Primeiro Estudo Dirigido

Programação Concorrente (ICP-361) - 2025-2
Prof. Silvana Rossetto

[1]IC/CCMN/UFRJ
1 de setembro de 2025

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 Considere o problema de encontrar o **valor mínimo** e o **valor máximo** em um vetor de inteiros com **N** ($N > 1$) elementos. Pense como resolver esse problema usando programação concorrente, fazendo com que todas as **T** ($T > 1$) threads executem a mesma função e dividam entre si igualmente a carga de trabalho. Assuma que N é sempre divisível por M. Escreva em C **a função que as threads deverão executar**, minimizando os custos da concorrência. Acrescente na forma de comentários todas as informações que julgar necessárias para entender a sua proposta.

Questão 3 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* (ou violação de condição lógica) em um programa concorrente?
(e) Como funciona a sincronização por exclusão mútua com bloqueio (que usa *locks*)?
(f) Como funciona a sincronização por condição com bloqueio (que usa as funções *wait, signal e broadcast*)?
(g) Por que mecanismos de comunicação e sincronização são necessários para a programação concorrente?
(h) Como funciona a comunicação entre threads por memória compartilhada?

Questão 4 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.

[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.

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

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 5 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 6 Uma aplicação dispara (uma única vez) as três threads (T1, T2 e T3) mostrados abaixo. **Tarefa:** (a) Verifique se os valores $x = 0, x = -1, x = 2, x = 4$ 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. (b) Essa aplicação tem **corrida de dados**, **violação de atomicidade** e **violação de ordem**? **Justifique suas respostas.**

```
int x=0;
1 void *T1(void *arg){      | void *T2(void *arg){        | void *T3(void *arg){
2  int a=0;                 |   int a=2;                  |   x--;
3  while(a<2) {             |   while(a>0) {              |   x++;
4    x++;                   |     x++;                    | }
5    x--;                   |     x--;                    |
6    if(x==0)               |     if(x==0)                |
7     {printf("x=%d\n",x);} |       {printf("x=%d\n",x);} |
8    a++;                   |     a--;                    |
9    } }                    |   } }                       |
```

Questão 7 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 8 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 9 Considere o seguinte problema. Dado um vetor de inteiros, precisamos calcular as somas parciais em cada posição desse vetor (somatório de todos os elementos que antecedem essas posições, incluindo o elemento da própria posição). Por exemplo, dado o vetor $[1, 4, -1, 7]$ como entrada, o vetor resultante com as somas parciais em todas as posições é: $[1, 5, 4, 11]$. O primeiro elemento se mantém, o segundo elemento é a soma de $1 + 4$, o terceiro elemento é a soma de $1 + 4 + (-1)$, e o quarto elemento é a soma de $1 + 4 + (-1) + 7$. O algoritmo sequencial para resolver esse problema é bastante simples:

```
for(int i=1; i<n; i++)
   vetor[i] = vetor[i] + vetor[i-1];
```

Repare que o próprio vetor de entrada é modificado, isto é, usamos apenas um vetor que é alterado como saída do programa. Uma possível solução concorrente para esse problema é mostrada abaixo. Para essa solução, o número de threads criadas é igual ao número de elementos no vetor; e todas as threads executarão a mesma função ($tarefa$).

```
//variaveis para uso na funcao 'barreira'
int bloqueadas = 0; //qtde de threads bloqueadas
pthread_mutex_t x_mutex; //sincronizacao de exclusao mutua, inicializado na main
pthread_cond_t x_cond; //sincronizacao condicional, inicializado na main

int *vetor; //inicializado na main
//funcao barreira
void barreira(int nthreads) {
    pthread_mutex_lock(&x_mutex);
    if (bloqueadas == (nthreads-1)) {
      pthread_cond_broadcast(&x_cond);
      bloqueadas=0;
    } else {
      bloqueadas++;
      pthread_cond_wait(&x_cond, &x_mutex);
    }
    pthread_mutex_unlock(&x_mutex);
}
//funcao das threads
void *tarefa (void *arg) {
   int id = *(int*)arg; //identificador unico da thread
   int salto, aux; //variaveis auxiliares
   for(salto=1; salto<nthreads; salto*=2) {
      if(id >= salto) {
         aux = vetor[id-salto];
         barreira(nthreads-salto);
         vetor[id] = aux + vetor[id];
         barreira(nthreads-salto);
      } else break;
   }
   pthread_exit(NULL);
}
```

Tarefa: Descreva como o algoritmo concorrente funciona, respondendo as seguintes perguntas (justifique todas as respostas): (a) Como a tarefa foi dividida entre as threads? (b) Quando cada thread termina sua execução? (c) Há balanceamento de carga na divisão das tarefas entre as threads? (d) Qual é a finalidade das duas chamadas da função $barreira$?

Questão 10 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? (c) Reescreva os códigos das thread A e B para que B **necessariamente** imprima X quando seu valor for exatamente igual a 50. O programa pode disparar várias threads A, mas apenas uma thread B. **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);
   }
```

Questão 11 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?

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: }}                           |   }}
```

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