LABORATÓRIO DE PROGRAMAÇÃO AVANÇADA
DÉCIMO PRIMEIRO TRABALHO PRÁTICO
- - PTHREADS - -

Neste trabalho vamos exercitar **múltiplas threads**. Vamos utilizar o Posix Threads (**Pthreads**). 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 = "Thread 1";
  char *msg2 = "Thread 2";
  int iret1, iret2;
  // Cria threads independentes mas cada uma vai executar 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 antes do main continuar.
  // 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 "**pthread1.c**" a forma de compilação é:

```
gcc -lpthread pthread1.c -o pthread1
```

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

```
Thread 2
Thread 1
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, 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 uma vai executar 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 antes do main continuar.
  // 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 );
}
```

ESPECIFICAÇÃO DO EXERCÍCIO 1

Você deve fazer um programa para **somar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**.

Para fins de comparação, a primeira coisa a ser feita é a **thread main** deve fazer é **realizar a soma dos valores do vetor** de forma sequencial (sem uso de threads).

Em seguida, você vai criar diversas threads para resolver o problema da soma. No caso, antes de criar as threads, você vai "dividir" o vetor em k partes e passará os índices de início e fim de cada parte para a respectiva thread. Por exemplo, suponha que N = 2000 e k = 4, nesse caso, cada thread vai ficar responsável por somar 500 valores. A thread[0] vai receber os valores 0 (início) e 499 (fim); thread[1] vai receber os valores 500 (início) e 999 (fim); thread[2] vai receber os valores 1000 (início) e 1499 (fim); e thread[3] vai receber os valores 1500 (início) e 1999 (fim). Sugiro que se crie um outro vetor `soma_parcial`, nesse caso de dimensão k, para armazenar os k valores parciais da soma. A função main deve somar os valores do vetor `soma_parcial`.

Produza um relatório para os seguintes valores de N: 500, 1000, 2000, 5000, 10000 e 50000; e para os seguintes valores de k: 2, 4, 8, 16, 32 e 64. Quer dizer, avalie N=500 e k=2, depois avalie N=500 e k=4, depois avalie N=500 e k=8; e assim por diante.

Avalie a solução para os seguintes valores de N: 500, 1000, 2000, 5000, 10000 e 50000; e para os seguintes valores de k: 2, 4, 8, 16, 32 e 64. Quer dizer, avalie N=500 e k=2, depois avalie N=500 e k=4, depois avalie N=500 e k=8; e assim por diante. O relatório deve ter mais ou menos o seguinte formato:

	500	1000	2000	5000	10000	50000
2						
4						
8						
16						
32						
64						

Cada execução deve ser medido o tempo de execução (via **`gettimeofday`**), tanto a computação feita somente pela **thread main**, quanto a que foi realizada pelas múltiplas threads.

Não esqueça que todas as variáveis globais (que podem ser acessadas pelas múltiplas threads concorrentemente) devem ser sincronizadas, isto é, tem-se que garantir o acesso exclusivo aos dados. Sugiro usar MUTEXES.

Faça comentários sobre os tempos de computação. Sugiro que os valores de N e k sejam passados via argumentos para o programa (`argc` e `argv`).

ESPECIFICAÇÃO DO EXERCÍCIO 2

Faça um programa que multiplique duas matrizes (A e B) usando Pthreads. Vamos assumir que ambas as matrizes são quadradas de dimensão **N**. Os valores das matrizes são geradas **aleatoriamente** (use **`srand`** e **`rand`**). Para fins de comparação, a **thread main** deve **realizar a multiplicação das matrizes e marcar o tempo de execução** (use **`gettimeofday`**). Em seguida, deve ser criado um conjunto de **k** threads. A tarefa de cada thread é calcular o valor da multiplicação para uma linha da matriz A por uma coluna da matriz B e armazenar no respectivo elemento da matriz C. O valor da linha e coluna deve ser passado pela função main. Meça o tempo de execução usando as threads e compare com o tempo de execução sequencial. Produza um relatório para os seguintes valores de N: **100, 200, 400, 800 e 1000**; e para os seguintes valores de k: **2, 4, 8, 16, 32 e 64**. Proponha uma estratégia para melhor resolver o problema da multiplicação de matrizes.

- - -

Este trabalho deve ser entregue no dia **02/07/2014** (quinta) até meia-noite.
IMPORTANTE! Após esta data, o trabalho não será mais aceito.
Envie para o professor (**xbarretox@gmail.com**) e o monitor (**gabriel.leitao@gmail.com**).
Coloque no assunto: [**LPAV-TP11**].