LABORATÓRIO DE PROGRAMAÇÃO AVANÇADA
DÉCIMO TRABALHO PRÁTICO
- - THREADS - -

Neste trabalho vamos exercitar **múltiplas threads** utilizando o Posix Threads (**Pthreads**) **necessariamente na linguagem de programação C**.

Com threads é possível gerar processos concorrentes, isto é, diversas tarefas que podem ser executadas em paralelo, usualmente são cooperativas, e que compartilham recursos, como memória por exemplo. É claro que é uma solução mais efetiva em arquiteturas de múltiplos processadores ou multi-cores onde o fluxo de processos pode ser escalonado para ser executado em outro processador (ou core) ganhando velocidade através de execuções paralelas. Threads (que são chamados de processos leves) têm muitas vantagens quando comparado com o "fork" de novos processos, onde o principal deles é que não precisa gerar todo um espaço de endereçamento, mas aproveita o do processo já criado. Nesse caso, as threads executam sob a política de memória compartilhada e que, tem-se que tomar cuidado para impor mecanismos de acesso mutuamente exclusivo ao mesmo recurso.

Abaixo segue uma breve (muito breve mesmo) introdução ao tema. Recomendo que se busque mais informações na Web. Os códigos abaixo foram tirados do Pthread Tutorial disponível no link: ***http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html***.

Criação e término de threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_msg( void *ptr );
main()
{
  pthread_t thr1, thr2;
  char *msg1 = "Inicio da Thread 1";
  char *msg2 = "Inicio da Thread 2";
  int iret1, iret2;
  // Cria threads independents, mas executa a mesma funcao
  iret1 = pthread_create(&thr1,NULL,print_msg,(void*) msg1);
  iret2 = pthread_create(&thr2,NULL,print_msg,(void*) msg2);

  // IMPORTANTE!
  // Espera ate que as threads se completem
  // Se nao esperar, corre-se o risco de sair e terminar
  // o processo antes que todas as threads finalizem
  pthread_join( thr1, NULL);
  pthread_join( thr2, NULL);
  printf("Thread 1 retorna: %d\n",iret1);
  printf("Thread 2 retorna: %d\n",iret2);
  exit(0);
}
```

```
void *print_msg( void *ptr )
{
  char *message;  message = (char *) ptr;
  printf("%s \n", message);
}
```

Se o arquivo for salvo com o nome "**thread1.c**" a forma de compilação é:

```
gcc -lpthread thread1.c -o thread1
```

onde "**pthread**" é o nome da biblioteca do PTHREADS. Quando executado, como as threads são assíncronas, PODE produzir o seguinte resultado:

```
Inicio da Thread 1
Inicio da Thread 2
Thread 1 retorna: 0
Thread 2 retorna: 0
```

Sincronização de threads - Mutexes

Mutexes, que são uma pequena simplificação no conceito de **semáforos**, são usados para evitar inconsistências de dados devido a "*condições de corrida*". Uma condição de corrida ocorre quando duas ou mais threads precisam fazer alguma operação na mesma área de memória (variáveis), mas os resultados da computação dependem da ordem no qual tais operações são feitas. Portanto, **mutexes** são utilizados para "serializar" os recursos compartilhados garantindo o acesso **mutualmente exclusivo** ao recurso. Neste caso, sempre que um recurso global é acessado por mais de uma thread, o recurso deve ter um mutex associado com ele. É como se o **mutex** protegesse um segmento de memória (região crítica) das outras threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;
main()
{
  int rc1, rc2;
  pthread_t thread1, thread2;
  // Cria threads independentes mas cada executa a mesma funcao
  if((rc1=pthread_create(&thread1, NULL, &functionC, NULL)))
  {
    printf("Falha na criacao da thread: %d\n", rc1);
  }
  if( (rc2=pthread_create(&thread2, NULL, &functionC, NULL)) )
  {
    printf("Falha na criacao da thread: %d\n", rc2);
  }
  // IMPORTANTE!
  // Espera ate que as threads se completem
```

```
  // Se nao esperar, corre-se o risco de sair e terminar
  // o processo antes que todas as threads finalizem
  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);
  exit(0);
}

void *functionC()
{
  pthread_mutex_lock( &mutex1 );
  counter++;
  printf("Contador: %d\n",counter);
  pthread_mutex_unlock( &mutex1 );
}
```

Se o arquivo for salvo com o nome "**mutex1.c**" a forma de compilação é:

```
gcc -lpthread mutex1.c -o mutex1
```

onde "**pthread**" é o nome da biblioteca do PTHREADS. Quando executado, mesmo as threads sendo assíncronas, o seguinte resultado é produzido:

```
Contador: 1
Contador: 2
```

Faça o teste retirando as linhas

```
pthread_mutex_lock(&mutex1) e pthread_mutex_unlock(&mutex1)
```

o resultado pode muito bem ser:

```
Counter value: 1
Counter value: 1
```

Você pode criar múltiplas threads usando vetores, da seguinte forma:

```
#define NTHREADS 10
pthread_t thread_id[NTHREADS];
...
for(i=0; i < NTHREADS; i++)
   pthread_create(   &thread_id[i],   NULL,   thread_function,
NULL );
```

Da mesma forma, você pode esperar que as múltiplas threads terminem usando vetores, da seguinte forma:

```
for(i=0; i < NTHREADS; i++)
   pthread_join( &thread_id[i], NULL );
```

ESPECIFICAÇÃO DO TRABALHO

Os exercícios requerem a execução e medição de tempo sob diferentes configurações de quantidade de threads e de quantidade dos dados. Estes resultados deverão ser colocados em uma tabela. Com o intuito de minimizar efeitos locais, é exigido que os dados dos relatórios contenham os valores médios de **pelo menos três** execuções em cada configuração.

Como Este trabalho visa medir o tempo de execução para diversas configurações. Para o cálculo do tempo de execução pode ser usado tanto o **`gettimeofday`** ou **`clock_gettime`**.

A **data de entrega** máxima será no dia 22 de junho (quinta), até meia-noite, via **GITHUB**. É obrigatório incluir todos relatórios e os códigos.

Se você decidir passar tanto a quantidade de threads quanto o tamanho do vetor como argumento de linha de comando (**`argc e argv`**), é importante que seja enviado também um **script** que automatize as várias execuções.

Inclua no relatório a **arquitetura da máquina** em que você executou os experimentos. No Linux esta informação pode ser obtida via o comando **`$ cat /proc/cpuinfo`**. No Mac OS X pode ser obtido através do comando **`sysctl -n machdep.cpu.brand_string`**.

EXERCÍCIO 1:

Implemente um programa para **somar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, utilizando múltiplas threads. As threads vão somar os valores em **uma única variável global protegida por um semáforo MUTEX**.

Antes de criar as threads, você vai "dividir" **o vetor em k partes** e passar os índices de início e fim de cada parte para a respectiva thread. Por exemplo, suponha que N = 32K e k = 4, nesse caso, cada thread vai ficar responsável por somar 8K valores. A thread[0] vai receber os valores 0 (início) e 8191 (fim); thread[1] vai receber os valores 8192 (início) e 16383 (fim); thread[2] vai receber os valores 16384 (início) e 24575 (fim); e thread[3] vai receber os valores 24576 (início) e 32767 (fim).

Escreva uma **relatório** que avalie o tempo de execução para os seguintes valores de N: 32K, 64K, 128K, 256K, 512K, 1024K, e 2048K; e para os seguintes valores de k: 4, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096. Quer dizer, avalie N=32K e k=4, depois avalie N=32K e k=16, depois avalie N=32K e k=32; e assim por diante. O relatório deve ter mais ou menos o seguinte formato:

	32K	64K	128K	256K	512K	1024K	2048K
4							
16							
32							
64							
128							

256							
512							
1024							
2048							
4096							

No relatório, inclua sete **gráficos em linhas**, um para cada tamanho do dado (32K, 64K, 128K, 256K, 512K, 1024K, e 2048K), onde a abscissa é a quantidade de threads (4, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096) e a ordenada é o tempo de execução. **Avalie se é possível colocar os sete gráficos em um único gráfico** (às vezes deve-se usar a escala **logarítmica**). É importante incluir as legendas nos gráficos.

EXERCÍCIO 2:

Da mesma forma que o exercício anterior, implemente um programa para **somar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, utilizando múltiplas threads. Dessa vez, as threads vão somar os valores e **armazenar diretamente em um outro vetor** `soma_parcial` de **dimensão k**, que será usado para armazenar os k valores parciais da soma.

A função `main` deve somar os valores do vetor `soma_parcial`. No cálculo do tempo de execução, **inclua** o tempo de cálculo do valor da soma final sobre o vetor `soma_parcial`.

As configurações da quantidade de dados e número de threads são as mesmas do exercício 1, isto é, o relatório deve estar no mesmo formato de tabela.

Da mesma forma, inclua no relatório sete **gráficos em linhas**, um para cada tamanho do dado (32K, 64K, 128K, 256K, 512K, 1024K, e 2048K), onde a abscissa é a quantidade de threads (4, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096) e a ordenada é o tempo de execução. **Avalie se é possível colocar os sete gráficos em um único gráfico** (às vezes deve-se usar a escala **logarítmica**). É importante incluir as legendas nos gráficos.

EXERCÍCIO 3:

Altere a implementação do exercício 2 no sentido de que, agora, os valores são somados em **uma variável local** e só após o loop é que esta variável local deverá ser movida para o vetor `soma_parcial` (dimensão k).

Da mesma forma que o exercício 2, a função main deve somar os valores do vetor `soma_parcial`. No cálculo do tempo de execução, inclua o tempo de cálculo do valor da soma final sobre o vetor `soma_parcial`.

As configurações da quantidade de dados e número de threads são das mesmas do exercício 1, isto é, o relatório deve estar no mesmo formato de tabela.

No relatório, inclua sete **gráficos em linhas**, um para cada tamanho do dado (32K, 64K, 128K, 256K, 512K, 1024K, e 2048K), onde a abscissa é a quantidade de threads (4, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096) e a ordenada é o tempo de execução. **Avalie se é possível colocar os sete gráficos em um único gráfico** (às vezes deve-se usar a escala **logarítmica**). É importante incluir as legendas nos gráficos.

EXERCÍCIO 4:

Compare os resultados dos três exercícios anteriores. Faça comentários sobre o tempos de execução quando comparados entre a quantidade de threads e tamanho dos dados. Inclua nos comentários as situações em que uma estratégia é melhor (ou pior) que as outras. Por exemplo, às vezes o desempenho (em termos de tempo de execução) é praticamente igual para certas quantidades de dados e número de threads, mas que o desempenho vai mudando de acordo com o incremento ou na quantidades de dados ou na número de threads. Outras vezes, o número de threads elevado, ao invés de ajudar, acaba atrapalhando por causa dos overheads. São esses tipos de comparações e comentários que devem ser feitos. Citar os casos específicos com números.

EXERCÍCIO 5:

Implemente um programa para **ordenar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, usando a estratégia **Quicksort Sequencial**. Escreva uma **relatório** que avalie o tempo de execução para os seguintes valores de N: 32K, 64K, 128K, 256K, 512K, 1024K, 2048K.

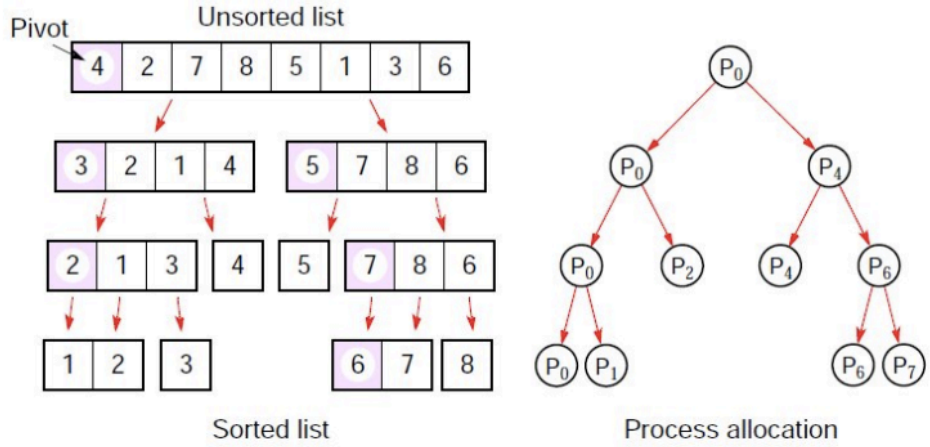
EXERCÍCIO 6:

Implemente um programa para **ordenar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**, usando a estratégia **Quicksort Paralelo**. Apesar de haver estratégias mais eficientes, esta implementação será ingênua (*naive*).

Uma maneira de paralelizar o quicksort é executá-lo inicialmente em uma única thread. Quando o algoritmo vai fazer as chamadas recursivas, deve-se atribuir SOMENTE um dos subproblemas para outra thread. O algoritmo termina quando o vetor não pode ser mais particionado. Esta formulação paralela do quicksort **usa n threads para ordenar n elementos**. **Veja na figura abaixo**.

Quicksort em paralelo

Usando uma atribuição de trabalho a processos em árvore.



Um problema dessa implementação é que o particionamento continua sendo feito de forma sequencial. O tratamento desta situação está fora do escopo deste exercício.

Escreva uma **relatório** que avalie o tempo de execução para os seguintes valores de N: 32K, 64K, 128K, 256K, 512K, 1024K, 2048K, usando a implementação do quicksort paralelo.

Plote um **gráfico** (pode ser **em linhas**) que compare os tempos de execução do quicksort sequencial com o quicksort paralelo. Neste caso, a **abscissa (eixo do X)** é a tamanho do dado (32K, 64K, 128K, 256K, 512K, 1024K, 2048K) e a **ordenada (eixo do Y)** é o tempo de execução. Portanto, o gráfico deve conter duas séries: quicksort sequencial; e quicksort paralelo. Às vezes pode ser necessário usar a escala **logarítmica**.

Adicione no relatório uma **comparação** entre os resultados do Quicksort sequencial com o do paralelo, simplesmente justificando as razões para os resultados obtidos para cada valor de N.