void InicLeit() {                                 !  void FimLeit() {
   pthread_mutex_lock(&mutex);                    !     pthread_mutex_lock(&mutex);
   while(escr > 0)                                !     leit--;
      pthread_cond_wait(&cond_leit, &mutex);      !     if(leit==0)
   leit++;                                        !        pthread_cond_signal(&cond_escr);
   pthread_mutex_unlock(&mutex);                  !     pthread_mutex_unlock(&mutex);
}                                                 !  }

//entrada escrita                                 !  //saida escrita
void InicEscr() {                                 !  void FimEscr() {
   pthread_mutex_lock(&mutex);                    !     pthread_mutex_lock(&mutex);
   while((leit>0) || (escr>0)) {                  !     escr--;
     pthread_cond_wait(&cond_escr, &mutex);       !     pthread_cond_signal(&cond_escr);
   }                                              !     pthread_cond_broadcast(&cond_leit);
   escr++;                                        !     pthread_mutex_unlock(&mutex);
   pthread_mutex_unlock(&mutex);                  !  }
}                                                 !
```

Questão 7 Implemente uma solução concorrente em C para o problema dos **produtores e consumidores** (implementar as funções $insere$ e $retira$ e parâmetros globais) com a seguinte variação do problema: a cada execução de um **produtor**, ele deve preencher o buffer inteiro, e não apenas um único item (para isso ele deve esperar o buffer ficar completamente vazio). O consumidor segue a lógica convencional, isto é, insere um item de cada vez. A aplicação poderá ter mais de uma thread produtora e mais de uma thread consumidora. Use variáveis de condição e locks para implementar os requisitos de sincronização.

Questão 8 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** (solução de Peterson[2]). 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?
(e) As threads podem entrar em estado de *deadlock*, *starvation* ou *livelock*?

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 9 Escreva um programa concorrente em C com duas threads que implementam o seguinte diálogo:

```
Thread1: olá, você está acessando a variável 'aux' agora?
Thread2: oi, não, não estou
Thread1: certo, então vou alterá-la, tá?
Thread2: tudo bem
Thread1: terminei a alteração da variável 'aux'
Thread2: perfeito, recebido!
```

Questão 10 Considere um progama que processa requisições feitas a uma base de dados. O programa recebe uma sequência finita de requisições e as processa uma a uma. O tratamento de uma requisição envolve: ler dados de entrada consultando a base de dados, processar esses dados, verificar se deve ou não escrever o resultado de volta na base de dados. Dadas as tarefas elementares desse problema: *ler dado da base*, *processar dado* e *escrever dado na base*, **como esse problema poderia se beneficiar de uma solução concorrente?**

[2]G.L. Peterson, "Myths about the mutual exclusion problem", Information Processing Letters, vol. 12, no. 3, pp. 115-116, 1981.