**Universidade Federal de Roraima**

Rua, Av. Cap. Ene Garcês, 2413 - Aeroporto,

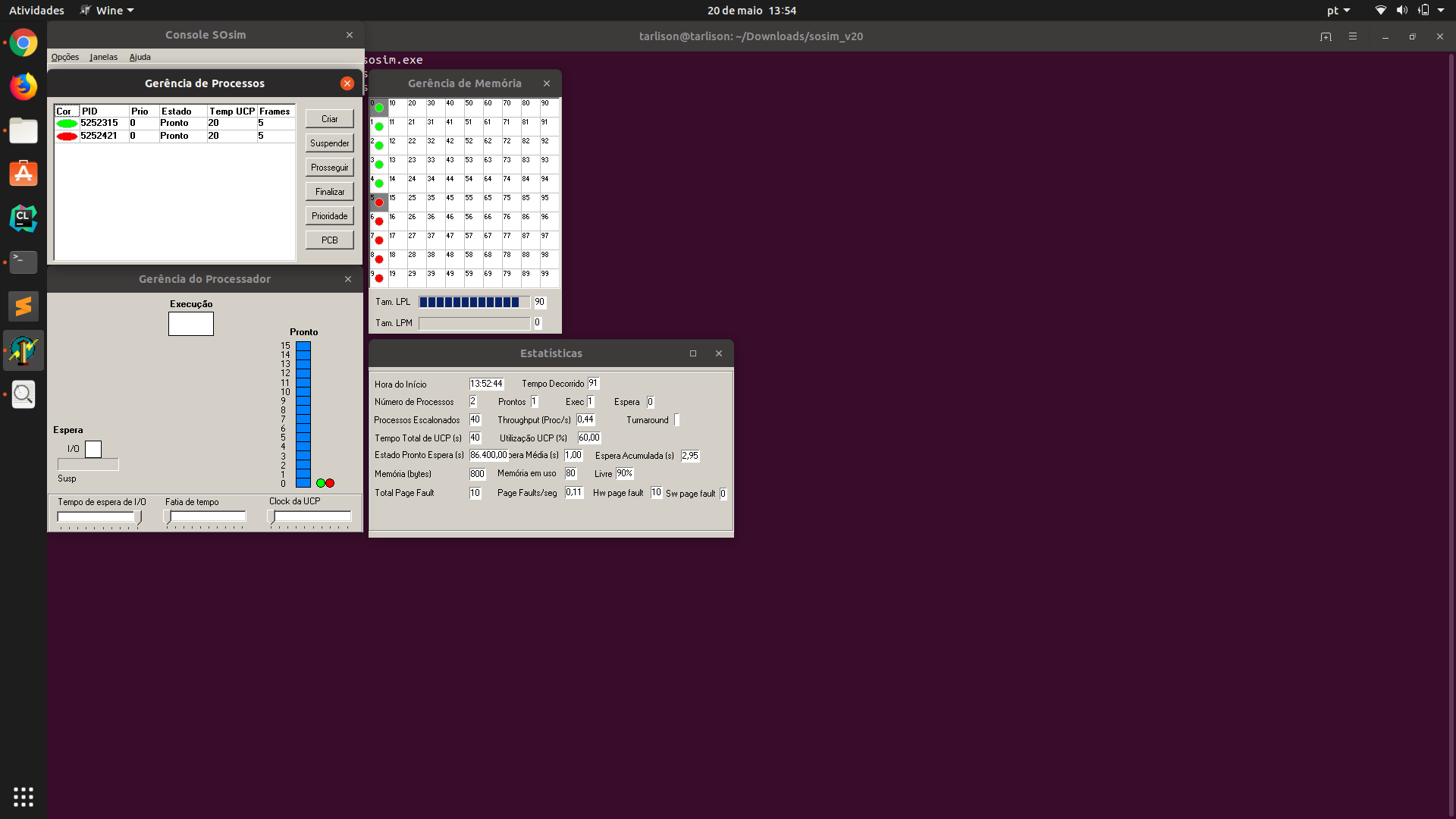
Boa Vista - RR, 69310-000

Aluno: Tarlison Sander Lima Brito

Matrícula: 2017013008

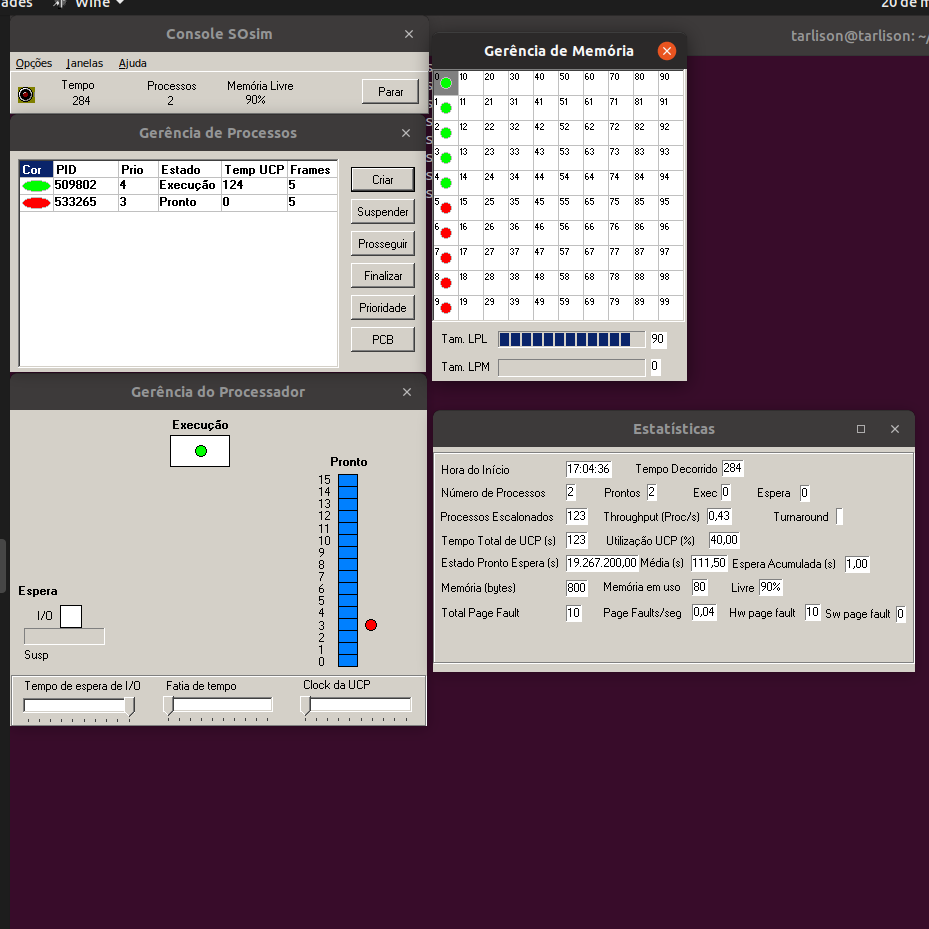
Laboratório de Sistemas Operacionais

**23 de maio de 2019**

**[Questão 1]:** **(Prática - A)**

Porque os processos estão sendo escalonados através de escalonamento circular e assim eles possuem uma ordem para serem executados. E esse tipo de escalonamento é organizado de uma maneira que cada um deles possua um determinado tempo da CPU e caso um desses processos não termine dentro do seu tempo ele é colocado no fim da fila e outro tempo é dado para o processo no começo da fila.

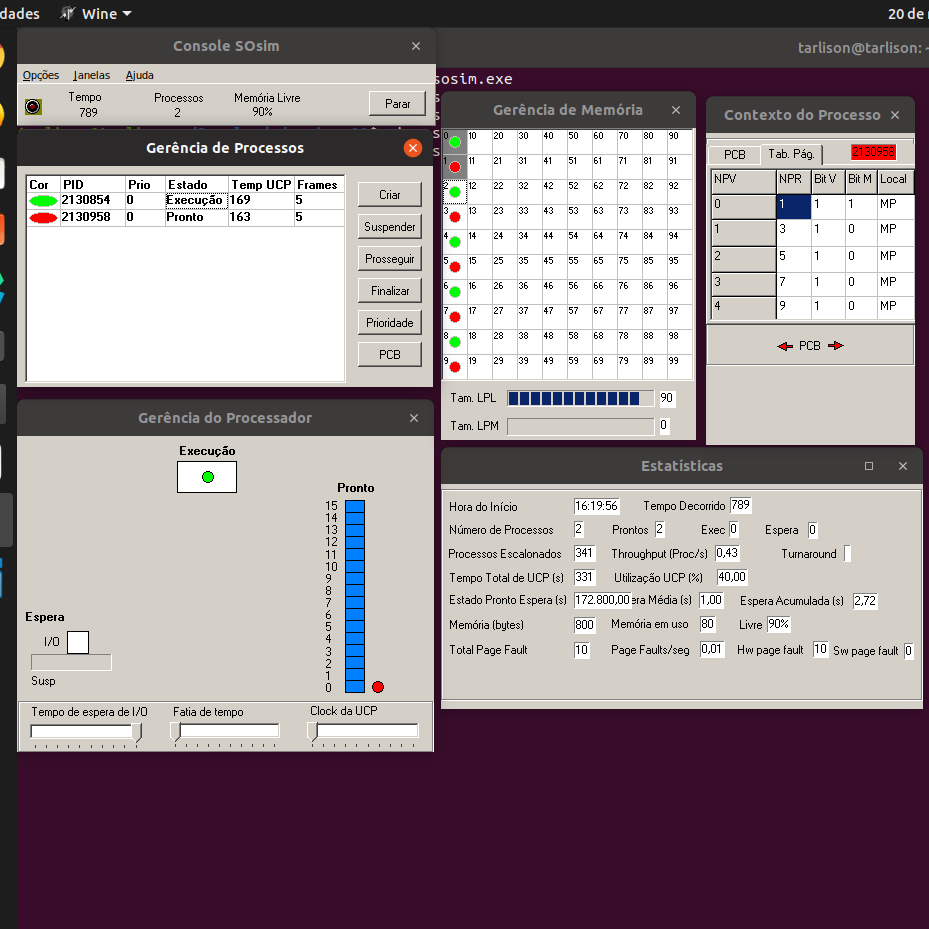
Com isso podemos ver na imagem em que os dois processos estão prontos e apenas esperando serem escalonados que o próximo processo será o verde, observável a partir da aba gerência de processos, pois é o tempo e a vez dele de ser executado e ele apenas está esperando o delay que ocorre até ele começar executar.

**(Prática - B)**

O starvation ocorre pois os processos possuem prioridades pré-determinadas e quando o escalonador vai setar o processo ele escolhe o que possui maior prioridade e assim o processo de prioridade 4 sempre irá executar na frente do processo de prioridade 3 fazendo com que esse processo nunca seja executado.

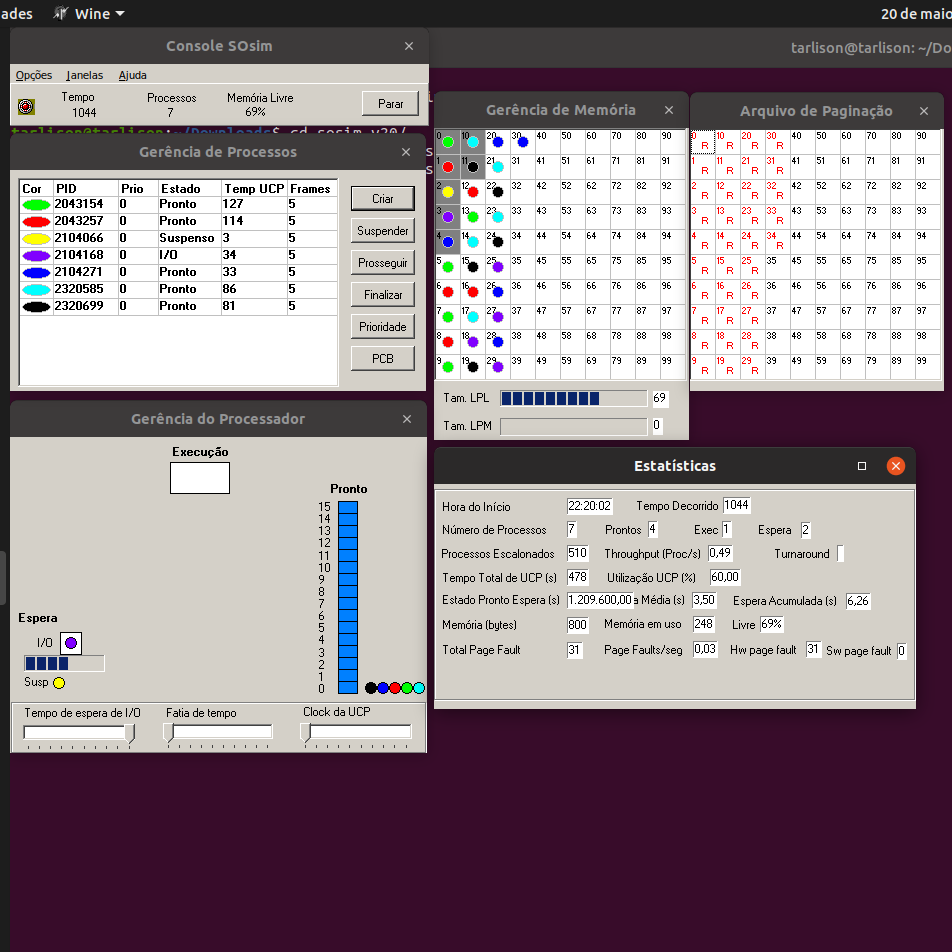
Uma solução seria a criação de uma fila que fosse feita a partir dos pedidos dos processos para alocar os recursos e assim sempre que um processo desejar um recurso o pedido é colocado no final da fila associada a ela e assim que o recurso for liberado o sistema seleciona o primeiro processo da fila. Ou seja, esquema FIFO.

Outra solução seria diminuir a prioridade do processo que está tomando a CPU ou até mesmo excluí-lo se possível para que os outros processos possam ser executados.

**(Prática - C)**

O espaço de endereçamento real máximo do processo será a quantidade de espaço da memória principal e virtual juntas. Já o espaço de endereçamento real mínimo do processo será o tamanho mínimo da tabela de mapeamento carregada.

O tamanho da página virtual irá variar de acordo com o processador utilizado e também da arquitetura de hardware, em algumas arquiteturas é possível que esse tamanho seja configurado.

**(Prática - D)**

É analisado qual dos processos que serão alocados que não possuem uma previsão de utilizar a CPU nos próximos instantes e assim ele manda esse processo para o arquivo de paginação.

Para que o processo seja transferido de volta para a memória principal é usado a política por demanda a qual uma página somente será carregada quando for referenciada assim ele irá levar para a memória principal somente as que forem realmente necessárias.

**[Questão 2]:**

O algoritmo funciona para solucionar um impasse explicado em que um banqueiro de uma pequena cidade pode negociar com um grupo de clientes para os quais ele libera linhas de crédito. E esse algoritmo verifica se a liberação de uma requisição é capaz de levar a um estado inseguro.

É possível verificar se um estado é seguro ou não observando esta tabela abaixo(único recurso). Estão disponíveis 7 créditos para serem usados pelos clientes e o banqueiro precisa gerenciar isso.

Esse estado mostrado na tabela é seguro, pois, com essa quantidade de recursos disponíveis ele pode dar a um dos clientes a quantidade que ele precisa para atingir o crédito máximo e assim que ele terminar de usar os créditos eles são devolvidos e assim os outros podem usá-lo e desta forma todos conseguem obter o crédito.

Caso o crédito disponível fosse, por exemplo, 3 não seria possível ele administrar isso para que todos pudessem usar, considerado assim um caso inseguro. E por fim caso a requisição do cliente seja dada como insegura ela será negada, caso contrário o recurso será liberado.

|  |  |  |
| --- | --- | --- |
| Cliente A | Quantidade de crédito | Crédito máximo |
| A | 0 | 6 |
| B | 0 | 4 |
| C | 0 | 5 |

Disponível: 7

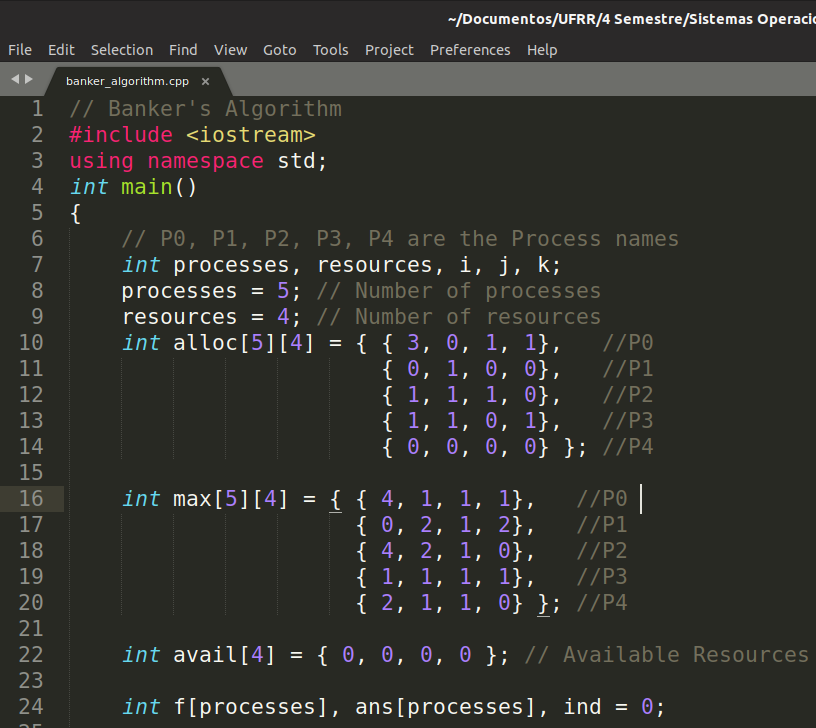
Já quando se trata de múltiplos recursos temos duas matrizes para analisar e obviamente vários recursos para administrar.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | 3 | 0 | 1 | 1 | X | A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 0 | 0 | X | B | 0 | 1 | 1 | 2 |
| C | 1 | 1 | 1 | 0 | X | C | 3 | 1 | 0 | 0 |
| D | 1 | 1 | 0 | 1 | X | D | 0 | 0 | 1 | 0 |
| E | 0 | 0 | 0 | 0 | X | E | 2 | 1 | 1 | 0 |

E = [6,3,4,2], P = [5,3,2,2], A = [1,0,2,0]

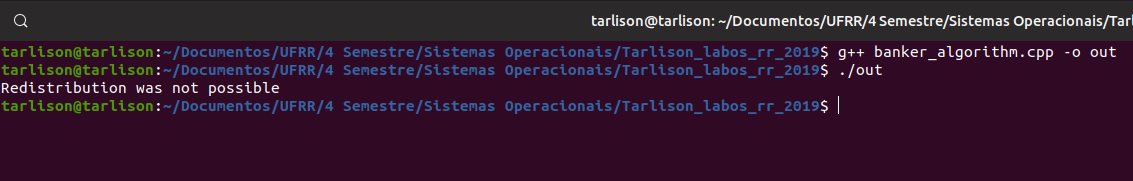
Observando a tabela acima, a tabela do lado esquerdo indica os recursos alocados e a tabela do lado direito os recursos ainda necessários para que cada cliente atinja o máximo. O vetor E indica os recursos alocados, P os recursos disponíveis e A os recursos disponíveis que é a diferença entre o que o banqueiro tem e o que está sendo usado pelos clientes.

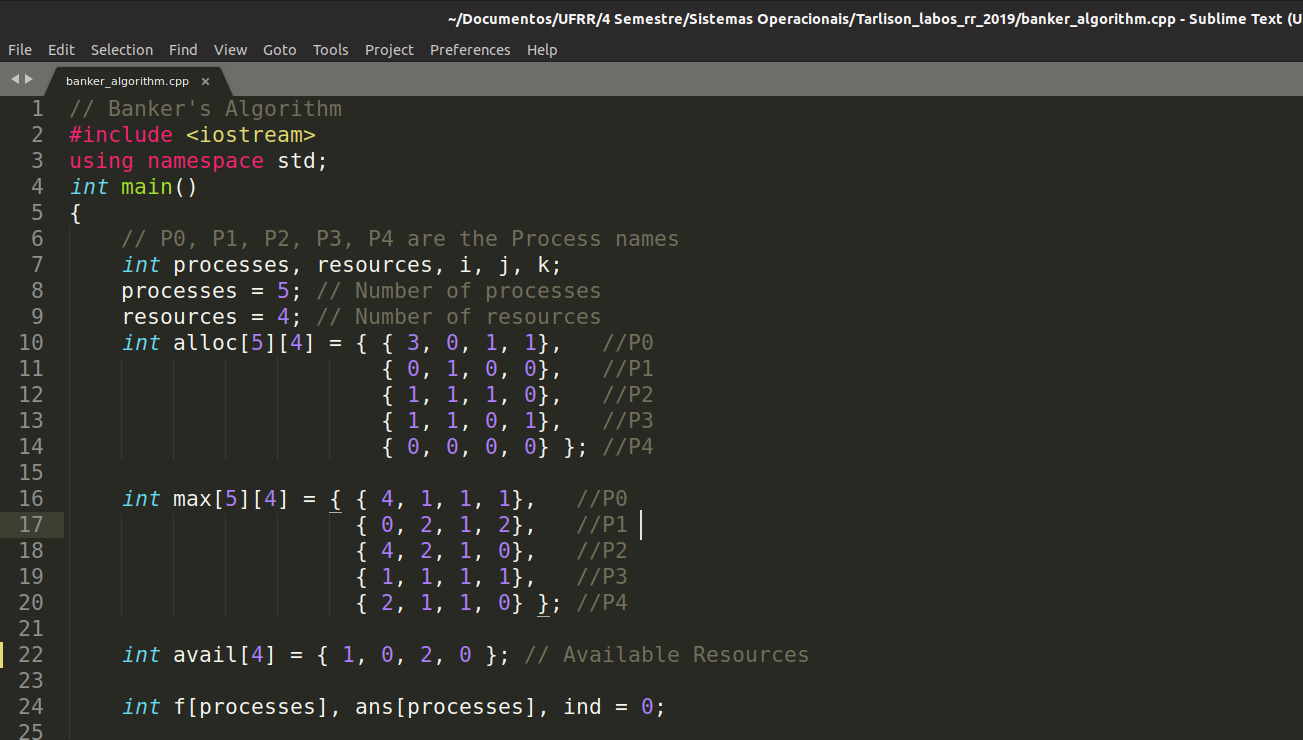
Considerando que o processo da linha escolhida requisita todos os recursos de que precisa e termina, é marcado como terminado e é adicionado ao vetor A todos os recursos que lhe pertenciam. Caso nenhum das linhas da tabela ao lado direito possua uma necessidade de recursos inferiores ou iguais a A, resultará em uma situação de impasse já que nenhum dos clientes poderá obter todos os créditos. E esses passos se repetem até que todos os clientes sejam atendidos ou até que haja um impasse.

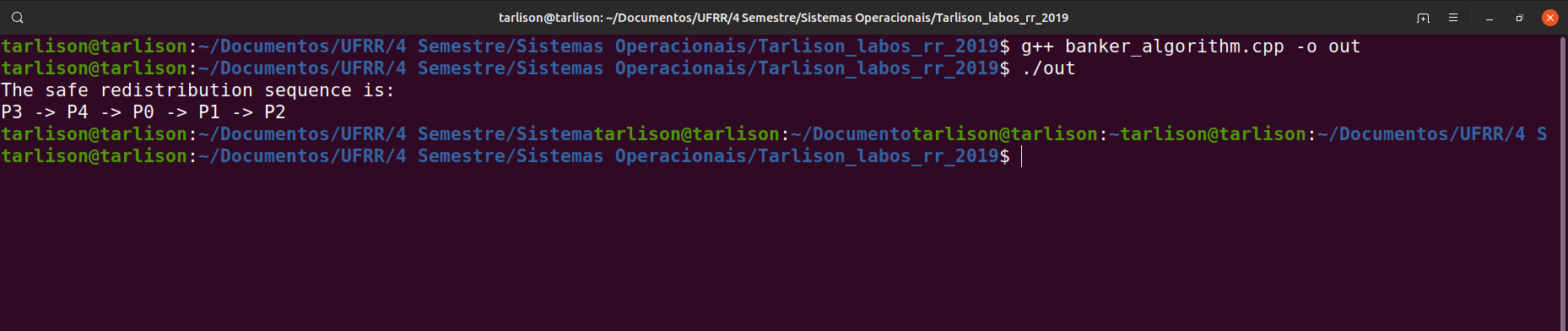


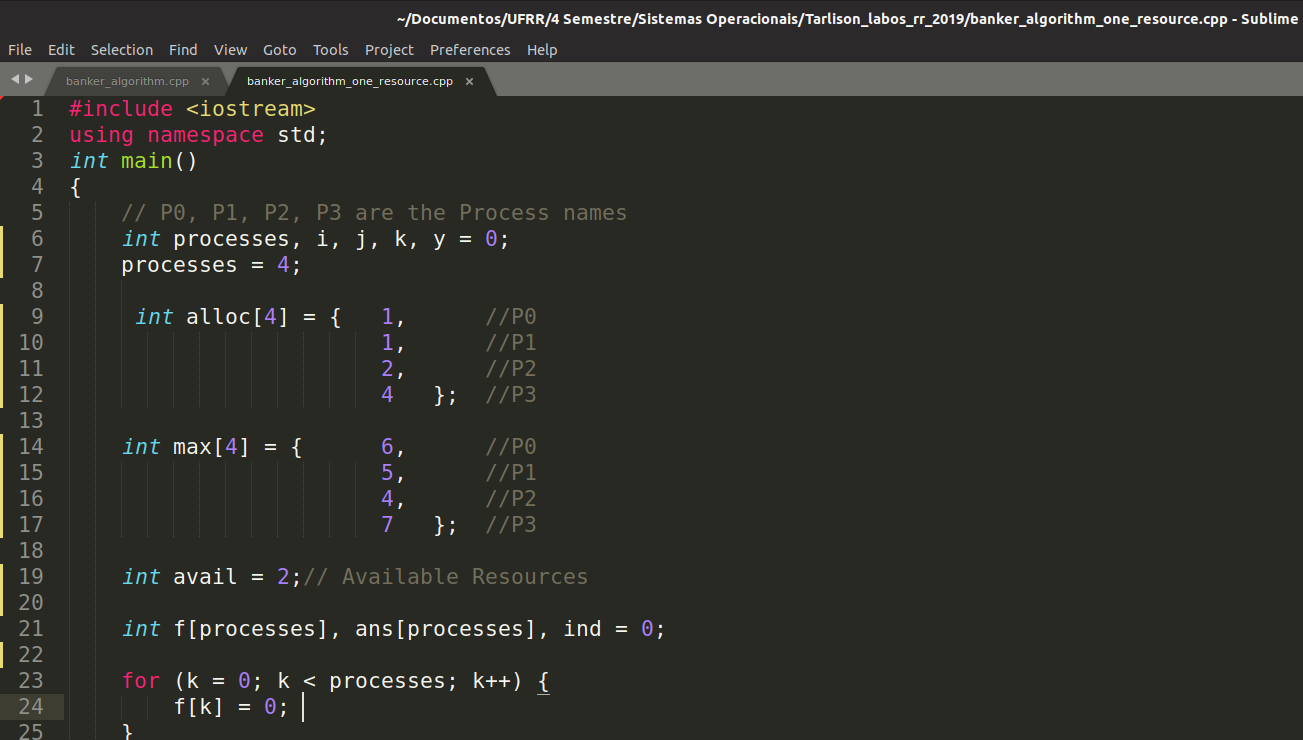
Aqui podemos ver os vetores utilizados no algoritmo do banqueiro para múltiplos recursos. O vetor alloc representa os recursos que estão sendo usados pelo processos (que são representados por P0, P1, P2,P3, P4), o vetor max os recursos máximos de cada processo e o vetor avail os recursos ainda disponíveis para serem distribuídos entre os processos.

Nesse caso a nossa saída será “Redistribution was not possible” pois a quantidade de recursos disponíveis é 0 (vetor avail). Imagem da saída:

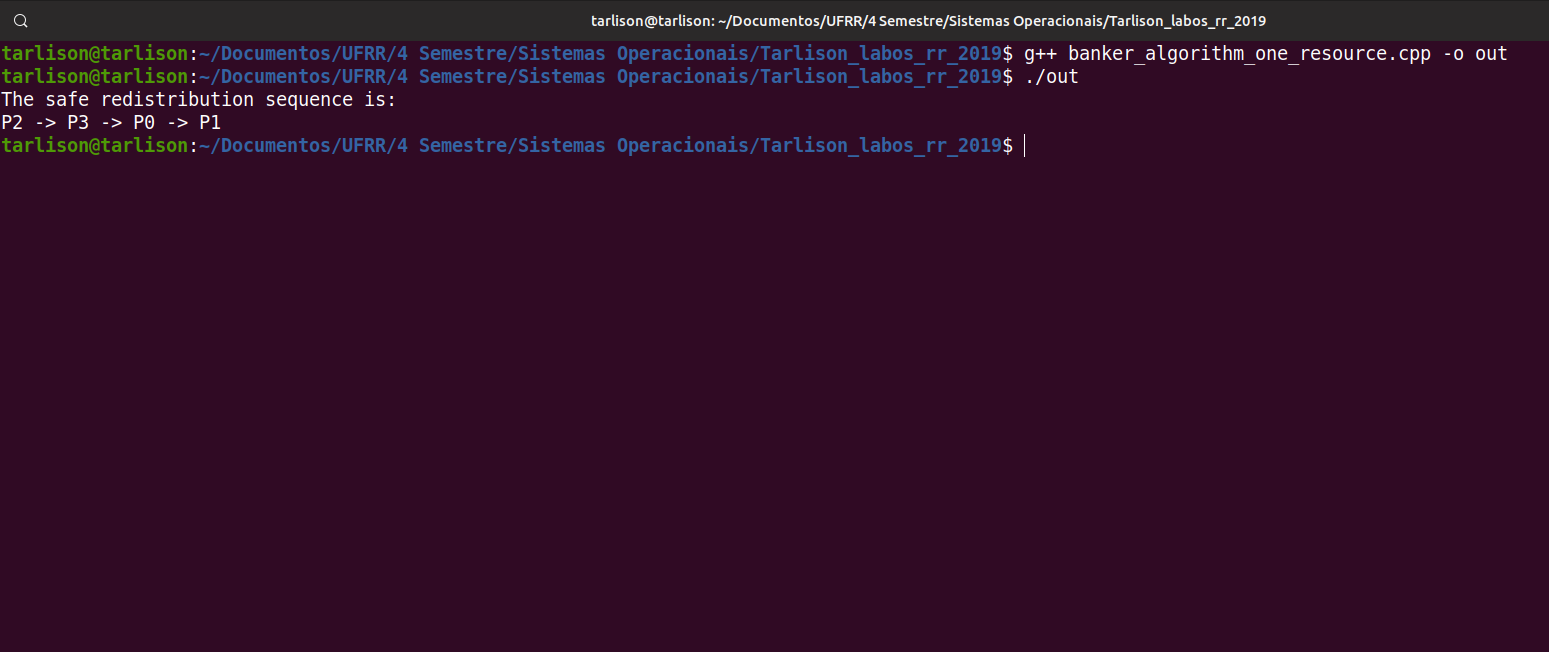


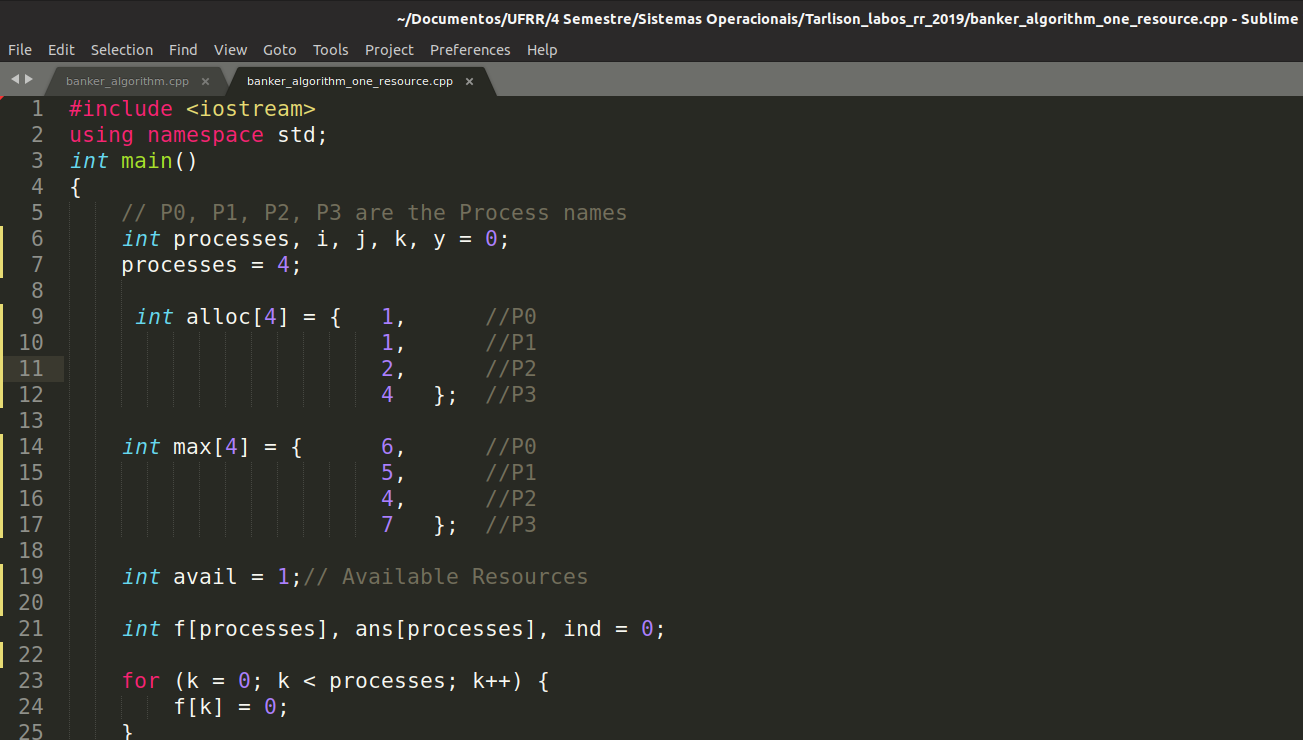
Agora no caso da imagem abaixo nós temos recursos disponíveis e a matriz de recursos alocados iguais ao exemplo dado no livro Sistemas operacionais modernos, 3ª edição do Andrew S. Tanenbaum página 281. Com esses recursos agora será possível fazer a distribuição de recursos entre todos sem que ocorra erros. Trecho do código modificado:

Irá gerar esta saída que significa a ordem que será preciso fazer a distribuição para que não ocorra erros (no caso P3 -> P4 -> P0 -> P1 -> P2):

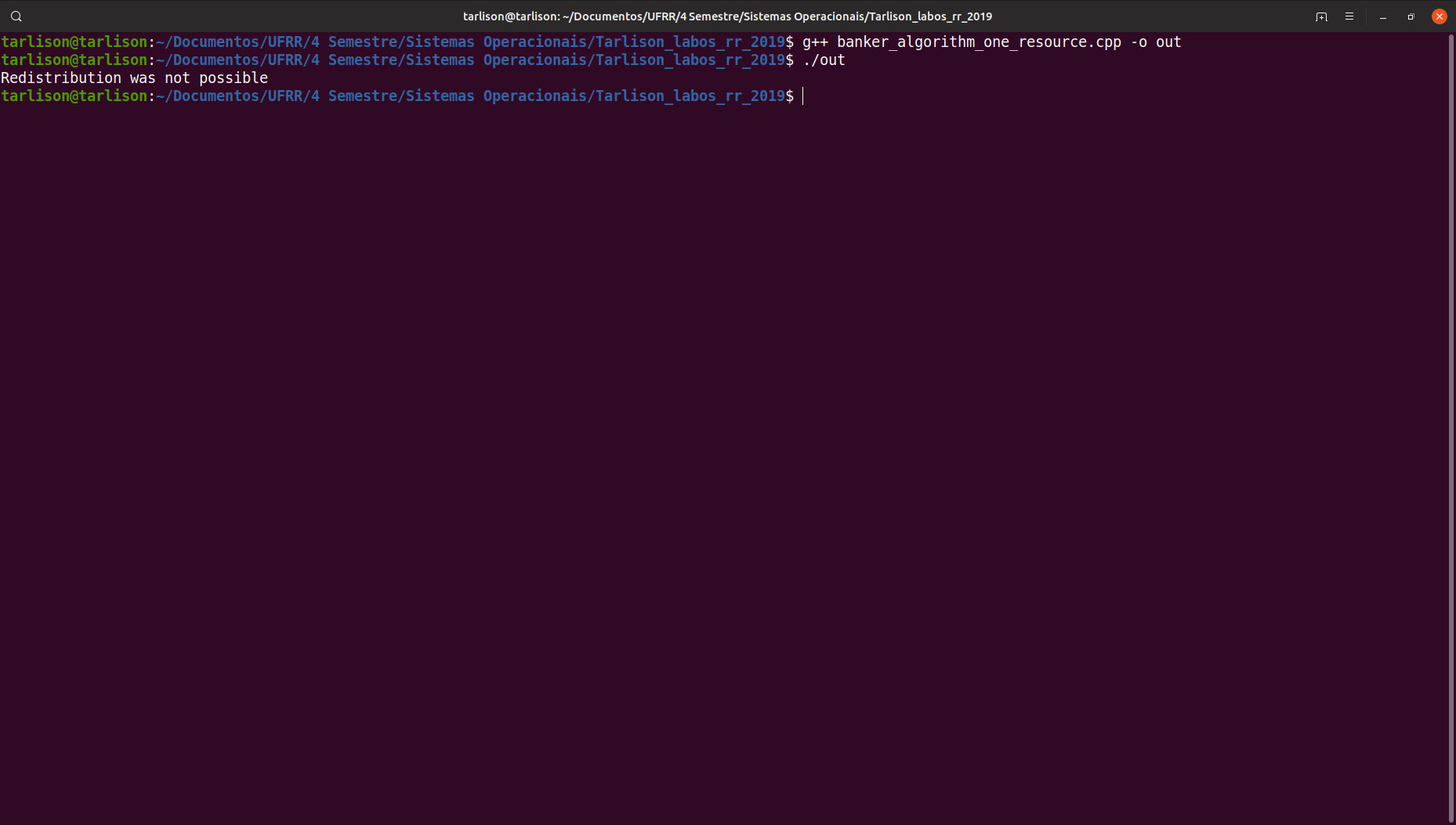
No algoritmo de apenas um recurso nós temos isso bem mais simplificado. Temos o vetor alloc que é um vetor unidimensional que armazena, respectivamente, os valores alocados por P0,P1,P2,P3, o vetor max que representa os valores máximos que cada processo aloca e a variável avail que possui a quantidade de recursos disponíveis. 

O algoritmo funciona da mesma forma que o de múltiplos recursos, mudando apenas a dimensão de algumas variáveis, mas a ideia permanece a mesma. Na sua execução essa será a saída:



Mudando a quantidade de recursos para 1 já não será mais possível distribuir de uma forma que todos os processos possam usar:

A saída é como o esperado, “Redistribution was not possible” pois não há recursos suficientes para isso.



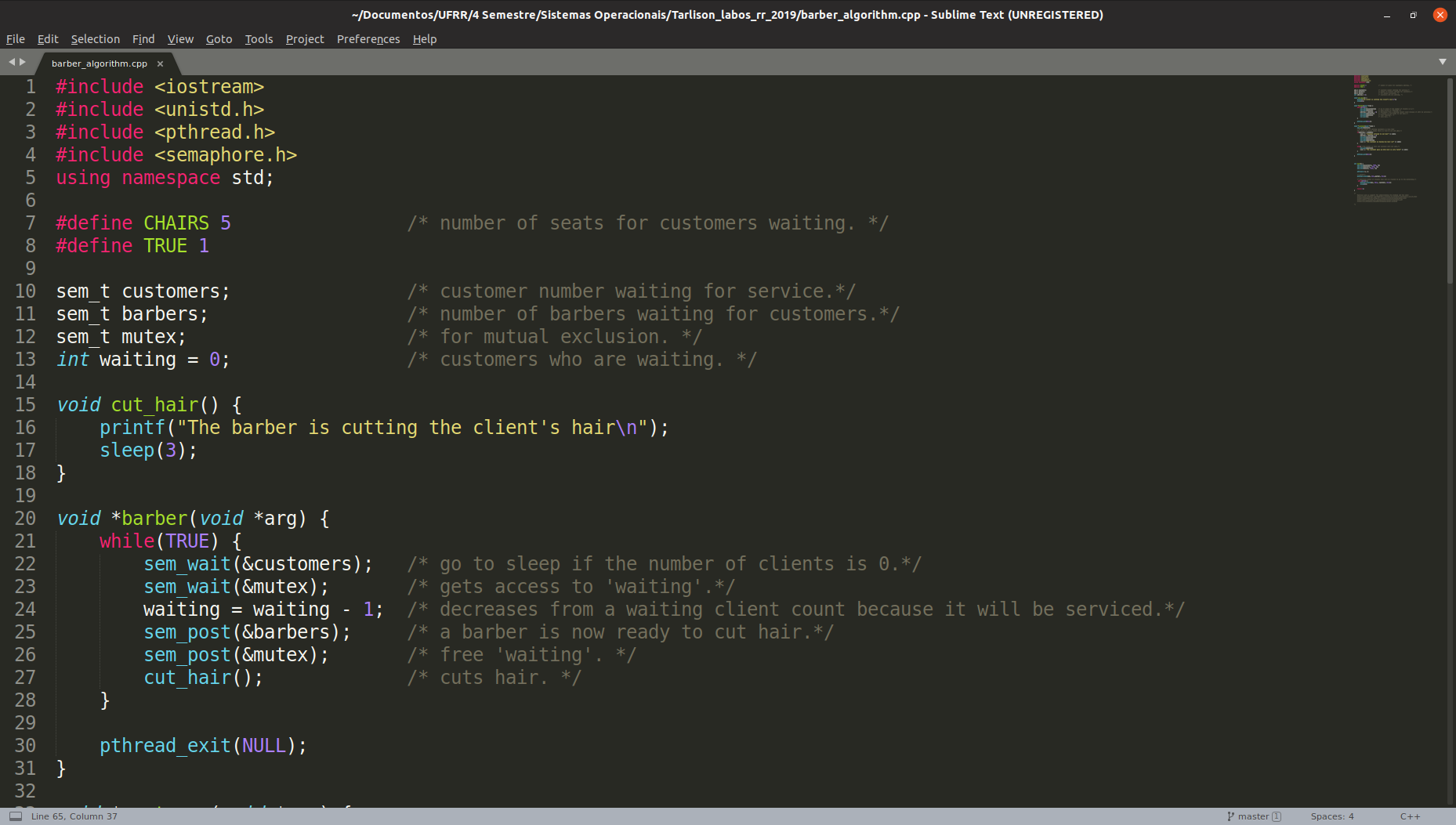
**[Questão 3]:**

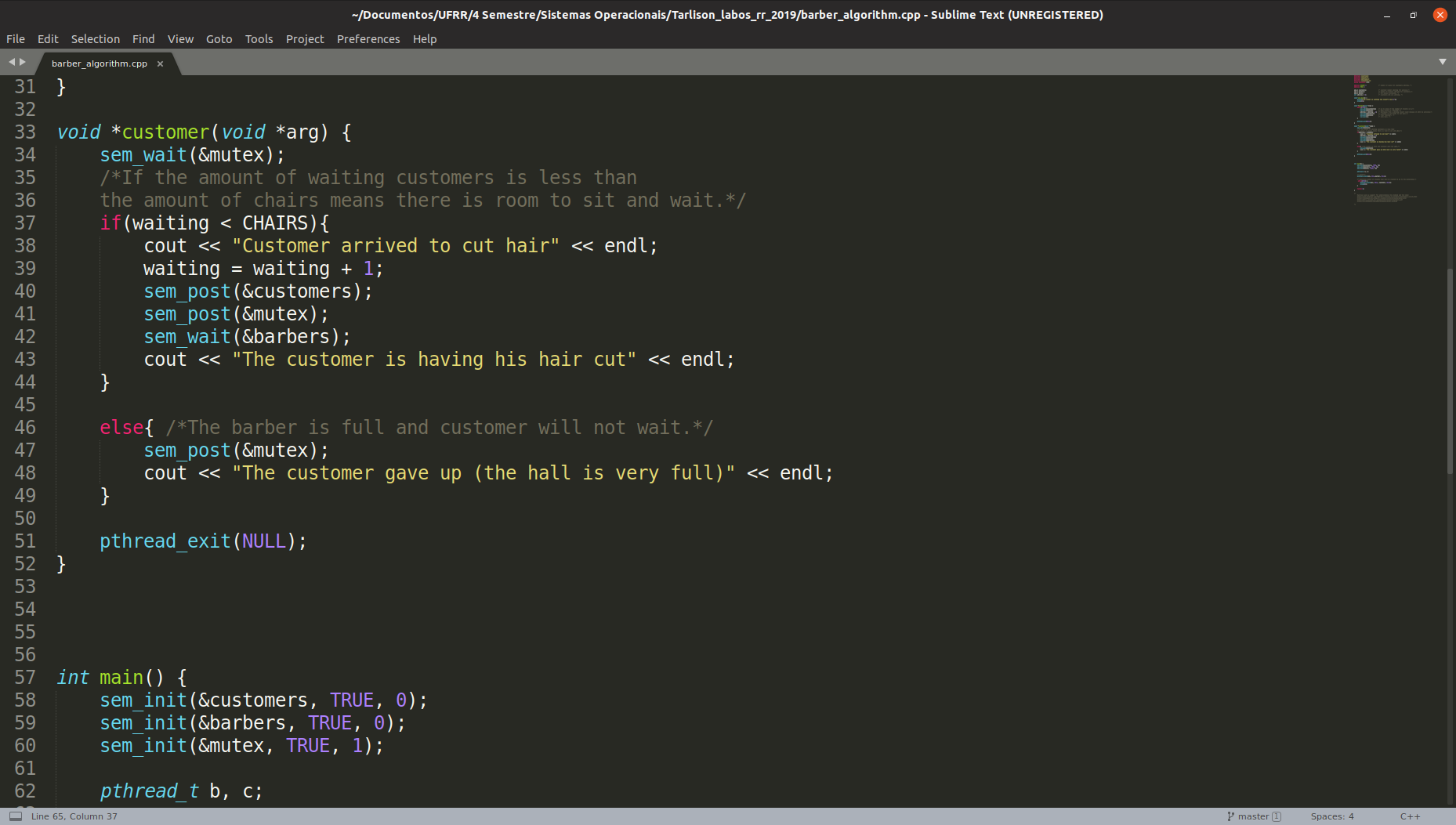
O algoritmo do Barbeiro dorminhoco funciona para resolver problemas de comunicações entre processos. O problema consiste em que em uma barbearia há um barbeiro, uma cadeira de barbeiro e N cadeiras para clientes poderem esperar a sua vez.

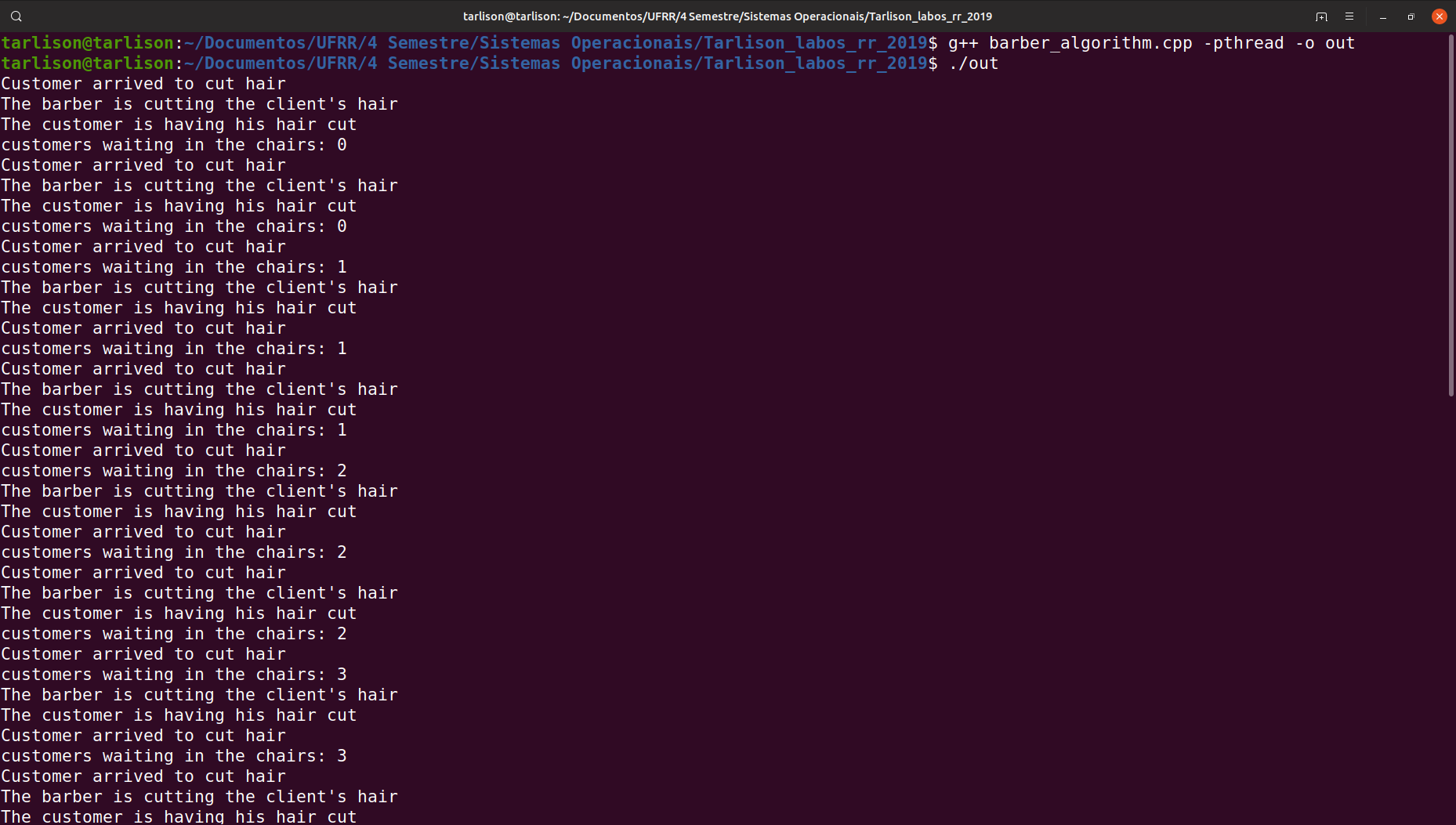
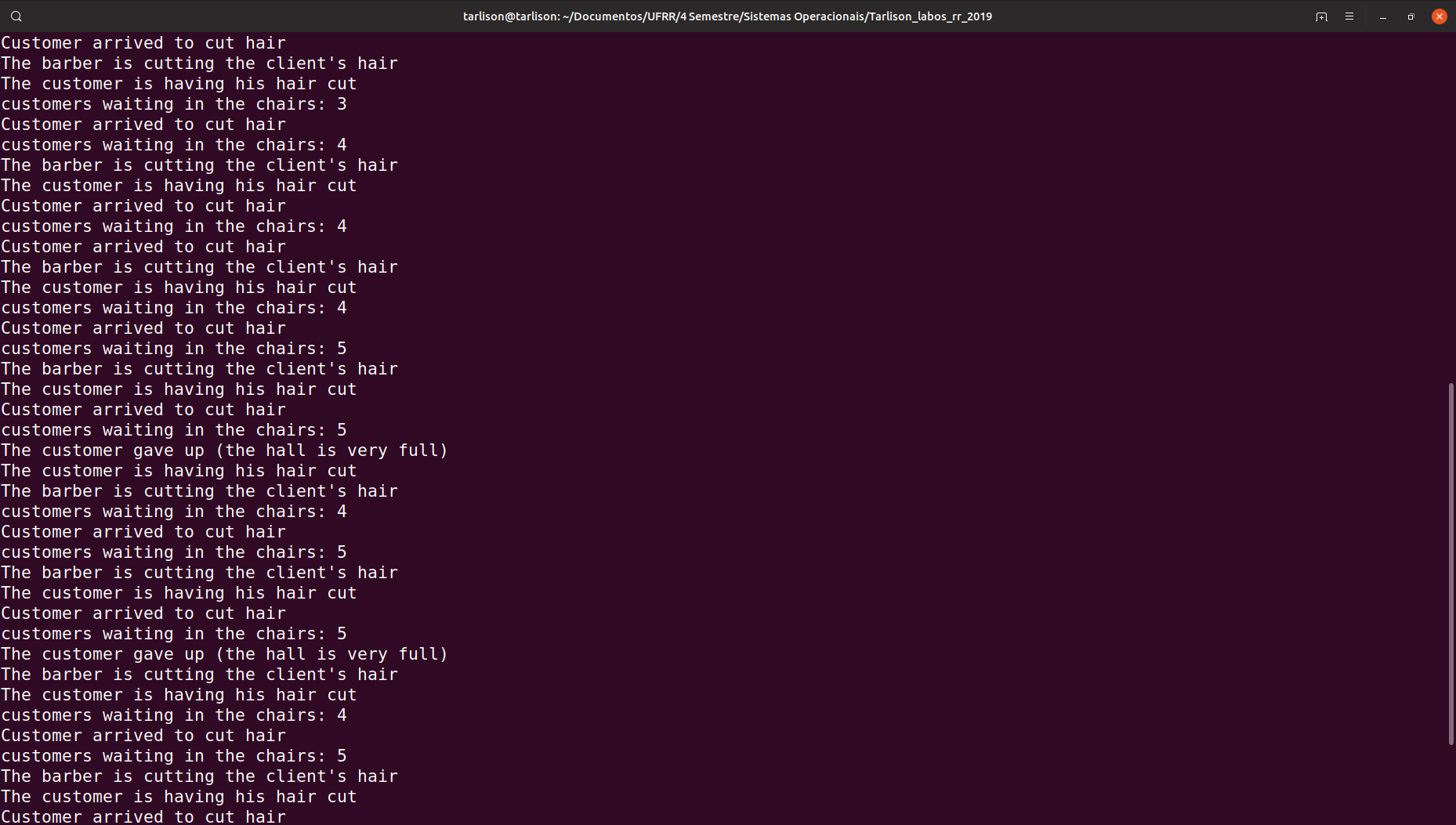
Quando não há clientes, o barbeiro senta-se na cadeira de barbeiro e cai no sono e assim que um cliente chega ele precisa acordar o barbeiro para atendê-lo. Caso outros clientes cheguem enquanto o barbeiro estiver atendendo outro cliente eles aguardam nas cadeiras caso haja cadeiras vazias, se não apenas vão embora. O problema é programar o barbeiro e os clientes para que não ocorra situações de disputa.

O código usa semáforos para controlar as regiões críticas. Quando começa o barbeiro chega para trabalhar e logo executa a função barber que irá fazer ele bloquear os semáforos customers e ele irá verificar que o número de clientes é 0 então irá dormir.

Quando o cliente chega ele inicia também o semáforo customers e o mutex, então acorda o barbeiro caso ele não esteja atendendo ninguém, caso ele esteja o cliente irá verificar se o número de clientes é menor do que o número de o número de cadeiras disponíveis.

Se for ele irá sentar e aguardar, acrescentando +1 ao valor da variável waiting (que é a quantidade de clientes esperando) e também dá um up no customers para acordar o barbeiro na sua vez . Se não ele sairá da barbearia e irá liberar o mutex. COmo podemos observar no código abaixo: 

Dependendo do número de cadeiras e do escalonamento das threads as cadeiras ficarão cheias mais rápido ou não. Essa será a saída considerando 5 cadeiras, se for alterado o código o mesmo acontecerá quando apenas mudará a quantidade de clientes que poderão ficar esperando:



Códigos e como compilar os mesmos estão disponíveis no github: <https://github.com/Tarlison/Tarlison_labos_rr_2019.git>

Este arquivo será disponibilizado em .odt, .docx e .pdf para evitar problemas de compatibilidade e/ou desformatação do mesmo.