RELATÓRIO DE LABORATÓRIO 5

SISTEMAS OPERACIONAIS

VICTOR MONEGO

ENGENHARIA ELETRÔNICA – UTFPR

29 DE ABRIL DE 2024

1.Introdução Geral

O seguinte relatório diz respeito ao Laboratório 05 de Sistemas Operacionais, focado na comparação de métodos de exclusão mútua.

Para a realização das análises a seguir, foi utilizada a plataforma WSL(Ubuntu) no Windows 11.

O código base consiste em 100 threads, onde cada thread tenta fazer 100.000 incrementos em uma variável global compartilhada "*sum*". Se o processo funcionar bem, o valor final da variável deve ser 10.000.000.

2.Apresentação dos Métodos e seus Tempos de Execução

A figura 01 abaixo apresenta o código "me1-none.c", que é o código que não implementa nenhuma solução.

```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) ;
}
```

Figura 1: código "me1-none.c"

A figura 02 abaixo mostra a medição de tempo de execução do código.

```
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me1
Sum should be 10000000 and is 2351854

real    0m0.064s
user    0m0.300s
sys     0m0.079s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me1
Sum should be 10000000 and is 2010125

real    0m0.062s
user    0m0.203s
sys     0m0.053s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me1
Sum should be 10000000 and is 3481866

real    0m0.062s
user    0m0.181s
sys     0m0.042s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me1
Sum should be 10000000 and is 3160123

real    0m0.063s
user    0m0.211s
sys     0m0.089s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$
```

Figura 2: tempo de execução método 1

A figura 03 abaixo apresenta o código “me2-naive.c”, que é o código que utiliza uma variável de controle na seção crítica.

```
/*
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
#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) ;
}
```

Figura 3: código "me2-naive.c"

A figura 04 abaixo mostra a medição de tempo de execução do código.

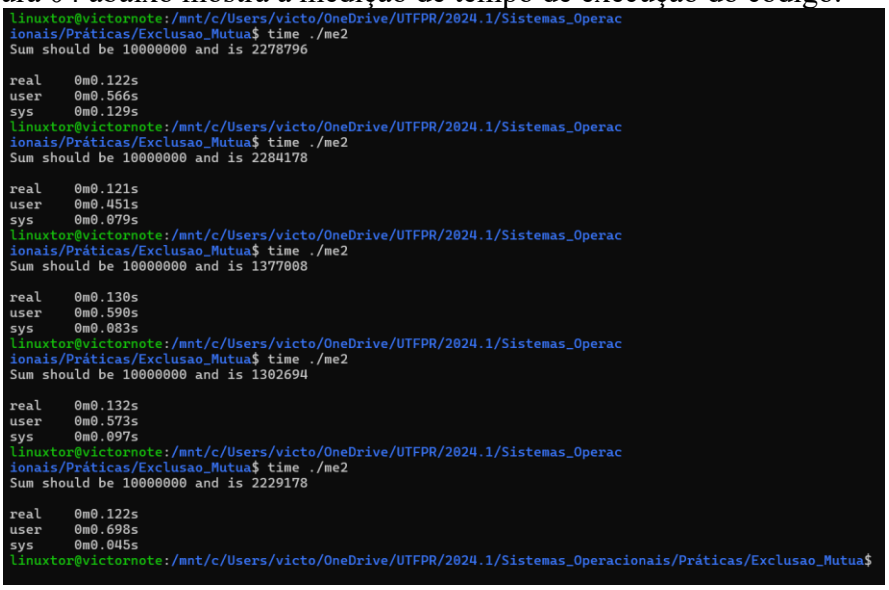

Figura 4: tempo de execução método 2

A figura 05 abaixo apresenta o código “me3-altern.c”, que é o código que .

```
/*
```

```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) ;
}
```

Figura 5: código "me3-altern.c"

A figura 06 abaixo mostra a medição de tempo de execução do código.

Figura 6: tempo de execução método 3

A figura 07 abaixo apresenta o código “me4-tsl.c”, que é o código que não implementa nenhuma solução.

```
/*
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>
```

```c
#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) ;
}
```

Figura 7: código "me4-tsl.c"

A figura 08 abaixo mostra a medição de tempo de execução do código.

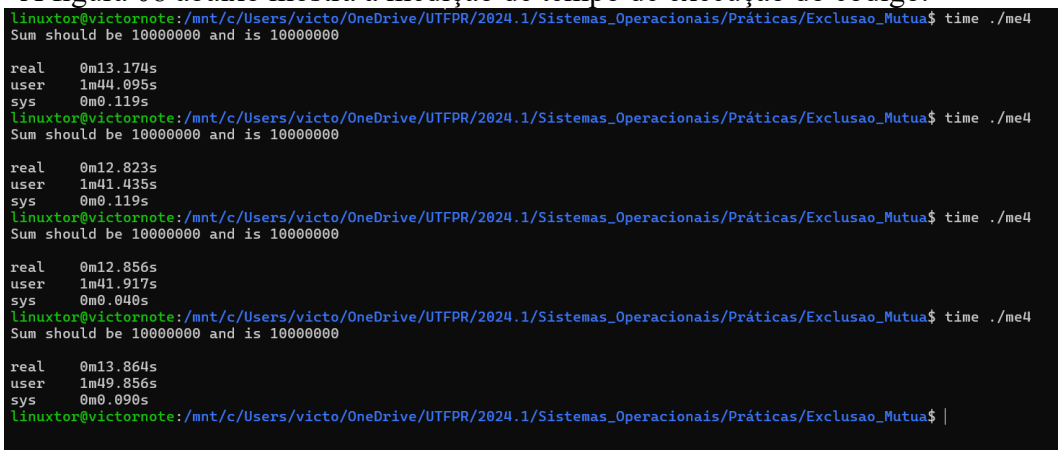

Figura 8: tempo de execução método 4

A figura 09 abaixo apresenta o código “me5-xchg.c”, que é o código que não implementa nenhuma solução.

```
/*
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) ;
}
```

Figura 9: código "me5-xchg"

A figura 10 abaixo mostra a medição de tempo de execução do código.

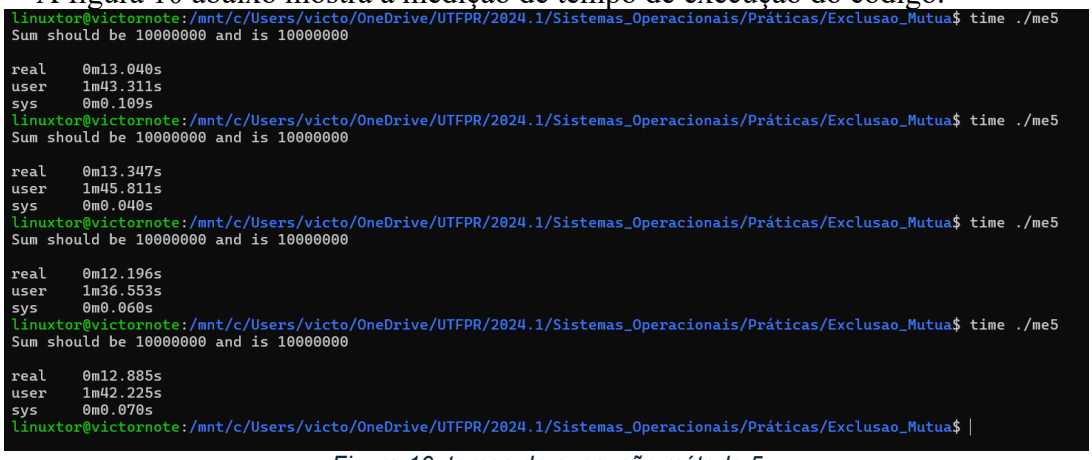

Figura 10: tempo de execução método 5

A figura 11 abaixo apresenta o código “me6-semaphore.c”, que é o código que não implementa nenhuma solução.

```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) ;
}
```

Figura 11: código "me6-semaphore.c"

A figura 12 abaixo mostra a medição de tempo de execução do código.

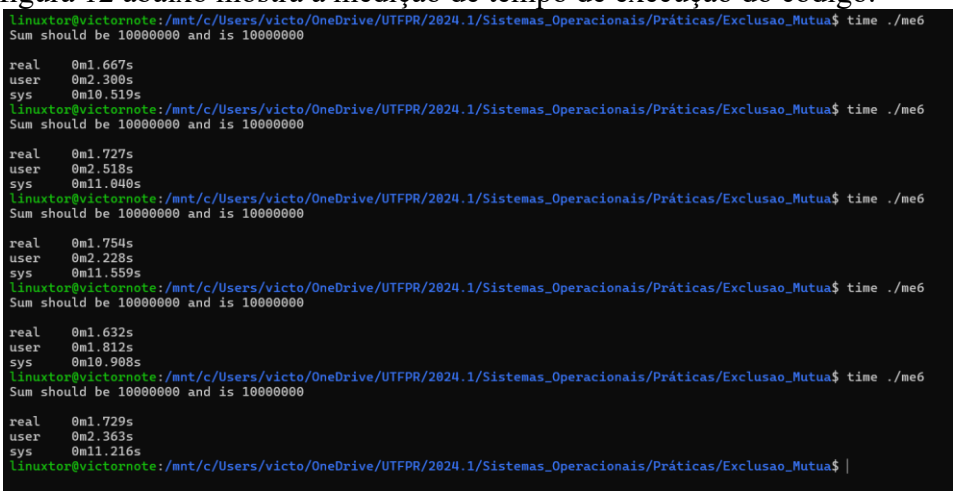

Figura 12: Tempo de execução método 6

A figura 13 abaixo apresenta o código “me7-none.c”, que é o código que não implementa nenhuma solução.

```
/*
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) ;
}
```

Figura 13: código "me7-mutex.c"

A figura 14 abaixo mostra a medição de tempo de execução do código.

```
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me7
Sum should be 10000000 and is 10000000

real    0m0.555s
user    0m0.950s
sys     0m3.107s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me7
Sum should be 10000000 and is 10000000

real    0m0.531s
user    0m0.944s
sys     0m3.019s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me7
Sum should be 10000000 and is 10000000

real    0m0.523s
user    0m1.174s
sys     0m2.643s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me7
Sum should be 10000000 and is 10000000

real    0m0.535s
user    0m0.877s
sys     0m3.009s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me7
Sum should be 10000000 and is 10000000

real    0m0.534s
user    0m0.932s
sys     0m3.037s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$ time ./me7
Sum should be 10000000 and is 10000000

real    0m0.534s
user    0m0.889s
sys     0m2.997s
linuxtor@victornote:/mnt/c/Users/victo/OneDrive/UTFPR/2024.1/Sistemas_Operacionais/Práticas/Exclusao_Mutua$
```

Figura 14: Tempo de execução método 7

3. Discussão sobre as soluções apresentadas

- “me1-none.c” - Este código não faz uso de mecanismos de exclusão mútua. A concorrência entre os threads é descontrolada e a soma nunca é igual à soma prevista. Por não existir controle, os threads incrementam a variável incondicionalmente, e isso provoca o fim do programa com a variável global não sendo incrementada até o valor projetado.

- “me2-naive.c” - Neste código, é utilizado um esquema de "espera ocupada" (busy waiting) para controlar o acesso à seção crítica. No entanto, isso não é eficiente, pois os threads continuam verificando repetidamente o estado da variável busy, o que pode causar um alto consumo de CPU e não garante a exclusão mútua de forma eficaz.

- “me3-altern.c” - Aqui, é implementada uma solução com alternância entre os threads, controlada pela variável turn. Embora garanta a exclusão mútua, pode levar a problemas de desempenho se um thread tiver prioridade sobre as outras, resultando em potencial inanição de alguns threads. Além disso, a implementação leva um tempo absurdo para se completar, visto que cada ciclo de iteração faz os threads incrementarem individualmente, o que é um desperdício de tempo de compilação e de recursos.

- “me4-tsl.c” - Este código utiliza a instrução “*__sync_fetch_and_or*” para implementar um mecanismo de exclusão mútua baseado em "*Test-and-Set Lock*". Isso garante a exclusão mútua de forma mais eficiente do que a espera ocupada, mas ainda pode ter problemas de desempenho em sistemas com muitos threads devido ao uso da instrução em loop.

- “me5-xchg.c” - Aqui, é usada a instrução de troca XCHG para implementar a exclusão mútua. Essa abordagem é mais eficiente do que as anteriores, mas pode não ser suportada em todas as arquiteturas e compiladores.

- “me6-semaphore.c” - Utiliza um semáforo binário para controlar o acesso à seção crítica. Essa é uma solução clássica para o problema de exclusão mútua e é eficiente em termos de desempenho e uso de CPU.

- “me7-mutex.c” - Este código usa um mutex para garantir a exclusão mútua. Essa é uma abordagem robusta e eficiente, geralmente preferida em situações em que a exclusão mútua é necessária.

4. Conclusão: qual método é melhor?

Analisando as abordagens, é possível observar que as duas melhores opções para se resolver o problema da exclusão mútua são o semáforo binário e o mutex. Ambas as soluções garantiram que a variável global fosse incrementada corretamente por todos os threads, porém sem sacrificar muitos recursos para isso. Mais especificamente, a solução com mutex acaba sendo ainda mais eficiente, pois pelos testes de compilação provou ser mais eficiente ainda, especificamente no quesito de tempo de execução.

5. Referências

- MAZIERO, C. A. **O Problema da Exclusão Mútua.** Disponível em: https://wiki.inf.ufpr.br/maziero/doku.php?id=so:exclusao_mutua. Acesso em: abril de 2024.