

Universidade Tecnológica Federal do Paraná - UTFPR
Disciplina: Sistemas Operacionais
Prof. Fernando Luiz Copetti

Relatório da Prática 8
Exclusão Mútua

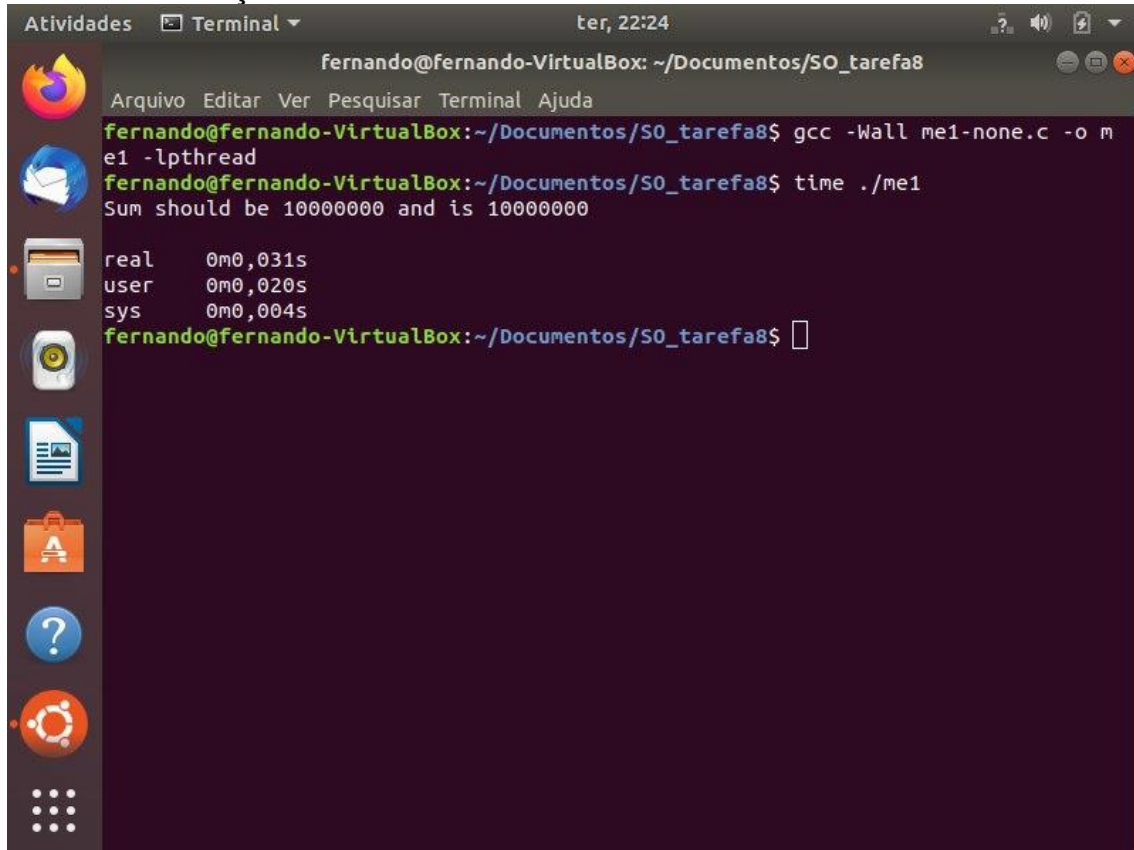
Fernando Itiro– 2137380

Curitiba
16 de junho de 2023

Introdução

Exclusão mútua em sistemas operacionais refere-se a um conceito fundamental que garante que recursos compartilhados, como memória, arquivos ou dispositivos, sejam acessados por processos ou threads de maneira ordenada e sem interferências. O objetivo é evitar condições de corrida, onde vários processos tentam acessar um recurso ao mesmo tempo, resultando em resultados inconsistentes ou incorretos.

1 Sem Coordenação



```
fernando@fernando-VirtualBox: ~/Documentos/SO_tarefa8
Arquivo Editar Ver Pesquisar Terminal Ajuda
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ gcc -Wall me1-none.c -o me1 -lpthread
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ time ./me1
Sum should be 10000000 and is 10000000

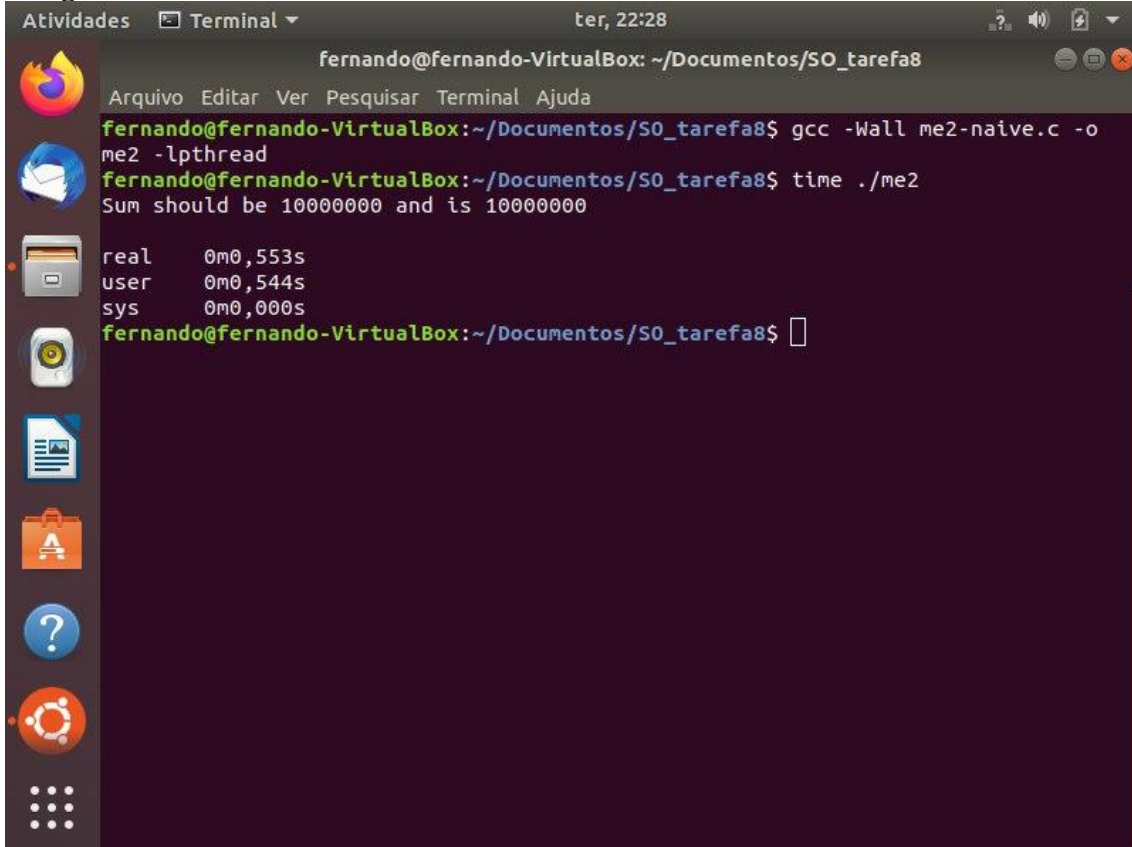
real    0m0,031s
user    0m0,020s
sys     0m0,004s
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$
```

Cria-se um número definido de threads e cada uma dessas threads incrementa uma variável chamada "sum" um determinado número de vezes. A seção crítica, onde a variável "sum" é atualizada, não possui proteção contra acesso simultâneo pelas threads.

O programa cria um total de 100 threads e cada uma delas executa um loop de 100.000 iterações, incrementando a variável "sum" em 1 a cada iteração. No entanto, como não há exclusão mútua implementada, várias threads podem acessar e modificar a variável "sum" simultaneamente, resultando em um comportamento indefinido.

Após todas as threads terem concluído sua execução, o programa exibe o valor esperado para a variável "sum" (calculado como o número de threads multiplicado pelo número de iterações) e o valor atual da variável "sum". No entanto, devido à falta de exclusão mútua, o valor atual pode diferir do valor esperado, demonstrando um problema de condição de corrida.

2 Ingênuo



```
fernando@fernando-VirtualBox: ~/Documentos/SO_tarefa8
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ gcc -Wall me2-naive.c -o me2 -lpthread
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ time ./me2
Sum should be 10000000 and is 10000000

real    0m0,553s
user    0m0,544s
sys     0m0,000s
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$
```

Este código também trata do acesso concorrente a uma variável por várias threads, mas desta vez apresenta uma solução "ingênuo" para lidar com a exclusão mútua.

No entanto, a abordagem utilizada aqui para garantir a exclusão mútua é conhecida como "busy waiting" (espera ocupada). Duas funções, `enter_cs()` e `leave_cs()`, são implementadas para controlar a entrada e saída da seção crítica, respectivamente.

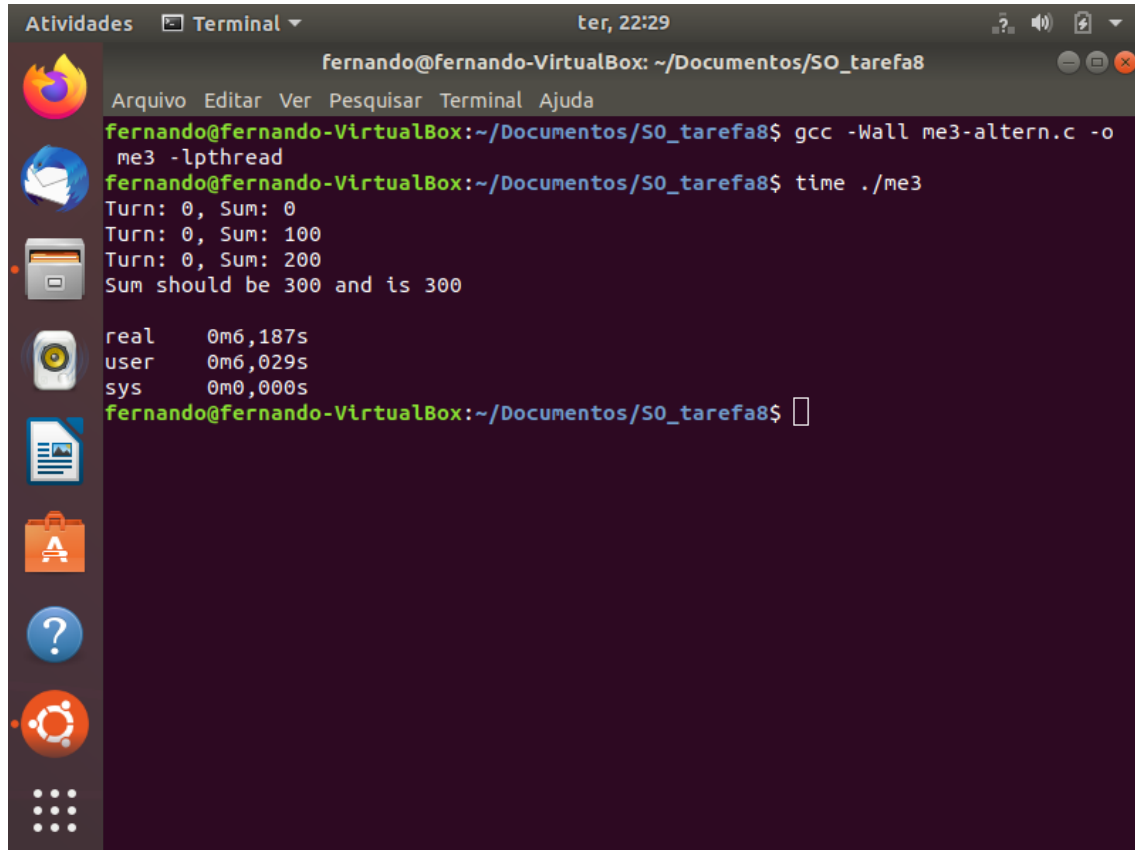
Na função `enter_cs()`, há um loop de espera ocupada (`while (busy)`) que verifica constantemente o valor da variável `busy`. Enquanto `busy` estiver definido como 1, o loop continuará executando, o que significa que a seção crítica está sendo utilizada por outra thread. Assim que `busy` for definido como 0, indicando que a seção crítica está livre, a função `enter_cs()` sai do loop e define `busy` como 1.

Já na função `leave_cs()`, a variável `busy` é simplesmente definida como 0, indicando que a seção crítica está sendo liberada e está disponível para outras threads.

Embora essa solução possa parecer funcionar em alguns casos, ela possui algumas desvantagens significativas. A principal delas é o fato de que, durante a espera ocupada, a CPU está sendo utilizada de forma ineficiente, pois a thread continua verificando constantemente o valor da variável `busy` em vez de esperar passivamente.

Além disso, essa abordagem não garante uma ordem justa de acesso à seção crítica. Pode ocorrer uma situação em que uma thread monopolize o acesso, resultando em outras threads ficando bloqueadas indefinidamente.

3 Alternância



```
fernando@fernando-VirtualBox: ~/Documentos/SO_tarefa8
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ gcc -Wall me3-altern.c -o me3 -lpthread
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ time ./me3
Turn: 0, Sum: 0
Turn: 0, Sum: 100
Turn: 0, Sum: 200
Sum should be 300 and is 300

real    0m6,187s
user    0m6,029s
sys     0m0,000s
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$
```

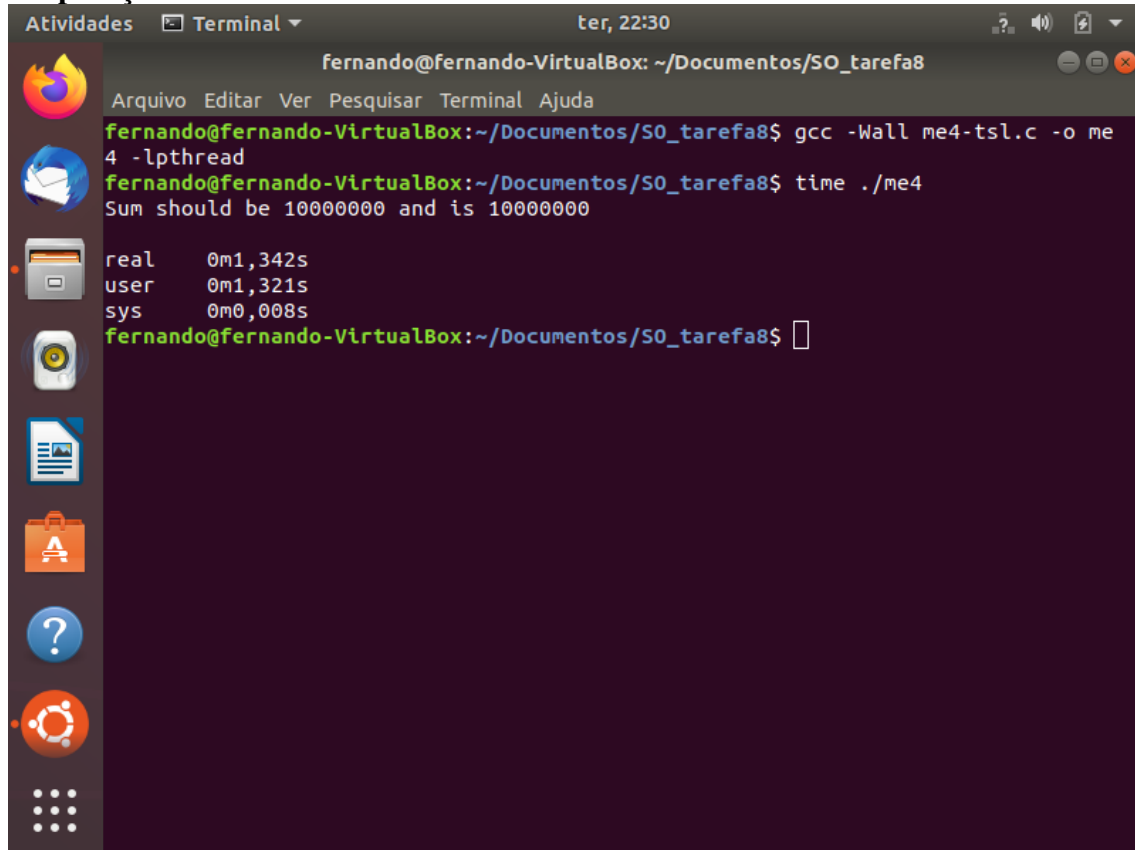
Este código apresenta uma solução com alternância para o acesso concorrente a uma variável por várias threads. O objetivo é garantir a exclusão mútua por meio da alternância entre as threads para acessar a seção crítica. Novamente, o código cria um número definido de threads e cada uma delas incrementa a variável "sum" um determinado número de vezes.

A função `enter_cs()` é responsável por controlar a entrada na seção crítica. Ela possui um loop de espera ocupada que verifica constantemente o valor da variável "turn". Cada thread possui um ID único, e a variável "turn" é usada para determinar qual thread pode acessar a seção crítica. Enquanto "turn" não for igual ao ID da thread atual, a thread permanecerá em espera ocupada. Quando "turn" for igual ao ID da thread atual, a thread poderá entrar na seção crítica.

A função `leave_cs()` é responsável por permitir que a próxima thread entre na seção crítica, alternando o valor da variável "turn" para o próximo ID de thread.

Durante a execução, a função `enter_cs()` imprime uma mensagem indicando o ID da thread atual e o valor atual da variável "sum" quando a condição `sum % 100 == 0` for satisfeita. Isso é apenas uma ilustração para mostrar que as threads estão alternando o acesso à seção crítica.

4 Operações Atômicas



```
fernando@fernando-VirtualBox: ~/Documentos/SO_tarefa8
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ gcc -Wall me4-tsl.c -o me4 -lpthread
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ time ./me4
Sum should be 10000000 and is 10000000

real    0m1,342s
user    0m1,321s
sys     0m0,008s
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$
```

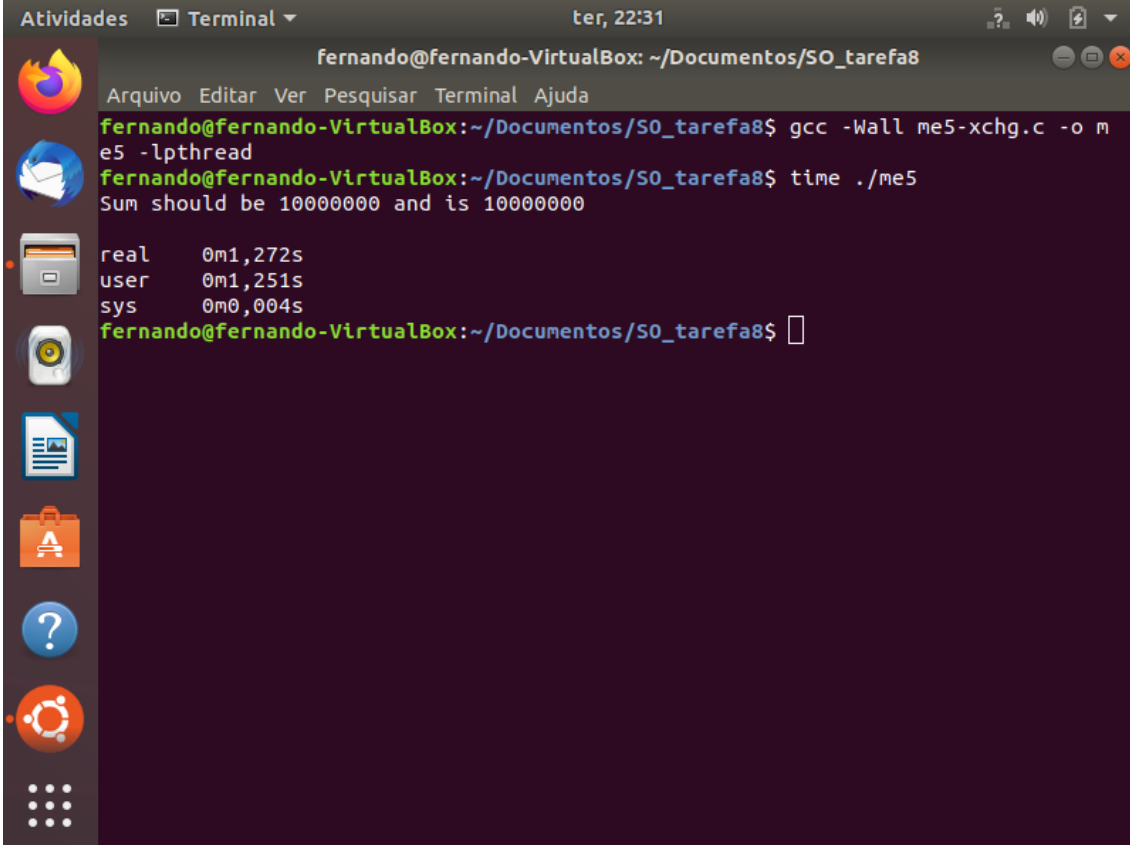
Este código apresenta uma solução para o acesso concorrente a uma variável por várias threads utilizando a operação TSL (Test-and-Set Lock). O objetivo é garantir a exclusão mútua por meio da operação atômica TSL. Novamente, o código cria um número definido de threads e cada uma delas incrementa a variável "sum" um determinado número de vezes.

A função `enter_cs()` é responsável por controlar a entrada na seção crítica. Ela utiliza a operação `__sync_fetch_and_or()` para realizar uma operação de teste-e-definição-atômica no valor da variável "lock". Enquanto o valor retornado pela operação for diferente de zero (indicando que o bloqueio está ativo), a thread permanecerá em espera ocupada. Assim que a operação `__sync_fetch_and_or()` definir o valor de "lock" como 1, a thread poderá entrar na seção crítica.

A função `leave_cs()` é responsável por liberar o bloqueio, definindo o valor de "lock" como 0.

Durante a execução, cada thread executa seu loop para incrementar a variável "sum" na seção crítica protegida pelas chamadas das funções `enter_cs()` e `leave_cs()`.

5 XCHG



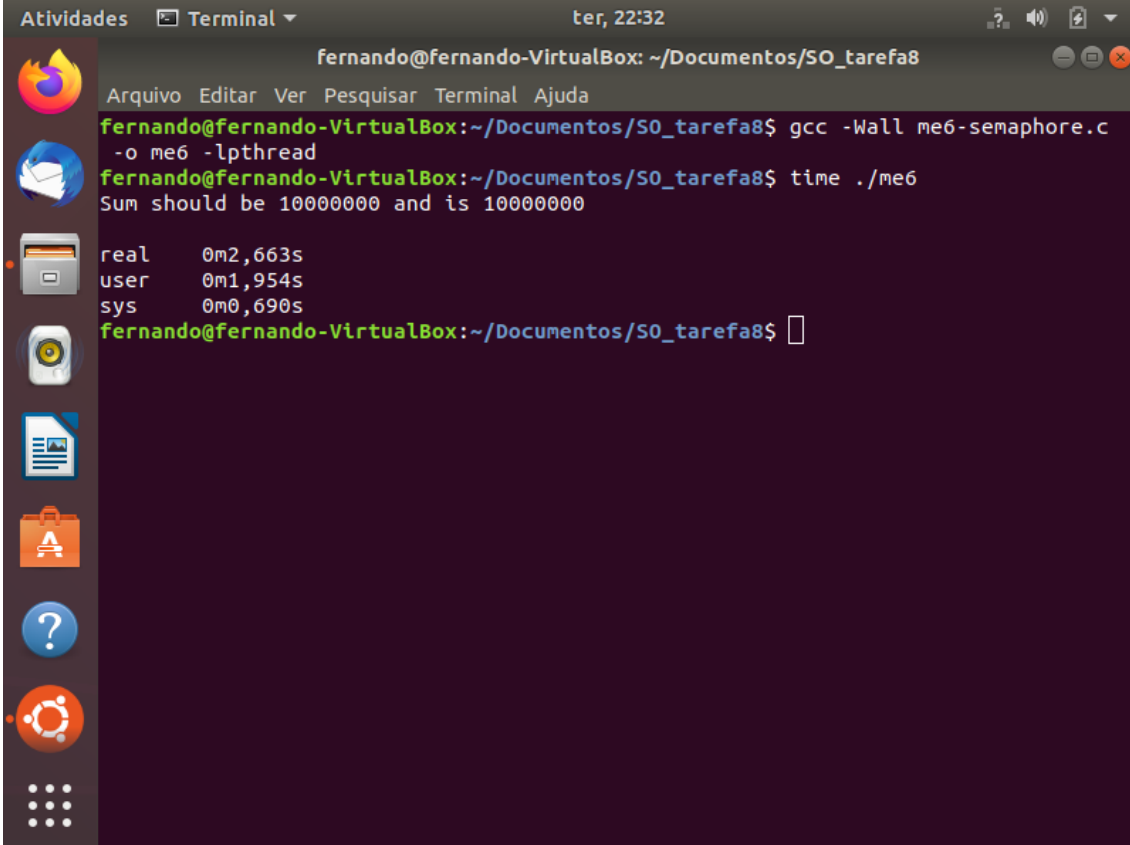
```
fernando@fernando-VirtualBox: ~/Documentos/SO_tarefa8
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ gcc -Wall me5-xchg.c -o me5 -lpthread
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ time ./me5
Sum should be 10000000 and is 10000000

real    0m1.272s
user    0m1.251s
sys     0m0.004s
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$
```

Este código apresenta uma solução para o acesso concorrente a uma variável por várias threads utilizando a instrução XCHG (Exchange). O objetivo é garantir a exclusão mútua por meio dessa instrução de troca atômica. Novamente, o código cria um número definido de threads e cada uma delas incrementa a variável "sum" um determinado número de vezes.

A função `enter_cs()` é responsável por controlar a entrada na seção crítica. Ela utiliza a instrução XCHG para realizar uma troca atômica entre o valor da variável "lock" e uma variável local "key". Enquanto o valor de "key" for diferente de zero (indicando que o bloqueio está ativo), a thread permanecerá em espera ocupada. Assim que a instrução XCHG trocar o valor de "lock" com o valor de "key" e armazenar o valor antigo de "lock" em "key", a thread poderá entrar na seção crítica.

6 Semáforos



```
fernando@fernando-VirtualBox: ~/Documentos/SO_tarefa8
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ gcc -Wall me6-semaphore.c
-o me6 -lpthread
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ time ./me6
Sum should be 10000000 and is 10000000

real    0m2,663s
user    0m1,954s
sys     0m0,690s
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$
```

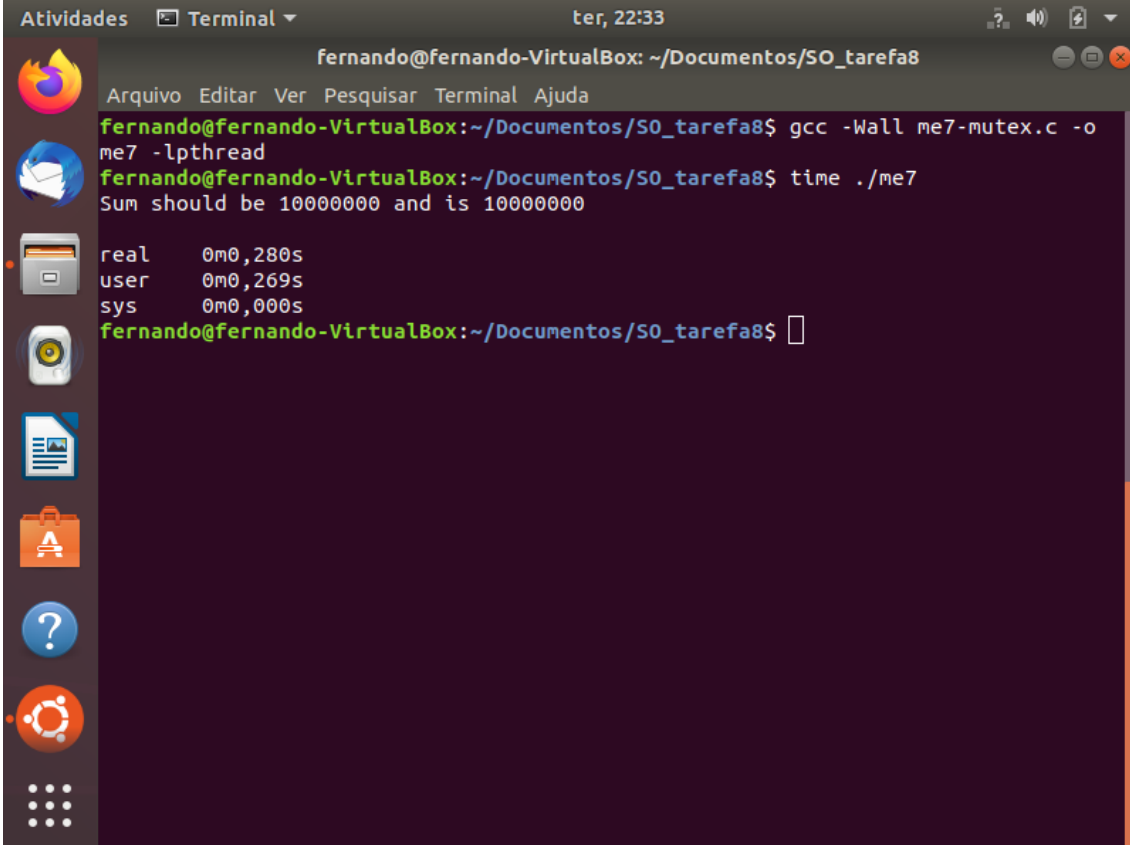
Este código apresenta uma solução para o acesso concorrente a uma variável por várias threads utilizando semáforos. O objetivo é garantir a exclusão mútua por meio do uso de semáforos.

O programa cria um número definido de threads, cada uma das quais incrementa a variável "sum" um determinado número de vezes. Um semáforo chamado "s" é utilizado para controlar o acesso à seção crítica.

A função `threadBody()` é executada por cada thread e contém o loop responsável por incrementar a variável "sum". Antes de entrar na seção crítica, a thread chama a função `sem_wait(&s)`, que aguarda até que o semáforo "s" tenha um valor maior que 0. Isso garante que somente uma thread possa entrar na seção crítica ao mesmo tempo. Após a execução da seção crítica, a thread chama `sem_post(&s)` para liberar o semáforo e permitir que outras threads possam entrar na seção crítica.

No `main()`, o semáforo é inicializado com o valor 1 através da função `sem_init(&s, 0, 1)`. Em seguida, as threads são criadas e executadas. Após todas as threads concluírem suas execuções, o programa exibe o valor esperado para a variável "sum" (calculado como o número de threads multiplicado pelo número de iterações) e o valor atual da variável "sum".

7 Mutex



The screenshot shows a terminal window titled "fernando@fernando-VirtualBox: ~/Documentos/SO_tarefa8". The terminal displays the following commands and output:

```
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ gcc -Wall me7-mutex.c -o me7 -lpthread
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$ time ./me7
Sum should be 10000000 and is 10000000

real    0m0,280s
user    0m0,269s
sys     0m0,000s
fernando@fernando-VirtualBox:~/Documentos/SO_tarefa8$
```

Este código apresenta uma solução para o acesso concorrente a uma variável por várias threads utilizando mutex. O objetivo é garantir a exclusão mútua por meio do uso de mutex.

O programa cria um número definido de threads, cada uma das quais incrementa a variável "sum" um determinado número de vezes. Um mutex chamado "mutex" é utilizado para controlar o acesso à seção crítica.

A função `threadBody()` é executada por cada thread e contém o loop responsável por incrementar a variável "sum". Antes de entrar na seção crítica, a thread chama a função `pthread_mutex_lock(&mutex)`, que bloqueia o mutex e impede que outras threads acessem a seção crítica. Após a execução da seção crítica, a thread chama `pthread_mutex_unlock(&mutex)` para desbloquear o mutex e permitir que outras threads possam acessar a seção crítica.

No `main()`, o mutex é inicializado através da função `pthread_mutex_init(&mutex, NULL)`. Em seguida, as threads são criadas e executadas. Após todas as threads concluírem suas execuções, o programa exibe o valor esperado para a variável "sum" (calculado como o número de threads multiplicado pelo número de iterações) e o valor atual da variável "sum".

Conclusões

Solução "Ingênua" (me2-naive.c): Essa solução utiliza uma abordagem de espera ocupada (busy waiting), em que cada thread aguarda em um loop até que a variável "busy" esteja livre para ser acessada. Essa abordagem não é eficiente, pois as threads consomem tempo de CPU desnecessariamente enquanto esperam. É considerada a pior solução entre as apresentadas.

Solução com Mutex (me7-mutex.c): Essa solução utiliza um mutex para garantir a exclusão mútua. Um mutex é uma estrutura de dados que permite que apenas uma thread por vez acesse a seção crítica. Essa solução é amplamente usada e recomendada devido à sua simplicidade e eficiência. Os mutexes são otimizados para diferentes plataformas e geralmente oferecem um bom desempenho.

Em geral, a solução com mutex (me7-mutex.c) é considerada uma abordagem robusta e eficiente para garantir a exclusão mútua em um ambiente de threads.