O Problema da Exclusão Mútua

Construir mecanismos para garantir a exclusão mútua entre processos ou *threads* não é uma tarefa trivial. Esta página traz alguns exemplos de mecanismos para obter a exclusão mútua - alguns dos quais não funcionam!

O código de base consiste em 100 *threads*, onde cada *thread* tenta fazer 100.000 incrementos em uma variável global compartilhada `sum`. Se tudo correr bem, o valor final da variável `sum` deve ser 100×100.000 = 10.000.000.

Para medir a duração da execução, use o seguinte comando:

```
$ time <programa-compilado>
```

Sem coordenação

Neste código, as threads acessam a variável compartilhada sem nenhum controle de concorrência.

me1-none.c

```
/*
Acesso concorrente a uma variável por muitas threads, sem controle.

Compilar com gcc -Wall me1-none.c -o me1 -lpthread

Carlos Maziero, DINF/UFPR 2020
*/

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

#define NUM_THREADS  100
#define NUM_STEPS 100000

int sum = 0 ;

void *threadBody (void *id)
{
  int i ;

  for (i=0; i< NUM_STEPS; i++)
  {
    sum += 1 ;   // critical section
  }

  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
  pthread_attr_t attr ;
  long i, status ;
```

```
  // define attribute for joinable threads
  pthread_attr_init (&attr) ;
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE) ;

  // create threads
  for(i=0; i<NUM_THREADS; i++)
  {
    status = pthread_create (&thread[i], &attr, threadBody, (void *) i) ;
    if (status)
    {
      perror ("pthread_create") ;
      exit (1) ;
    }
  }

  // wait all threads to finish
  for (i=0; i<NUM_THREADS; i++)
  {
    status = pthread_join (thread[i], NULL) ;
    if (status)
    {
      perror ("pthread_join") ;
      exit (1) ;
    }
  }

  printf ("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum) ;

  pthread_attr_destroy (&attr) ;
  pthread_exit (NULL) ;
}
```

A solução ingênua

Esta solução consiste em definir uma variável de controle `busy`, que controla a entrada na seção crítica (incremento da variável `sum`).

me2-naive.c

```
/*
Acesso concorrente a uma variável por muitas threads, solução "ingênua".

Compilar com gcc -Wall me2-naive.c -o me2 -lpthread

Carlos Maziero, DINF/UFPR 2020
*/

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

#define NUM_THREADS  100
```

```c
#define NUM_STEPS 100000

int sum  = 0 ;
int busy = 0 ;

// enter critical section
void enter_cs ()
{
  while ( busy ) ;   // busy waiting
  busy = 1 ;
}

// leave critical section
void leave_cs ()
{
  busy = 0 ;
}

void *threadBody(void *id)
{
  int i ;

  for (i=0; i< NUM_STEPS; i++)
  {
    enter_cs () ;
    sum += 1 ;   // critical section
    leave_cs () ;
  }

  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
  pthread_attr_t attr ;
  long i, status ;

  // define attribute for joinable threads
  pthread_attr_init (&attr) ;
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE) ;

  // create threads
  for(i=0; i<NUM_THREADS; i++)
  {
    status = pthread_create (&thread[i], &attr, threadBody, (void *) i) ;
    if (status)
    {
      perror ("pthread_create") ;
      exit (1) ;
    }
  }

  // wait all threads to finish
  for (i=0; i<NUM_THREADS; i++)
  {
```

```
    status = pthread_join (thread[i], NULL) ;
    if (status)
    {
      perror ("pthread_join") ;
      exit (1) ;
    }
  }

  printf ("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum) ;

  pthread_attr_destroy (&attr) ;
  pthread_exit (NULL) ;
}
```

Alternância

Esta solução consiste em definir uma variável `turn` que indica quem pode acessar a seção crítica a cada instante. Ao sair da seção crítica, cada *thread* incrementa o valor dessa variável, fazendo com que a próxima *thread* tenha acesso à seção crítica.

Esta solução funciona, mas é muuuuito lenta. Por que?

me3-altern.c

```
/*
Acesso concorrente a uma variável por muitas threads, solução com alternância.

Compilar com gcc -Wall me3-altern.c -o me3 -lpthread

Carlos Maziero, DINF/UFPR 2020
*/

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

#define NUM_THREADS  100
#define NUM_STEPS 100000

int sum  = 0 ;
int turn = 0 ;

// enter critical section
void enter_cs (long int id)
{
  while (turn != id) ;   // busy waiting

  if (sum % 100 == 0)
    printf ("Turn: %d, Sum: %d\n", turn, sum) ;
}

// leave critical section
void leave_cs ()
{
```

```
  turn = (turn + 1) % NUM_THREADS ;
}

void *threadBody(void *id)
{
  int i ;

  for (i=0; i< NUM_STEPS; i++)
  {
    enter_cs ((long int) id) ;
    sum += 1 ;   // critical section
    leave_cs () ;
  }

  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
  pthread_attr_t attr ;
  long i, status ;

  // define attribute for joinable threads
  pthread_attr_init (&attr) ;
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE) ;

  // create threads
  for(i=0; i<NUM_THREADS; i++)
  {
    status = pthread_create (&thread[i], &attr, threadBody, (void *) i) ;
    if (status)
    {
      perror ("pthread_create") ;
      exit (1) ;
    }
  }

  // wait all threads to finish
  for (i=0; i<NUM_THREADS; i++)
  {
    status = pthread_join (thread[i], NULL) ;
    if (status)
    {
      perror ("pthread_join") ;
      exit (1) ;
    }
  }

  printf ("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum) ;

  pthread_attr_destroy (&attr) ;
  pthread_exit (NULL) ;
}
```

Operações atômicas

Esta solução consiste em usar operações atômicas, como *Test-and-Set Lock* e similares. No exemplo abaixo é usada uma operação OR atômica.

me4-tsl.c

```
/*
Acesso concorrente a uma variável por muitas threads, solução com TSL.

Compilar com gcc -Wall me4-tsl.c -o me4 -lpthread

Carlos Maziero, DINF/UFPR 2020
*/

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

#define NUM_THREADS  100
#define NUM_STEPS 100000

int sum  = 0 ;
int lock = 0 ;

// enter critical section
void enter_cs (int *lock)
{
  // atomic OR (Intel macro for GCC)
  while (__sync_fetch_and_or (lock, 1)) ;   // busy waiting
}

// leave critical section
void leave_cs (int *lock)
{
  (*lock) = 0 ;
}

void *threadBody(void *id)
{
  int i ;

  for (i=0; i< NUM_STEPS; i++)
  {
    enter_cs (&lock) ;
    sum += 1 ;   // critical section
    leave_cs (&lock) ;
  }

  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
```

```
  pthread_attr_t attr ;
  long i, status ;

  // define attribute for joinable threads
  pthread_attr_init (&attr) ;
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE) ;

  // create threads
  for(i=0; i<NUM_THREADS; i++)
  {
    status = pthread_create (&thread[i], &attr, threadBody, (void *) i) ;
    if (status)
    {
      perror ("pthread_create") ;
      exit (1) ;
    }
  }

  // wait all threads to finish
  for (i=0; i<NUM_THREADS; i++)
  {
    status = pthread_join (thread[i], NULL) ;
    if (status)
    {
      perror ("pthread_join") ;
      exit (1) ;
    }
  }

  printf ("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum) ;

  pthread_attr_destroy (&attr) ;
  pthread_exit (NULL) ;
}
```

A Instrução XCHG

Esta solução é similar à anterior, mas usa a instrução XCHG da plataforma Intel.

me5-xchg.c

```
/*
Acesso concorrente a uma variável por muitas threads, solução com instrução
XCHG.

Compilar com gcc -Wall me5-xchg.c -o me5 -lpthread

Carlos Maziero, DINF/UFPR 2020
*/

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

```
#define NUM_THREADS  100
#define NUM_STEPS 100000

int sum  = 0 ;
int lock = 0 ;

// enter critical section
void enter_cs (int *lock)
{
  int key = 1 ;
  while (key)  // busy waiting
  {
    // XCHG lock, key
    __asm__ __volatile__("xchgl %1, %0"     // assembly template
                         : "=r"(key)      // output
                         : "m"(*lock), "0"(key)    // input
                         : "memory") ;        // clobbered registers
  }
}

// leave critical section
void leave_cs (int *lock)
{
  (*lock) = 0 ;
}

void *threadBody(void *id)
{
  int i ;

  for (i=0; i< NUM_STEPS; i++)
  {
    enter_cs (&lock) ;
    sum += 1 ;   // critical section
    leave_cs (&lock) ;
  }

  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
  pthread_attr_t attr ;
  long i, status ;

  // define attribute for joinable threads
  pthread_attr_init (&attr) ;
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE) ;

  // create threads
  for(i=0; i<NUM_THREADS; i++)
  {
    status = pthread_create (&thread[i], &attr, threadBody, (void *) i) ;
    if (status)
```

```
    {
      perror ("pthread_create") ;
      exit (1) ;
    }
  }

  // wait all threads to finish
  for (i=0; i<NUM_THREADS; i++)
  {
    status = pthread_join (thread[i], NULL) ;
    if (status)
    {
      perror ("pthread_join") ;
      exit (1) ;
    }
  }

  printf ("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum) ;

  pthread_attr_destroy (&attr) ;
  pthread_exit (NULL) ;
}
```

Com Semáforos

Esta solução controla a entrada na seção crítica através de um semáforo genérico POSIX.

me6-semaphore.c

```
/*
Acesso concorrente a uma variável por muitas threads, solução com semáforo.

Compilar com gcc -Wall me6-semaphore.c -o me6 -lpthread

Carlos Maziero, DINF/UFPR 2020
*/

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

#define NUM_THREADS  100
#define NUM_STEPS 100000

int sum = 0 ;
sem_t s ;

void *threadBody(void *id)
{
  int i ;

  for (i=0; i< NUM_STEPS; i++)
```

```
  {
    sem_wait (&s) ;
    sum += 1 ;   // critical section
    sem_post (&s) ;
  }

  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
  pthread_attr_t attr ;
  long i, status ;

  // initialize semaphore to 1
  sem_init (&s, 0, 1) ;

  // define attribute for joinable threads
  pthread_attr_init (&attr) ;
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE) ;

  // create threads
  for(i=0; i<NUM_THREADS; i++)
  {
    status = pthread_create (&thread[i], &attr, threadBody, (void *) i) ;
    if (status)
    {
      perror ("pthread_create") ;
      exit (1) ;
    }
  }

  // wait all threads to finish
  for (i=0; i<NUM_THREADS; i++)
  {
    status = pthread_join (thread[i], NULL) ;
    if (status)
    {
      perror ("pthread_join") ;
      exit (1) ;
    }
  }

  printf ("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum) ;

  pthread_attr_destroy (&attr) ;
  pthread_exit (NULL) ;
}
```

Com Mutex

Esta solução controla a entrada na seção crítica através de um mutex POSIX.

me7-mutex.c

```c
/*
Acesso concorrente a uma variável por muitas threads, solução com mutex.

Compilar com gcc -Wall me7-mutex.c -o me7 -lpthread

Carlos Maziero, DINF/UFPR 2020
*/

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

#define NUM_THREADS  100
#define NUM_STEPS 100000

int sum = 0 ;
pthread_mutex_t mutex ;

void *threadBody(void *id)
{
  int i ;

  for (i=0; i< NUM_STEPS; i++)
  {
    pthread_mutex_lock (&mutex) ;
    sum += 1 ;   // critical section
    pthread_mutex_unlock (&mutex) ;
  }

  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
  pthread_attr_t attr ;
  long i, status ;

  // initialize mutex
  pthread_mutex_init (&mutex, NULL) ;

  // define attribute for joinable threads
  pthread_attr_init (&attr) ;
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE) ;

  // create threads
  for(i=0; i<NUM_THREADS; i++)
  {
    status = pthread_create (&thread[i], &attr, threadBody, (void *) i) ;
    if (status)
    {
      perror ("pthread_create") ;
      exit (1) ;
    }
```

```
  }

  // wait all threads to finish
  for (i=0; i<NUM_THREADS; i++)
  {
    status = pthread_join (thread[i], NULL) ;
    if (status)
    {
      perror ("pthread_join") ;
      exit (1) ;
    }
  }

  printf ("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum) ;

  pthread_attr_destroy (&attr) ;
  pthread_exit (NULL) ;
}
```