Relatório - Laboratório 8

José Mário Nishihara de Albuquerque

19 de dezembro de 2023

1 Introdução

Neste relatório, exploraremos sete exercícios sobre mecanismos para implementar a exclusão mútua entre processos ou threads. A exclusão mútua é uma propriedade que garante que apenas um processo ou thread possa acessar um recurso compartilhado por vez, evitando conflitos e inconsistências. No entanto, nem todos os mecanismos propostos nos exercícios são capazes de garantir essa propriedade. A ideia é analisar como cada mecanismo funciona e identificar as suas vantagens e desvantagens, bem como os motivos que levam alguns deles a falhar.

2 Códigos

2.1 Exercício 1

Algoritmo 1 me1-none.c

```
0: #include <pthread.h>
0: #include <stdio.h>
0: #include <stdlib.h>
0: #define NUM_THREADS 100
0: #define NUM_STEPS 100000
0: sum = 0
0:
0: void *threadBody (void *id)
0: int i
1: for i = 0 to i < NUM_STEPS do
1:     sum = sum + 1    // critical section
2: end for
2: pthread_exit (NULL)
2:
2: int main (int argc, char argv[])
2: pthread_t thread [NUM_THREADS]
2: pthread_attr_t attr
2: long i, status
2:
2:   // define attribute for joinable threads
2: pthread_attr_init (&attr)
2: pthread_attr_setdetachState (&attr, PTHREAD_CREATE_JOINABLE)
2:
2:   // create threads
3: for i = 0 to i < NUM_THREADS do
3:     status = pthread_create(&thread[i], &attr, threadBody, (void*)i)
4:     if status then
4:         perror ("pthread_create")
4:         exit (1)
5:     end if
6: end for
6:
6:   // wait all threads to finish
7: for i = 0 to i < NUM_THREADS do
7:     status = pthread_join(thread[i], NULL)
8:     if status then
8:         perror ("pthread_join")
8:         exit (1)
9:     end if
10: end for
10:
10: printf ("Sum should be %d and is %d\n", NUM_THREADSNUM_STEPS, sum)
10:
10: pthread_attr_destroy (&attr)
10: pthread_exit (NULL) =0
```

Esse código não é efetivo para garantir a exclusão mútua entre processos, pois ele permite que duas ou mais threads leiam e modifiquem a variável sum ao mesmo tempo, causando uma condição de corrida. A condição de corrida ocorre quando o resultado de uma operação depende da ordem de execução das threads, que é imprevisível. Isso pode levar a resultados incorretos ou inconsistentes, como o valor final de sum ser diferente do esperado.

A seção crítica do código é a linha sum += 1 ;, que incrementa a variável sum em uma unidade. Essa operação não é atômica, ou seja, ela envolve mais de um passo: ler o valor de sum, somar um a ele, e escrever o novo valor de sum. Se duas threads executam essa operação ao mesmo tempo, elas podem ler o mesmo valor de sum, somar um a ele, e escrever o mesmo valor de sum, perdendo assim um incremento. Por exemplo, se sum vale 10 e duas threads executam a seção crítica, elas podem ler 10, somar 1, e escrever 11, em vez de 12.

2.1.1 Saídas

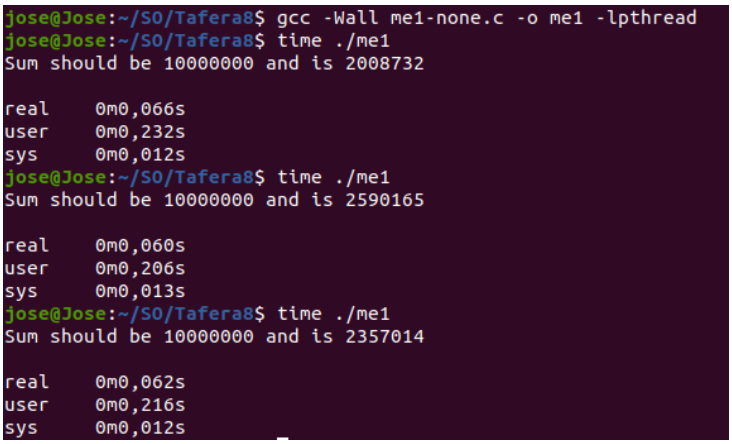

Figura 1: Saídas me1-none.c.

2.2 Exercício 2

Algoritmo 2 me2-naive.c

```
Require: #include <pthread.h>
Require: #include <stdio.h>
Require: #include <stdlib.h>
 0: #define NUM_THREADS 100
 0: #define NUM_STEPS 100000
 0: sum ← 0
 0: busy ← 0   // enter critical section
 0: procedure ENTER_CS
 0:   while busy do
 0:   busy ← 1
      // leave critical section
 0: procedure LEAVE_CS
 0:   busy ← 0
 0: procedure THREADBODY(id)
 1: for i ← 0 to NUM_STEPS − 1 do
 1:     enter_cs
 1:     sum ← sum + 1   // critical section
 1:     leave_cs
 2: end for
 2: pthread_exit(NULL)
 2procedure MAIN(argc, argv)
 2: pthread_t thread[NUM_THREADS]
 2: pthread_attr_t attr
 2: i, status   // define attribute for joinable threads
 2: pthread_attr_init(&attr)
 2: pthread_attr_setdetachState(&attr, PTHREAD_CREATE_JOINABLE)
    // create threads
 3: for i ← 0 to NUM_THREADS − 1 do
 3:   status ← pthread_create(&thread[i], &attr, threadBody, (void *) i)
 4:   if status then
 4:     perror("pthread_create")
 4:     exit(1)
 5:   end if
 6: end for   // wait all threads to finish
 7: for i ← 0 to NUM_THREADS − 1 do
 7:   status ← pthread_join(thread[i], NULL)
 8:   if status then
 8:     perror("pthread_join")
 8:     exit(1)
 9:   end if
10: end for
10: printf("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS,
    sum)
10: pthread_attr_destroy(&attr)
10: pthread_exit(NULL)
=0
```

Esse código não é efetivo para garantir a exclusão mútua entre processos, pois ele usa uma variável busy para indicar se a seção crítica está ocupada ou não. Essa solução é ingênua porque ela também sofre de condição de corrida, ou seja, duas ou mais threads podem ler e modificar a variável busy ao mesmo tempo, causando uma inconsistência. Por exemplo, se busy vale zero e duas threads executam a função enter_cs ao mesmo tempo, elas podem ler zero, sair do laço while, e escrever um, entrando assim na seção crítica simultaneamente.

2.2.1 Saídas

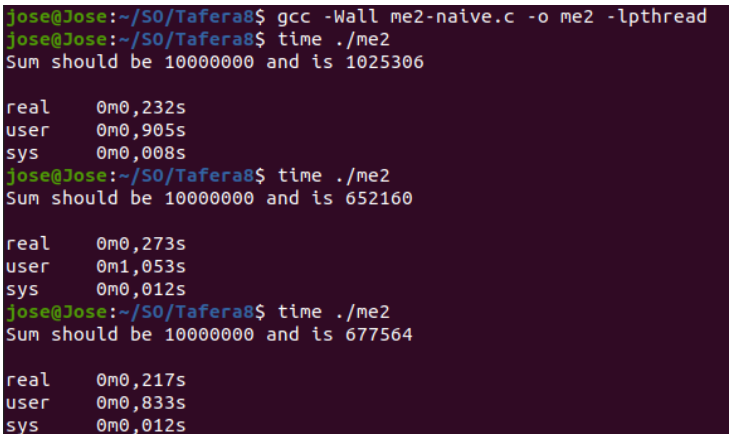

Figura 2: Saídas me2-naive.c.

2.3 Exercício 3

Algoritmo 3 me3-altern.c

Require: `#include <pthread.h>`
Require: `#include <stdio.h>`
Require: `#include <stdlib.h>`
0: `#define NUM_THREADS 100`
0: `#define NUM_STEPS 100000`
0: $sum \leftarrow 0$
0: $turn \leftarrow 0$ // enter critical section
0: **procedure** ENTER_CS(id)
0: **while** $turn \neq id$ **do**
1: **if** $sum \bmod 100 = 0$ **then**
1: `printf("Turn: %d, Sum: %d\n", turn, sum)`
2: **end if**
2: // leave critical section
2: **procedure** LEAVE_CS
2: $turn \leftarrow (turn + 1) \bmod NUM\_THREADS$
2: **procedure** THREADBODY(id)
3: **for** $i \leftarrow 0$ **to** $NUM\_STEPS - 1$ **do**
3: enter_csid
3: $sum \leftarrow sum + 1$ // critical section
3: leave_cs
4: **end for**
4: `pthread_exit(NULL)`
4: **procedure** MAIN($argc, argv$)
4: `pthread_t thread[NUM_THREADS]`
4: `pthread_attr_t attr`
4: $i, status$ // define attribute for joinable threads
4: `pthread_attr_init(&attr)`
4: `pthread_attr_setdetachState(&attr, PTHREAD_CREATE_JOINABLE)`
// create threads
5: **for** $i \leftarrow 0$ **to** $NUM\_THREADS - 1$ **do**
5: $status \leftarrow$ `pthread_create(&thread[i], &attr, threadBody, (void *) i)`
6: **if** $status$ **then**
6: `perror("pthread_create")`
6: `exit(1)`
7: **end if**
8: **end for** // wait all threads to finish
9: **for** $i \leftarrow 0$ **to** $NUM\_THREADS - 1$ **do**
9: $status \leftarrow$ `pthread_join(thread[i], NULL)`
10: **if** $status$ **then**
10: `perror("pthread_join")`
10: `exit(1)`
11: **end if**
12: **end for**
12: `printf("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS, sum)`
12: `pthread_attr_destroy(&attr)`
12: `pthread_exit(NULL)`

Esse código é efetivo para garantir a exclusão mútua entre processos, pois ele usa uma variável turn para indicar qual thread tem a vez de entrar na seção crítica. Ele permite a dois ou mais processos ou subprocessos compartilharem um recurso sem conflitos, utilizando apenas memória compartilhada para a comunicação.

Para que o código seja efetivo, ele usa a função enter_cs para verificar se a thread tem a vez de entrar na seção crítica, comparando o seu identificador id com a variável turn. Se forem iguais, a thread pode entrar na seção crítica e imprimir o valor de sum se ele for múltiplo de 100. Se forem diferentes, a thread deve esperar até que sejam iguais, fazendo uma espera ocupada (busy waiting). Depois de sair da seção crítica, a thread usa a função leave_cs para passar a vez para a próxima thread, incrementando a variável turn e calculando o seu resto pela divisão pelo número de threads. Dessa forma, cada thread tem a sua vez de entrar na seção crítica, seguindo uma ordem circular.

2.3.1 Saídas

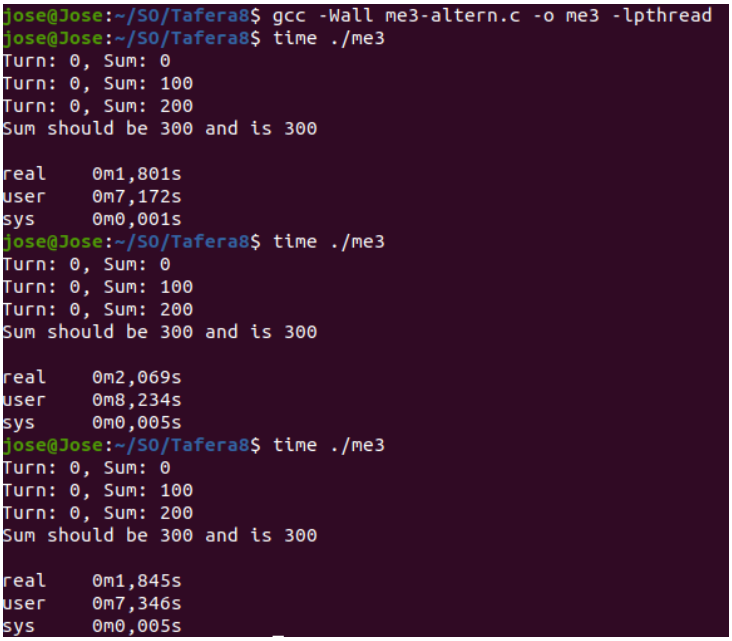

Figura 3: Saídas me3-altern.c.

2.4 Exercício 4

Algoritmo 4 me4-tsl.c

```
Require: #include <pthread.h>
Require: #include <stdio.h>
Require: #include <stdlib.h>
 0: #define NUM_THREADS 100
 0: #define NUM_STEPS 100000
 0: sum ← 0
 0: lock ← 0   // enter critical section
 0: procedure ENTER_CS(lock)   // atomic OR (Intel macro for GCC)
 0:   while __sync_fetch_and_or(lock, 1) do
      // leave critical section
 0: procedure LEAVE_CS(lock)
 0:   lock ← 0
 0: procedure THREADBODY(id)
 1: for i ← 0 to NUM_STEPS − 1 do
 1:     enter_cs&lock
 1:     sum ← sum + 1   // critical section
 1:     leave_cs&lock
 2: end for
 2: pthread_exit(NULL)
 2procedure MAIN(argc, argv)
 2: pthread_t thread[NUM_THREADS]
 2: pthread_attr_t attr
 2: i, status   // define attribute for joinable threads
 2: pthread_attr_init(&attr)
 2: pthread_attr_setdetachState(&attr, PTHREAD_CREATE_JOINABLE)
    // create threads
 3: for i ← 0 to NUM_THREADS − 1 do
 3:   status ← pthread_create(&thread[i], &attr, threadBody, (void *) i)
 4:   if status then
 4:     perror("pthread_create")
 4:     exit(1)
 5:   end if
 6: end for   // wait all threads to finish
 7: for i ← 0 to NUM_THREADS − 1 do
 7:   status ← pthread_join(thread[i], NULL)
 8:   if status then
 8:     perror("pthread_join")
 8:     exit(1)
 9:   end if
10: end for
10: printf("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS,
    sum)
10: pthread_attr_destroy(&attr)
10: pthread_exit(NULL)
 =0
```

Esse código é efetivo para garantir a exclusão mútua entre processos, pois ele usa uma instrução atômica chamada __sync_fetch_and_or para manipular a variável lock, que indica se a seção crítica está ocupada ou não. Uma instrução atômica é uma operação que é executada de forma indivisível, ou seja, sem interrupções ou interferências de outros processos ou threads1. Assim, a instrução atômica garante que apenas uma thread por vez possa modificar a variável lock, evitando condições de corrida.

Para que o código seja efetivo, ele usa a função enter_cs para verificar se a seção crítica está livre, usando a instrução atômica __sync_fetch_and_or para fazer uma operação lógica OU entre a variável lock e o valor 1. Essa instrução retorna o valor original de lock e armazena o resultado da operação em lock. Assim, se lock for zero, significa que a seção crítica está livre, e a instrução retorna zero e deixa lock como um. Se lock for um, significa que a seção crítica está ocupada, e a instrução retorna um e mantém lock como um. A função enter_cs faz uma espera ocupada (busy waiting) até que a instrução atômica retorne zero, permitindo que a thread entre na seção crítica. Depois de sair da seção crítica, a thread usa a função leave_cs para liberar a seção crítica, atribuindo zero à variável lock.

2.4.1 Saídas

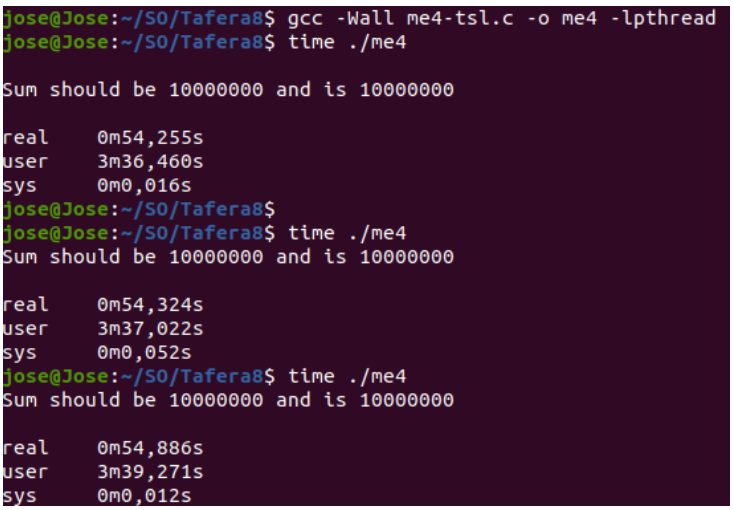

Figura 4: Saídas me4-tsl.c.

2.5 Exercício 5

Algoritmo 5 me5-xchg.c

```
Require: #include <pthread.h>
Require: #include <stdio.h>
Require: #include <stdlib.h>
 0: #define NUM_THREADS 100
 0: #define NUM_STEPS 100000
 0: sum ← 0
 0: lock ← 0   // enter critical section
 0: procedure ENTER_CS(lock)
 0:   key ← 1
 0:   while key do   // XCHG lock, key
 0:     __asm__ __volatile__("xchgl %1, %0":  "=r"(key) :
    "m"(*lock), "0"(key) :  "memory")
      // leave critical section
 0: procedure LEAVE_CS(lock)
 0:   lock ← 0
 0: procedure THREADBODY(id)
 1: for i ← 0 to NUM_STEPS − 1 do
 1:     enter_cs&lock
 1:     sum ← sum + 1   // critical section
 1:     leave_cs&lock
 2: end for
 2: pthread_exit(NULL)
 2procedure MAIN(argc, argv)
 2: pthread_t thread[NUM_THREADS]
 2: pthread_attr_t attr
 2: i, status   // define attribute for joinable threads
 2: pthread_attr_init(&attr)
 2: pthread_attr_setdetachState(&attr, PTHREAD_CREATE_JOINABLE)
    // create threads
 3: for i ← 0 to NUM_THREADS − 1 do
 3:   status ← pthread_create(&thread[i], &attr, threadBody, (void *) i)
 4:   if status then
 4:     perror("pthread_create")
 4:     exit(1)
 5:   end if
 6: end for   // wait all threads to finish
 7: for i ← 0 to NUM_THREADS − 1 do
 7:   status ← pthread_join(thread[i], NULL)
 8:   if status then
 8:     perror("pthread_join")
 8:     exit(1)
 9:   end if
10: end for
10: printf("Sum should be %d and is %d\n", NUM_THREADS*NUM_STEPS,
    sum)
10: pthread_attr_destroy(&attr)
10: pthread_exit(NULL)
=0
```

Esse código é efetivo para garantir a exclusão mútua entre processos, pois ele usa uma instrução atômica chamada XCHG para trocar os valores da variável lock e da variável key. Ele usa a função enter_cs para verificar se a seção crítica está livre, usando a instrução atômica XCHG para trocar os valores da variável lock e da variável key. Essa instrução retorna o valor original de lock e armazena o valor original de key em lock. Assim, se lock for zero, significa que a seção crítica está livre, e a instrução retorna zero e deixa lock como um. Se lock for um, significa que a seção crítica está ocupada, e a instrução retorna um e mantém lock como um. A função enter_cs faz uma espera ocupada (busy waiting) até que a instrução atômica retorne zero, permitindo que a thread entre na seção crítica. Depois de sair da seção crítica, a thread usa a função leave_cs para liberar a seção crítica, atribuindo zero à variável lock.

2.5.1 Saídas

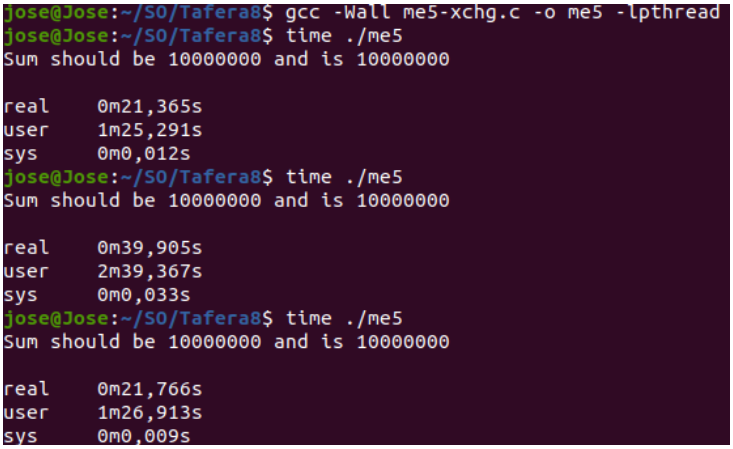

Figura 5: Saídas me5-xchg.c.

2.6 Exercício 6

Algoritmo 6 me6-semaphore.c

```
Require: #include <pthread.h>
Require: #include <stdio.h>
Require: #include <stdlib.h>
Require: #include <semaphore.h>
 0: #define NUM_THREADS 100
 0: #define NUM_STEPS 100000
 0: sum ← 0
 0: s  // semaphore
 0: sem_init(&s, 0, 1)  // initialize semaphore to 1
 0: procedure THREADBODY(id)
 1: for i ← 0 to NUM_STEPS − 1 do
 1:     sem_wait(&s)
 1:     sum ← sum + 1  // critical section
 1:     sem_post(&s)
 2: end for
 2:   pthread_exit(NULL)
 2:
 2:   procedure MAIN(argc, argv)
 2:     pthread_t thread[NUM_THREADS]
 2:     pthread_attr_t attr
 2:     i, status  // define attribute for joinable threads
 2:     pthread_attr_init(&attr)
 2:     pthread_attr_setdetachState(&attr, PTHREAD_CREATE_JOINABLE)
   // create threads
 3: for i ← 0 to NUM_THREADS − 1 do
 3:       status ← pthread_create(&thread[i], &attr, threadBody, (void *) i)
 4:   if status then
 4:         perror("pthread_create")
 4:         exit(1)
 5:   end if
 6: end for  // wait all threads to finish
 7: for i ← 0 to NUM_THREADS − 1 do
 7:       status ← pthread_join(thread[i], NULL)
 8:   if status then
 8:         perror("pthread_join")
 8:         exit(1)
 9:   end if
10: end for
10:     printf("Sum should be %d and is %d\n",
   NUM_THREADS*NUM_STEPS, sum)
10:     pthread_attr_destroy(&attr)
10:     pthread_exit(NULL)
    =0
```

Esse código é efetivo para garantir a exclusão mútua entre processos, pois ele usa um mecanismo de sincronização chamado semáforo, que é uma variável inteira que pode ser incrementada ou decrementada por operações atômicas, chamadas de up e down1. O semáforo pode ser usado para controlar o acesso a um recurso compartilhado, como a variável sum, de forma que se o semáforo vale zero, significa que o recurso está ocupado, e se vale um, significa que o recurso está livre. Assim, antes de entrar na seção crítica, a thread deve fazer um down no semáforo, e depois de sair, deve fazer um up no semáforo. Dessa forma, se o semáforo estiver zero, a thread terá que esperar até que ele fique um, e se estiver um, a thread poderá entrar na seção crítica e deixar o semáforo zero.

2.6.1 Saídas

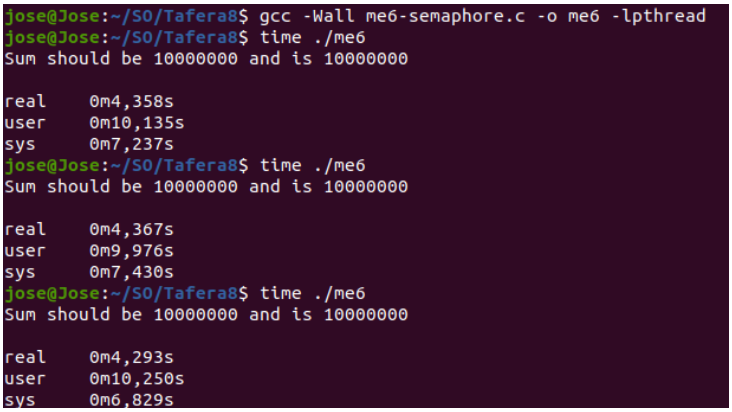

Figura 6: Saídas me6-semaphore.c.

2.7 Exercício 7

Algoritmo 7 me7-mutex.c

```
Require: #include <pthread.h>
Require: #include <stdio.h>
Require: #include <stdlib.h>
 0: #define NUM_THREADS 100
 0: #define NUM_STEPS 100000
 0: sum ← 0
 0: mutex   // pthread mutex
 0: pthread_mutex_init(&mutex, NULL)   // initialize mutex
 0: procedure THREADBODY(id)
 1: for i ← 0 to NUM_STEPS − 1 do
 1:     pthread_mutex_lock(&mutex)
 1:     sum ← sum + 1   // critical section
 1:     pthread_mutex_unlock(&mutex)
 2: end for
 2:   pthread_exit(NULL)
 2:
 2:   procedure MAIN(argc, argv)
 2:     pthread_t thread[NUM_THREADS]
 2:     pthread_attr_t attr
 2:     i, status   // define attribute for joinable threads
 2:     pthread_attr_init(&attr)
 2:     pthread_attr_setdetachState(&attr, PTHREAD_CREATE_JOINABLE)
   // create threads
 3: for i ← 0 to NUM_THREADS − 1 do
 3:       status ← pthread_create(&thread[i], &attr, threadBody, (void *) i)
 4:   if status then
 4:         perror("pthread_create")
 4:         exit(1)
 5:   end if
 6: end for   // wait all threads to finish
 7: for i ← 0 to NUM_THREADS − 1 do
 7:       status ← pthread_join(thread[i], NULL)
 8:   if status then
 8:         perror("pthread_join")
 8:         exit(1)
 9:   end if
10: end for
10:     printf("Sum should be %d and is %d\n",
   NUM_THREADS*NUM_STEPS, sum)
10:     pthread_attr_destroy(&attr)
10:     pthread_exit(NULL)
    =0
```

Esse código é efetivo para garantir a exclusão mútua entre processos, pois ele usa um mecanismo de sincronização chamado mutex, que é um tipo especial de semáforo binário que pode ser bloqueado e desbloqueado apenas pelo mesmo processo ou thread1. O mutex pode ser usado para controlar o acesso a um recurso compartilhado, como a variável sum, de forma que se o mutex estiver bloqueado, significa que o recurso está ocupado, e se estiver desbloqueado, significa que o recurso está livre. Assim, antes de entrar na seção crítica, a thread deve bloquear o mutex, e depois de sair, deve desbloquear o mutex. Dessa forma, se o mutex estiver bloqueado, a thread terá que esperar até que ele fique desbloqueado, e se estiver desbloqueado, a thread poderá entrar na seção crítica e bloquear o mutex.

2.7.1 Saídas

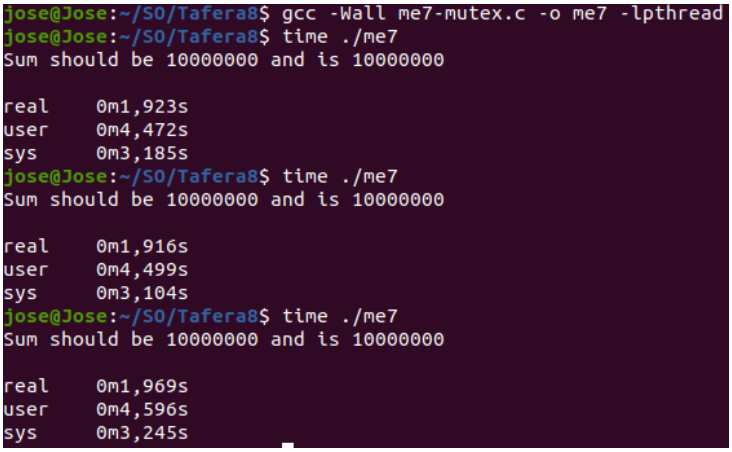

Figura 7: Saídas me7-mutex.c.