Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Faculdade de Computação
Bacharelado em Ciências da Computação
Laboratório de Sistemas Operacionais

Threads em Linux – Exclusão Mútua

Introdução

Mutex é a abreviação em inglês de *Mutual Exclusion* que em português significa Exclusão Mútua. As variáveis do tipo *mutex* são utilizadas para sincronização de *threads* e proteção de dados compartilhado quando múltiplas operações de escritas podem ocorrer. É uma prevenção para que leituras incorretas sejam efetuadas quando operações de escritas ocorrem em uma área compartilhada.

Uma variável *mutex* funciona como uma trava de proteção para o acesso a dados compartilhados. Nas *threads* que utilizam *Pthreads* somente uma *thread* pode travar, ou liberar, uma variável *mutex* por vez. Caso várias *threads* tentem travar uma variável *mutex* ao mesmo tempo, somente uma conseguirá travar. Somente a *thread* que efetuou o travamento da variável *mutex* pode liberá-la.
Em computação, as variáveis *mutex* são usadas para prevenir condições de corrida, tais como a ilustrada abaixo. A variável *Balance* deverá ser do tipo *mutex* para que não ocorra erro na sua atualização.

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Variáveis *mutex*

Variáveis *mutex* devem ser declaradas com o tipo *pthread_mutex_t* e deve ser inicializada antes de seu uso. Existem 02 (duas) formas de inicializar uma variável *mutex*:

- Estaticamente, quando for declarada
  - *pthread _mutex_t mymutex = PTHREAD_MUTEX_INITALIZER;*
- Dinamicamente. Este método permite ajustar os atributos, *attr*.
  - *phtread_mutex_init()*

As variáveis *mutex* são criadas destravadas e as funções usadas para travar e destravar uma variável *mutex* são:

- *pthread_mutex_lock(mutex)*
  - Usada para obter a trava de uma variável *mutex*.
  - Se a variável *mutex* já estiver travada por outra *thread*, a thread irá ser bloqueada até a liberação da variável *mutex*.
- *pthread_mutex_trylock(mutex)*
  - Igual a anterior, porém caso a variável esteja travada a função retorna erro de código *busy* e continua a sua execução.
- *pthread_mutex_unlock(mutex)*

- Executará a liberação da variável *mutex* caso a thread seja a sua proprietária.

A boa prática de programação indica que todas as variáveis travadas devem ser destravas, senão a ocorrência de *deadlock* poderá travar o programa.

Produto escalar

O exemplo abaixo ilustra o uso de variáveis mutex em threads que calculam um produto escalar. O dado principal fica visível para todas as *threads* acessarem através de uma estrutura de dados global e cada *thread* executa a tarefa em uma parte diferente dos dados. A *thread* principal aguarda que as demais *threads* terminem os cálculos para depois mostrar o resultado.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

#define NUMTHRDS 4                    // Número de threads
#define VECLEN 100000
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;          // Variável mutex

void *dotprod(void *arg)
{

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   pthread_mutex_lock (&mutexsum);    // Executa o travamento da variável a ser
manipulada
   dotstr.sum += mysum;
   printf("Thread %ld did %d to %d:  mysum=%f global
sum=%f\n",offset,start,end,mysum,dotstr.sum);
   pthread_mutex_unlock (&mutexsum);  // Destrava a variável

   pthread_exit((void*) 0);
}

int main (int argc, char *argv[])
```

```
{
long i;
double *a, *b;
void *status;
pthread_attr_t attr;

a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

for (i=0; i<VECLEN*NUMTHRDS; i++) {
  a[i]=1;
  b[i]=a[i];
  }

dotstr.veclen = VECLEN;
dotstr.a = a;
dotstr.b = b;
dotstr.sum=0;

// attr = NULL aceita atributos default
pthread_mutex_init(&mutexsum, NULL);

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0;i<NUMTHRDS;i++)
  {
   pthread_create(&callThd[i], &attr, dotprod, (void *)i);
   }

pthread_attr_destroy(&attr);

// Aguarda a finalização de todas as threads
for(i=0;i<NUMTHRDS;i++) {
  pthread_join(callThd[i], &status);
  }

printf ("Sum =  %f \n", dotstr.sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

Fonte: https://computing.llnl.gov/tutorials/pthreads/samples/dotprod_mutex.c

TAREFA

- Explicar o programa acima e como as variáveis *mutex* são utilizadas para controlar o acesso à memória compartilhada.
- Implementar o controle do saldo (*Balance*) em uma conta corrente de acordo com a figura acima sem e com variável *mutex*.
    - Duas *threads* devem ser criadas e devem executar a operação de retirada e depósito de forma aleatória;
    - Cada *thread* deve executar 5 operações de depósito e 5 operações de retirada, sendo que os valores dessas operações devem ser aleatórios e pertencer ao intervalo [100,500];
    - A conta corrente deverá ter saldo inicial de 10.000.