LABORATÓRIO DE PROGRAMAÇÃO AVANÇADA
DÉCIMO SEGUNDO 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 YoLinux 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;
  // Create independent threads each of which will execute function
  iret1=pthread_create(&thr1,NULL,print_msg,(void*) msg1);
  iret2 = pthread_create(&thr2,NULL,print_msg,(void*) msg2);

  // Wait till threads are complete before main continues. Unless we
  // wait we run the risk of executing an exit which will terminate
  // the process and all threads before the threads have completed
  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);
  printf("Thread 1 returns: %d\n",iret1);
  printf("Thread 2 returns: %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 returns: 0
Thread 2 returns: 0
```

Sincronização de threads - Mutexes

Mutexes 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. Mutexes são utilizados para "serializar" os recursos compartilhados. 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;
  /* Create independent threads which will execute functionC */
  if((rc1=pthread_create( &thread1, NULL, &functionC, NULL)))
  {
    printf("Thread creation failed: %d\n", rc1);
  }
  if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
  {
    printf("Thread creation failed: %d\n", rc2);
  }
```

```
  // Wait till threads are complete before main continues. Unless we
  // wait we run the risk of executing an exit which will terminate
  // the process and all threads before the threads have completed
  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);
  exit(0);
}

void *functionC()
{
  pthread_mutex_lock( &mutex1 );
  counter++;
  printf("Counter value: %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:

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

EXERCÍCIO:

Você deve fazer um programa para **somar** um conjunto de **N** números armazenados em um vetor gerado **aleatoriamente**. Para fins de comparação, a **thread main** deve fazer é **realizar a soma dos valores do vetor**. Em seguida, deve ser criado um conjunto de **k** threads, onde cada thread tem acesso ao vetor completo. A thread vai remover dois números, adiciona-os e inclui o resultado na adição novamente no vetor. Deve haver uma variável **global** que controla a quantidade de elementos no vetor, que vai diminuindo na medida que as threads vão somandos os elementos dois a dois. Quando existir somente um valor no vetor, todas as threads terminam, e a thread principal deve imprimir os DOIS valores do somatório, a primeira feita pela thread main, e a segunda, feita pelas threads (**só para efeito**

de comparação). Preste atenção na necessidade de sincronização (**mutex**) nas variáveis compartilhadas.

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.

Avalie a solução para os seguintes valores de N: 100, 200, 500, 1000, 2000 e 5000; e para os seguintes valores de k: 2, 3, 5, 10, 20 e 50. Quer dizer, avalie N=100 e k=2, depois avalie N=100 e k=3, depois avalie N=100 e k=5; e assim por diante. 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`).

- - -

Este trabalho deve ser entregue no dia **25/07/2014**. Entretanto, vou liberar a entrega até a data máxima de **28/07/2014** (segunda) até meia-noite.

IMPORTANTE! Após esta data, o trabalho não será mais aceito.

Envie para o professor (**xbarretox@gmail.com**) e o monitor (**marrco.santos@gmail.com**).

Coloque no assunto: [**LPAV-TP12**].