Lista de Exercícios - 2

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

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

Questão 1 Responda as questões abaixo, justificando todas as respostas:

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

Questão 2 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 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:
void init (...) {
  mThread=PR_CreateThread(mMain,...);
   ...
}
```

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

Questão 4 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
void InicLeit() {
   pthread_mutex_lock(&mutex);
   while(escr > 0)
      pthread_cond_wait(&cond_leit, &mutex);
   leit++;
   pthread_mutex_unlock(&mutex);
}

//entrada escrita
void InicEscr() {
   pthread_mutex_lock(&mutex);
   while((leit>0) || (escr>0)) {
     pthread_cond_wait(&cond_escr, &mutex);
   }
   escr++;
   pthread_mutex_unlock(&mutex);
}
```

```
//saida leitura
void FimLeit() {
   pthread_mutex_lock(&mutex);
   leit--;
   if(leit==0)
      pthread_cond_signal(&cond_escr);
   pthread_mutex_unlock(&mutex);
}

//saida escrita
void FimEscr() {
   pthread_mutex_lock(&mutex);
   escr--;
   pthread_cond_signal(&cond_escr);
   pthread_cond_broadcast(&cond_leit);
   pthread_mutex_unlock(&mutex);
}
```

Questão 5 O probrama abaixo foi implementado por um colega em uma atividade de laboratório de uma edição anterior da disciplina. O roteiro foi o seguinte: *Implemente um programa com 5 threads:*

```
A thread 1 imprime a frase \Oi Maria!";
A thread 2 imprime a frase \Oi José!";
A thread 3 imprime a frase \Sente-se por favor.";
A thread 4 imprime a frase \Até mais José!";
A thread 5 imprime a frase \Até mais Maria!".
```

As threads devem ser criadas todas de uma vez na função main. As regras de impressão das mensagens (execução das threads) serão: (i) A ordem em que as threads 1 e 2 imprimem suas mensagens não importa, mas ambas devem sempre imprimir suas mensagens antes da thread 3. (ii) A ordem em que as threads 4 e 5 imprimem suas mensagens não importa, mas ambas devem sempre imprimir suas mensagens depois da thread 3. **Responda as perguntas abaixo, justificando suas respostas:** (a) A solução do colega atende aos requisitos colocados? (b) Os blocos `while` da funções `permanencia` e `saida` poderiam ser substituídos por blocos `if`?

```
//variaveis globais
int chegadas = 0;
int sentados = 0;
pthread_mutex_t x_mutex;
pthread_cond_t chegada_cond;
pthread_cond_t sentado_cond;
```

```
void *chegada(void *arg) {
    int thread_id = *(int*)arg;
    if (thread_id == 0)
        printf("Oi José!\n");
    else
        printf("Oi Maria!\n");
    pthread_mutex_lock(&x_mutex);
    chegadas++;
    if (chegadas == 2) {
        pthread_cond_signal(&chegada_cond);
    }
    pthread_mutex_unlock(&x_mutex);
}

void *permanencia(void *arg) {
    pthread_mutex_lock(&x_mutex);
    while (chegadas != 2)
       pthread_cond_wait(&chegada_cond, &x_mutex);
    printf("Sentem-se por favor.\n");
    sentados++;
    pthread_cond_broadcast(&sentado_cond);
    pthread_mutex_unlock(&x_mutex);
}

void *saida(void *arg) {
    pthread_mutex_lock(&x_mutex);
    while(sentados != 1)
       pthread_cond_wait(&sentado_cond, &x_mutex);
    pthread_mutex_unlock(&x_mutex);
    int thread_id = *(int*)arg;
    if (thread_id == 3) {
        printf("Tchau José!\n");
    else
        printf("Tchau Maria!\n");
    }
}

int main(int argc, char *argv[]) {
  pthread_t threads[NTHREADS];

  pthread_mutex_init(&x_mutex, NULL);
  pthread_cond_init (&chegada_cond, NULL);
  pthread_cond_init (&sentado_cond, NULL);

  int thread_ids[NTHREADS] = {0, 1, 2, 3, 4};

  pthread_create(&threads[0], NULL, chegada, &thread_ids[0]);
  pthread_create(&threads[1], NULL, chegada, &thread_ids[1]);
  pthread_create(&threads[2], NULL, permanencia, &thread_ids[2]);
  pthread_create(&threads[3], NULL, saida, &thread_ids[3]);
  pthread_create(&threads[4], NULL, saida, &thread_ids[4]);

  pthread_exit(NULL);
  return 0;
}
```

Questão 6 (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 7 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*, **responda, justificando suas decisões ou escolhas**.

(a) Como esse problema poderia se beneficiar de uma solução concorrente?

(b) Projete uma solução concorrente para esse problema (algoritmo que será executado por cada thread).