Exemplo

Uso de MUTEXES em *threads* com partilha de dados

Impacto na coerência dos dados e na performance

Existem dois cenários típicos onde o uso de threads é vantajoso: i) a necessidade de um programa efetuar várias tarefas em simultâneo, e ii) a execução de um algoritmo que é paralelizável e se pretende obter mais desempenho aproveitando a existência de diversos núcleos/processadores. Este exemplo encaixa na segunda situação.

Este exemplo pretende contar o número de valores pares entre 1 e um determinado limite especificado pelo utilizador. A estratégia implementada é sintética (artificial), não sendo a melhor em termos de lógica ou performance, mas tem por objetivo mostrar a ocorrência de *race condition* para evidenciar a necessidade de uso de mutexes.

Este exemplo:

- **Utiliza várias *threads* para contar o número de números pares.**
  - **O exemplo é propositadamente trivial de forma a focar acima de tudo:**
    - As consequências (neste caso, dados corrompidos) do acesso não protegido a dados partilhados entre threads
    - A identificação de secções críticas e o uso correto de MUTEXes para resolução dos problemas decorrentes de acesso concorrente a dados
    - As consequências na performance decorrentes do uso de MUTEXes

- **O programa conta o número de números pares existentes num intervalo de 1 a um valor N especificado pelo utilizador. O programa permite indicar, por linha de comandos**
  - O valor N
  - O número de *threads* a usar (1 ou mais). Se for 1, não existe concorrência
  - Usar mutexes (S/N)

Estratégia usada

- Existe uma variável com um contador de números pares encontrados até agora, e uma variável com o próximo número a analisar. Ambas as variáveis são acessíveis às threads, que as consultam e atualizam de forma concorrente
- A função *main* inicializa as variáveis descritas acima e lança *threads*. E no final aguarda pela sua terminação para apresentar os resultados
- Cada *thread* obtém o próximo número a analisar lendo a variável partilhada, incrementando-a para que as outras *threads* considerem o número seguinte e não o mesmo
- Sempre que uma *thread* encontra um número par, incrementa a variável partilhada com essa informação
- Cada *thread* tem também contadores privados do número de pares que encontrou
- Dependendo do uso ou não de mutexes, os resultados encontrados não irão ser coerentes, havendo vários cenários possíveis e cuja explicação foi analisada na aula

Importante:

- O exemplo é trivial propositadamente de forma a focar o assunto de mutexes e secções críticas e não propriamente assuntos periféricos de algoritmia.

- O uso de mutexes é linear e segue o API descrito noutro documento.

- O código contém duas versões da thread: uma sem mutexes e outra com mutexes, sendo executada aquela que foi a escolhida pelo utilizador.
    - Esta repetição de código tem como objetivo tornar cada uma das versões mais simples de ler.

- **IMPORTANTE:** A estratégia seguida não é a mais eficiente (uma logica melhor seria, obviamente, a divisão à partida do espaço de procura pelas várias *threads*, tornando-se mais rápido e com menos problemas de acesso concorrente a dados). A estratégia apresentada aqui foi escolhida propositadamente para mostrar problemas de acesso concorrente a dados.
- **IMPORTANRE:** em certas combinações de valores de configuração, o uso de *threads* piora a performance. Este resultado parece contraditório, mas deve-se apenas à extrema taxa de atualização de variáveis partilhadas, que causa a necessidade de renovar o conteúdo da cache dos núcleos do processador. Esta taxa elevadíssima de atualização de dados partilhados é artificial e propositada, não representa a realidade da generalidade dos programas: normalmente, quanto mais *threads*, melhor a performance.

O código é apresentado abaixo. Segue a estratégia habitual de programação com *threads*.

Headers necessários

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <unistd.h>
#include <locale.h>
#include <sys/time.h>
```

Protótipos das funções das threads

- **Duas versões: uma sem mutexes, e outra com mutexes. O algoritmo é o mesmo, apenas diferem no uso (ou não) de mutexes.**

```
void * identificaPares(void *);   // sem mutex

void * identificaPares_M(void *); // com mutex
```

Estrutura de dados para controlo das *threads* (lógica habitual + informação acerca dos mutexes)

```
typedef struct {
   pthread_t tid;
   int * current;    // <- aponta p/ variável da main
   int last;
   int found;
   int * allFound;   // <- aponta p/ variável da main

   pthread_mutex_t * pMutCurr;  // <-
   pthread_mutex_t * pMutFound; // <-
} TDADOS;
```

Significado de alguns campos da estrutura de dados de controlo das *threads*

```
// current  -> valor atual a ser analisado (atualz. p/ threads
// allFound -> total acumulado, atualizado pelas threads

// mutexes -> partilhados pelas threads
```

Função main

- **Analisa as escolhas do utilizador especificadas na linha de comandos**
- **Lança as *threads***
- **Aguarda que as *threads* terminem e colige os resultados**
- **No final, apresenta a duração do tempo de execução**

```
#define MAXIMUMT 20   // Num Maximo de threads

int main(int argc, char * argv[]) {
   int numThreads, last;
   TDADOS workers[MAXIMUMT];
   pthread_mutex_t mutCurr;  //  = PTHREAD_MUTEX_INITIALIZER;
   pthread_mutex_t mutFound; // = PTHREAD_MUTEX_INITIALIZER;

   int mainFound = 0;
   int current = 1;
   int allFound = 0;
   int useMutex = 0;

   struct timeval tvalBefore, tvalAfter;
   int execTime = 0;

   int i;

   if (argc<4) {
      printf("\n%s max threads useMutex(s/n)\n", argv[0]);
      return 1;
   }

   last = atoi(argv[1]);
   numThreads = atoi(argv[2]);
   useMutex = argv[3][0] == 's' ? 1 : 0;

   printf("\nAnalisar até = %d", last);
   printf("\nNum Threads = %d", numThreads);
   printf("\n--> Usar mutexes: %s\n", useMutex==1 ? "SIM" : "NAO");

   pthread_mutex_init(& mutCurr, NULL);    // inicializa mutexes mesmo
   pthread_mutex_init(& mutFound, NULL);   // que depois nao os use

   gettimeofday (&tvalBefore, NULL);


   for (i = 0; i< numThreads; i++) {
       workers[i].current = & current;
       workers[i].last = last;
       workers[i].pMutCurr = & mutCurr;
       workers[i].pMutFound = & mutFound;
       workers[i].allFound = & mainFound;
       if (useMutex)
           pthread_create(& workers[i].tid, NULL, identificaPares_M, workers + i);
       else
           pthread_create(& workers[i].tid, NULL, identificaPares, workers + i);
   }

   printf("\nThreads iniciadas\n");

   // aguarda ate todas chegarem ao limite estipulado (numero maximo)
   // neste caso isso significa aguardar que as threads terminem
```

```
   for (i=0; i<numThreads; ++i) {
      pthread_join(workers[i].tid, NULL);
      printf("\nThread %d -> encontrou = %d", i, workers[i].found);
      allFound += workers[i].found;
   }

   gettimeofday (&tvalAfter, NULL);
   execTime = tvalAfter.tv_sec - tvalBefore.tv_sec;

   setlocale(LC_NUMERIC, "");

   printf("\n\n Somatorio encontradas por todas as threads = %'d", allFound);
   printf("\n\n Total encontrados na var da main = %'d", mainFound);
   printf("\n\n Valor deveria ser = %'d", last/2);
   printf("\n\n Tempo de execução = %d segundos", execTime);
   printf("\n\n");
   return 0;
}
```

Função da(s) *thread*(s) – SEM mutexes

```
// sem mutex
void * identificaPares(void * p) {
   TDADOS * myDados = (TDADOS *) p;
   int i, myCurrent, myFound = 0;

   while (1) {  // nao é infinito pq tem uma condição de paragem

      // --- obtem prox numero ---
      myCurrent = *(myDados->current);

      // chegou ao fim? - se sim, sai
      if (myCurrent > myDados->last)
          break;

      // atualiza "current" p/ outras threads processam prox número
      ++(*(myDados->current));   // esta var é partilhada

      // --- verifica se é par ---
      if (myCurrent % 2 == 0) {
         // atualiza contador "privado" da thread"
         ++myFound;   // esta var nao é partilhada

         // atualiza contador (partilhado   )
         ++(*(myDados->allFound));  // esta var é partilhada
      }
   }

   // passa contador "local" para a estrutura desta thread
   // acessível à main (tb podia ter indo atualizando logo)
   myDados->found = myFound ;
   return NULL;
}
```

Função da(s) thread(s) – COM mutexes

```
// com mutex
void * identificaPares_M(void * p) {
   TDADOS * myDados = (TDADOS *) p;
   int i, myCurrent, myFound = 0;

   while (1) {  // nao é infinito pq tem uma condição de paragem
      // --- obtem prox numero ---

      // Seccao Critica  1
      pthread_mutex_lock(myDados->pMutCurr);
        myCurrent = *(myDados->current);
        if (myCurrent > myDados->last) {
            pthread_mutex_unlock(myDados->pMutCurr);
            break;
        }
        ++(*(myDados->current));
      pthread_mutex_unlock(myDados->pMutCurr);


      // --- verifica se é par ---
      if (myCurrent % 2 == 0) {
         ++myFound;   // esta var não é partilhada

         // Seccao Critica 2
         pthread_mutex_lock(myDados->pMutFound);
           ++(*(myDados->allFound));  // esta é partilhada
         pthread_mutex_unlock(myDados->pMutFound);
      }
   }
   myDados->found = myFound ;
   return NULL;
}
```

Experiências que se podem fazer com este exemplo

- Ver o funcionamento com e sem *mutexes* e perceber por que razão, sem mutexes os resultados não são nem corretos nem coerentes entre aquilo que as *threads* dizem que encontraram e aquilo que a função main coligiu.
- Ver o impacto em termos de performance causado pelo uso de *threads*, mesmo que seja no caso de apenas uma *thread*.