Programação Concorrente (ICP361)

Gabarito Primeira Prova (2025/2) — Profa. Silvana Rossetto

[1]IC/CCMN/UFRJ — 18 de setembro de 2025

Questão 1 (3,0 pts) Escolha **UM dos problemas** listados abaixo e escreva uma implementação concorrente em C usando as funções da biblioteca PThread: (i) contar a quantidade de números negativos em um vetor de inteiros; (ii) contar a quantidade de números ímpares em um vetor de inteiros; (iii) encontrar a média de todos os valores de um vetor de inteiros. Por simplicidade, considere que o **vetor** e seu **tamanho** já foram inicializados e estão no **escopo global**, assim como a **quantidade de threads**. Todas as threads deverão executar a mesma função e dividir a tarefa total entre elas de forma balanceada.

```
int *g_vet; //vetor
long int g_tam; //tamanho do vetor
short int g_nthreads; //qtde de threads

//funcao das threads - numeros impares
void* tarefa (void* args) {
   long int id = (long int) args; //recebe id da thread
   int fatia = g_tam/g_threads;
   int ini = id * fatia;
   int fim = ini + fatia;
   if(id == (g_nthreads-1)) fim = g_tam; //ultima thread vai ate o final
   long int impares = 0;
   for (int i=ini; i<fim; i++) {
      if(g_vet[i]%2) impares++;
   }
   pthread_exit((void*)impares); //retorno por valor
}

//funcao principal
int main(void) {
    int totalImpares = 0; //valor final
    long int aux; //recebe retorno parcial das threads
    //carrega e inicializa variaveis globais (...)
    pthread_t tid[g_nthreads]; //identificadores das threads no sistema
    //cria as threads
    for(long int i=0; i < g_nthreads; i++)
       if(pthread_create(&tid[i], NULL, tarefa, (void*) i)) return(1);
    for(int i=0; i < g_nthreads; i++) {
       if(pthread_join(tid[i], (void*) &aux)) return(2);
       totalImpares += aux;
    }
    printf("Total impares encontrados: %d\n", totalImpares);
    return 0;
}
```

Questão 2 (3,5 pts) O código abaixo implementa o padrão leitores/escritores com os seguintes requisitos: (i) os leitores podem ler simultaneamente a região de dados compartilhada; (ii) apenas um escritor pode escrever a cada instante na região de dados compartilhada; (iii) se um escritor está escrevendo, nenhum leitor pode estar lendo a região de dados compartilhada. Na aplicação que usa esse padrão, todas as threads leitoras chamam a função `AntesLeitura` antes de ler a região de dados compartilhada e a função `DepoisLeitura` após terminarem a leitura. E as threads escritoras só escrevem de dentro da função `Escreve`. (a) Essa implementação atende corretamente a todos os requisitos do padrão? **Justifique** considerando cada um dos requisitos colocados. (b) Foi necessário lidar com os problemas de **corrida de dados**, **violação de atomicidade** e **violação de ordem**? Justifique. Descreva quais mecanismos de sincronização foram usados para tratá-los.

Resp.: Na linha 21, considerar 'cond_escr' como 'escrita'.

(a) Sim, atende a todos os requisitos. Os leitores só completam a chamada da função 'AntesLeitura' se conseguirem o lock 'mutex', e este permanece alocado para um escritor enquanto realiza a escrita. Se um leitor consegue completar a função 'AntesLeitura', outros leitores também o podem fazer, uma vez que o lock 'mutex' é liberado no final dessa função. Desse modo, mais de um leitor pode ler ao mesmo tempo. O escritor só avança para a linha 20 (onde escreve) se não houver leitor lendo, uma vez que a variável de estado 'leitores' é verificada corretamente, e atualizada nas funções de entrada e saída para leitura. Apenas um escritor consegue escrever de cada vez pois a escrita está dentro de uma bloco atômico, o que impede também a entrada de novos leitores enquanto a escrita ocorre pois o lock 'mutex' permanece alocado.

(b) Sim, foi necessário lidar com os três problemas.

(i) corrida de dados: foi necessário usar uma variável de estado 'leitores' para registrar quantos leitores estão ativos lendo, e o acesso a essa variável para incrementá-la e decrementá-la precisou ser feito com exclusão mútua, por meio da variável de lock 'mutex' para evitar o problema de corrida de dados.

(ii) violação de atomicidade: foi necessário verificar o estado da aplicação armazenado na variável 'leitores' e tomar uma ação em seguida. No primeiro caso, na função 'DepoisLeitura', linhas 12 e 13, um leitor sinaliza um escritor apenas de não houver mais leitores ativos. No segundo caso, na função 'Escreve', um escritor deve se bloquear apenas se houver leitor ativo (linhas 18 e 19). Em ambos os casos foi usado o mecanismo de exclusão mútua com o lock 'mutex'.

(iii) violação de ordem: foi necessário bloquear o escritor no caso dele tentar escrever quando já houver leitor ativo (linha 19) e sinalizá-lo quando todos os leitores ativos concluirem a leitura (linha 13). Para isso uma variável de condição foi usada ('escrita'). Também foi necessário bloquear o leitor no caso dele tentar ler quando já houver um escritor escrevendo. Isso foi feito no código usando um lock de exclusão mútua ('mutex'), uma vez que ele é mantido pelo escritor enquanto o mesmo realiza a escrita.

```
01 short int leitores=0;
02 pthread_mutex_t mutex; //inicializada na main
03 pthread_cond_t escrita; //inicializada na main
04 void AntesLeitura() {
05   pthread_mutex_lock(&mutex);
06   leitores++;
07   pthread_mutex_unlock(&mutex);
08 }
09 void DepoisLeitura () {
10   pthread_mutex_lock(&mutex);
11   leitores--;
12   if(leitores==0)
13      { pthread_cond_signal(&escrita); }
14   pthread_mutex_unlock(&mutex);
15 }
16 void Escreve(void * args) {
17   pthread_mutex_lock(&mutex);
18   while(leitores > 0)
19      { pthread_cond_wait(&escrita, &mutex); }
20   //realiza a escrita de args (...)
21   pthread_cond_signal(&cond_escr);
22   pthread_mutex_unlock(&mutex);
23 }
```

Questão 3 (3,5 pts) Uma aplicação que processa requisições precisará passar por uma auditoria interna. Para isso, a cada 10000 requisições atendidas, uma cópia do banco de dados precisará ser feita (chamar a função pré-definida `backup()`). Enquanto a cópia é feita, não deverá ocorrer processamento de novas requisições. Você foi chamado para implementar uma versão concorrente da aplicação: (a) escreva o novo fluxo de execução (*thread*) responsável por executar o método de cópia; e (b) altere o fluxo de execução principal de modo que os dois fluxos interajam apropriadamente. (não é necessário criar a nova thread, apenas implementar os trechos de interação entre os fluxos). **Minimize os custos da concorrência**.

```
//variaveis globais inicializadas na main
pthread_cond_t principal, auxiliar;
pthread_mutex_t mutex;
char estado = 0; //variavel de estado (1: fazer backup; 0: processar requisicao)

void main () {
   while(1) {
      for(int i=0; i<10000; i++) {
         TRec rec = recebeRequisição();
         processa(rec);
      }
      pthread_mutex_lock(&mutex);
      estado = 1;
      pthread_cond_signal(&auxiliar);
      pthread_cond_wait(&principal, &mutex);
      pthread_mutex_unlock(&mutex);
   }
}

void * fluxoAuxiliar (void *args) {
   while(1) {
      pthread_mutex_lock(&mutex);
      if(!estado)
         pthread_cond_wait(&auxiliar, &mutex);
      backup()
      estado = 0;
      pthread_mutex_unlock(&mutex);
      pthread_cond_signal(&principal);
   }
}
```