UNIVERSIDADE FEDERAL DE SÃO CARLOS
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

Sistemas Operacionais (1001535)

Turmas A e B

Profa. Dra. Kelen Cristiane Teixeira Vivaldini

Trabalho 1:
Problema de sincronização entre tarefas

Eric Sales Vitor Junior. RA 758565. Engenharia de Computação.
Fernanda Malheiros Assi. RA 743534. Engenharia de Computação.
João Gabriel Viana Hirasawa. RA 759055. Engenharia de Computação.
Pabolo Vinícius da Rosa Pires. RA 760648. Ciência da Computação.
Víctor Cora Colombo. RA 727356. Engenharia de Computação.

25 de Outubro de 2020
São Carlos, SP

Sumário

1 Introdução

Para este trabalho, foram implementadas duas soluções na linguagem de programação *C* para o problema da *Hilzer's Barbershop* (Barbearia de Hilzer). O objetivo foi aplicar os conceitos de Sistemas Operacionais, como o uso de *threads*, mecanismos de sincronização e prevenção de impasse e inanição. Além disso, foi feita uma análise de desempenho entre as implementações dos problemas.

2 Descrição do problema

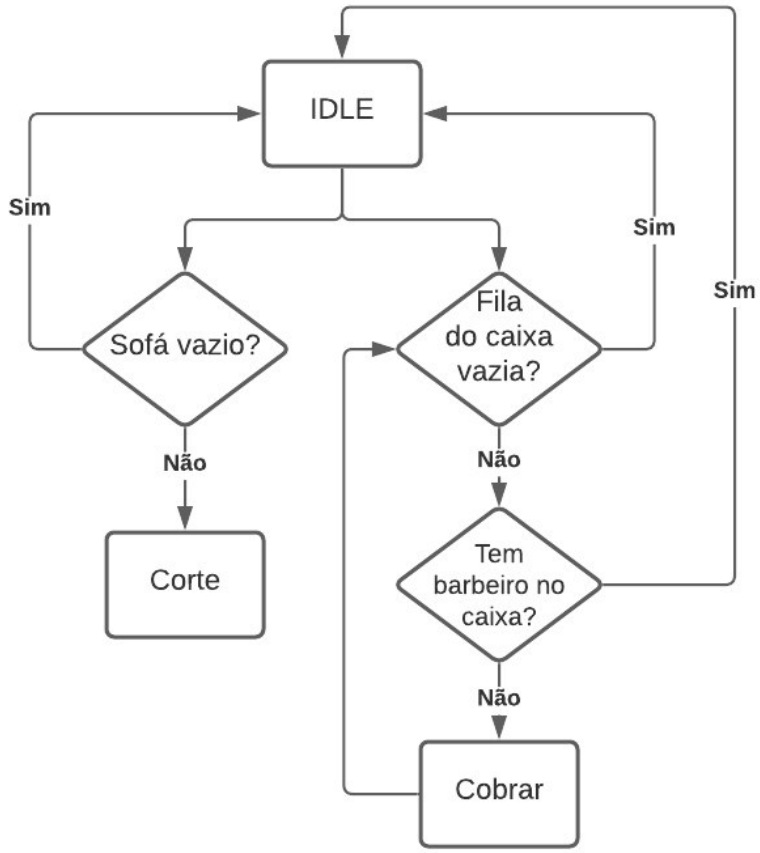
Este problema envolve o gerenciamento de uma barbearia que possui três barbeiros e três cadeiras para cortar o cabelo dos clientes. Nessa barbearia, há uma área de espera com um sofá de quatro lugares e uma parte para esperar de pé. A lotação é de 20 clientes, e os clientes adicionais são barrados na entrada.

Quando um cliente entra na loja, ele pode sentar no sofá, caso haja vaga, ou esperar de pé, mas a ordem de fila sempre é respeitada. Se um barbeiro está livre, é atendido o cliente que está no sofá há mais tempo, e a pessoa de pé há mais tempo, caso haja, pode sentar no sofá.

Ao finalizar um corte, um dos barbeiros deve cobrar o cliente, mas há apenas um caixa, então apenas um cliente pode pagar por vez. Caso não haja mais clientes para um barbeiro atender, ele pode dormir em sua cadeira, esperando por novos clientes.

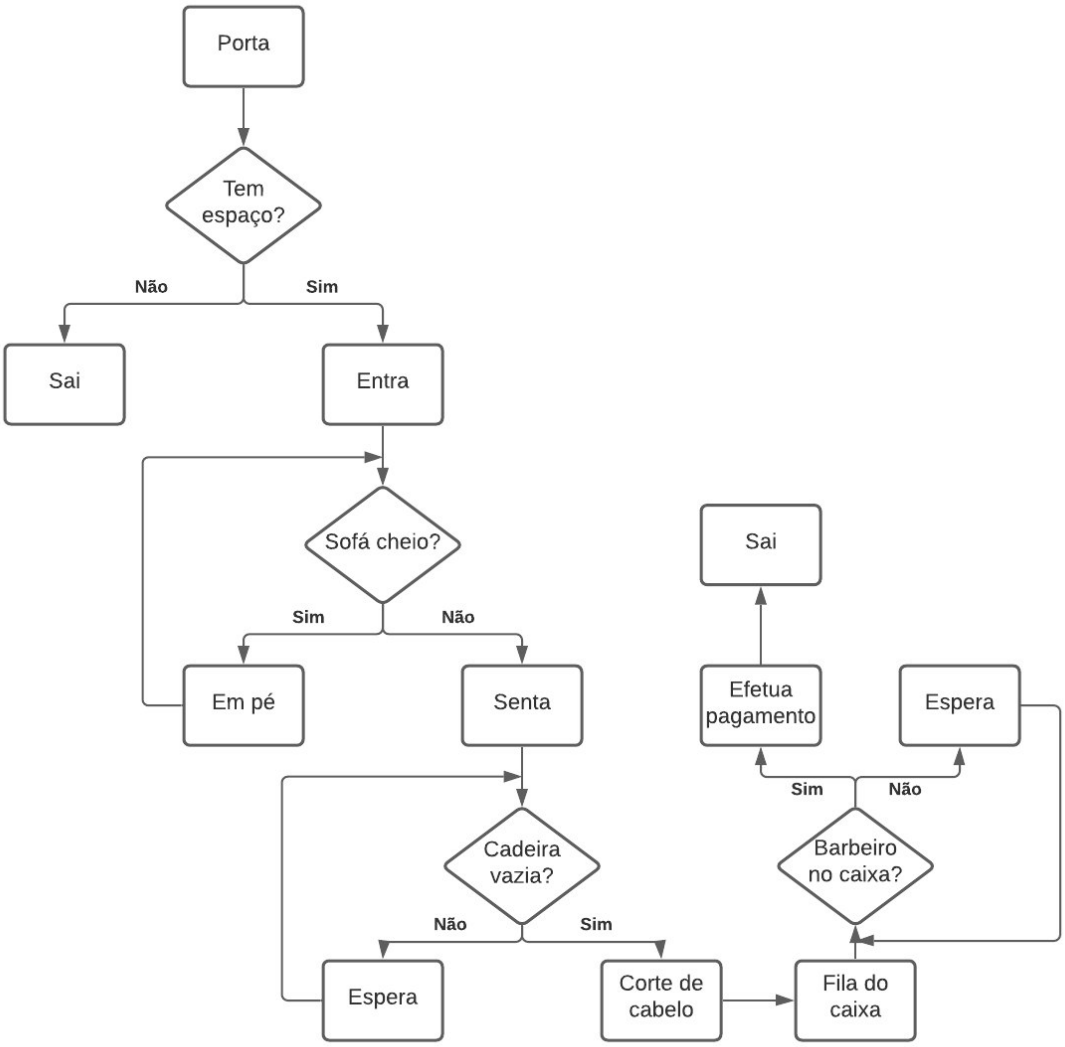
Foram feitos esquemas para detalhar as funções de barbeiro e cliente, nas Figuras 1 e 2, respectivamente.

Figura 1 – Esquemático das funções e estados do barbeiro



Fonte: Organizado pelos autores.

Figura 2 – Esquemático das funções e estados do cliente



Fonte: Organizado pelos autores.

3 Primeira implementação

3.1 Criação e inicialização

Primeiramente, o programa pode receber parâmetros que definem os valores da situação-problema. Por meio de argumentos na execução no terminal, é definido o número de clientes, número de barbeiros, número de lugares no sofá e o número máximo de pessoas na loja, respectivamente. Esses argumentos são guardados em uma *struct* de parâmetros, que é criada em `setupParameters` e utilizada ao longo do programa.

A inicialização de semáforos e *mutexes* está encapsulada em um procedimento `initSemaphoresAndMutexes` e a inicialização das filas a serem utilizadas em `initQueues`.

A criação de todas as *threads*, por sua vez, está encapsulada no procedimento `doThreadsCreationAndJoins`.

3.2 Semáforos

Foi implementada uma *struct* com dois semáforos que serão utilizados no controle dos clientes. O semáforo `shopToClient_sem` da *struct* serve para controle da Secretaria e do Barbeiro para permitir que o cliente sente no sofá ou na cadeira, sinalizar o corte do cabelo e receber o pagamento. `clientToShop_sem`, também da *struct*, para sinalizar que o cliente está pronto para cortar realização do pagamento.

Além destes dois semáforos, o cliente utiliza outros 4: `standup_sem`, que limita o número máximo de clientes na loja; `sofa_sem`, que indica os lugares no sofá; `clientWaitingForSofa_sem`, que indica a espera do cliente por um lugar no sofá; e `clientWaitingForBarberChair_sem`, que indica a espera do cliente por uma cadeira para cortar.

3.3 *Mutexes*

São utilizados três *mutexes* para garantir a exclusão mútua das filas e do caixa: `standupToSofa_mutex`, para a fila dos clientes esperando uma vaga no sofá; `sofaToBarber_mutex`, para a fila dos clientes esperando uma cadeira para cortar; e `payment_mutex`, para realizar os pagamentos.

3.4 Filas

As filas utilizadas são adaptações do site *GeeksforGeeks* [1]. O conteúdo é a *struct* `Client` e essas filas, então, representam as filas para sentar no sofá e para cortar o cabelo.

[1] Disponível em: `https://www.geeksforgeeks.org/queue-linked-list-implementation/`. Acesso em: 22 out 2020.

3.5 Cliente

3.5.1 *Thread* principal

O cliente, inicialmente, verifica a lotação da loja e, caso não haja espaço, ele apenas vai embora (`leaveShop`). Caso contrário, começa-se a rotina padrão: ele entra na loja (`enterShop`), espera para poder sentar no sofá (`waitAndSeatOnSofa`), espera para sentar na cadeira do barbeiro (`seatOnBarberChair`), tem o cabelo cortado (`getHaircut`), realiza o pagamento (`pay`) e sai da loja (`leaveShop`).

Após finalizada uma dessas duas rotinas possíveis, o cliente deixa de existir.

3.5.2 `enterShop`

Com esse procedimento, o cliente entra na loja e vai para a área de espera. O *mutex* da fila do sofá é trancado e o cliente é adicionado à fila para sentar no sofá. Após isso, o *mutex* é liberado e o cliente estará esperando para sentar.

3.5.3 `waitAndSeatOnSofa`

Aqui, o cliente avisa que está aguardando na fila para o sofá e espera até a secretaria informar que pode sentar. O *mutex* da fila para cortar o cabelo é trancado e o cliente entra nessa fila. Após isso, o *mutex* é liberado.

3.5.4 `seatOnBarberChair`

O cliente notifica o barbeiro de que está esperando para cortar o cabelo e aguarda que seja sinalizado para sentar na cadeira.

3.5.5 `getHaircut`

O cliente, já sentado na cadeira, informa o barbeiro de que está pronto para cortar e aguarda que o barbeiro corte seu cabelo.

3.5.6 `pay`

O cliente informa o barbeiro que está pronto para realizar o pagamento e aguarda que o barbeiro o atenda no caixa.

3.5.7 `leaveShop`

O cliente sai da loja. Caso não tenha conseguido entrar na loja, nada é feito. Caso contrário, ele sai liberando uma vaga para clientes na lotação da loja.

3.6 Barbeiro

3.6.1 *Thread* principal

Quando o barbeiro é inicializado, ele entra em um *loop* infinito. Primeiro, aguarda que algum cliente esteja esperando para cortar o cabelo. Então, remove esse cliente da fila (`dequeueClientFromSofa`), o coloca na cadeira (`callClientToChair`), corta o cabelo do cliente (`cutHair`) e aceita o pagamento (`acceptPayment`). Após isso, descansa por uns instantes e repete esse procedimento.

3.6.2 `dequeueClientFromSofa`

O *mutex* da fila para cortar o cabelo é trancado, e então o cliente é retirado da fila e o espaço no sofá é liberado. O cliente é retornado nessa função.

3.6.3 `callClientToChair`

É enviado ao cliente o sinal de que há uma cadeira livre para cortar o cabelo, complementando `seatOnBarberChair`.

3.6.4 `cutHair`

Aguarda-se que o cliente se sente na cadeira e, após um tempo, é enviado ao cliente de que o cabelo foi cortado. Complementa `getHaircut`.

3.6.5 `acceptPayment`

O barbeiro aguarda que o cliente queira pagar, e então bloqueia o *mutex* de pagamento por determinado tempo e aceita o pagamento.

3.7 Secretaria

3.7.1 *Thread* principal

A secretaria também é colocada em um loop infinito, e sua única função é aguardar que um cliente esteja esperando para sentar no sofá, aguardar que haja lugar livre no sofá e mandar o cliente para o sofá com `sendClientToSofa`.

3.7.2 `sendClientToSofa`

A secretaria tranca o *mutex* da fila dos clientes esperando para sentar no sofá e então retira um cliente dessa fila, desbloqueia o *mutex* e avisa o cliente de que pode sentar no sofá.

4 Segunda implementação

4.1 Criação e inicialização

Nesse caso, os parâmetros do número de clientes, número de barbeiros, número de lugares no sofá e número máximo de pessoas na loja são definidos por meio de *macros*.

A inicialização dos *mutexes* está em um procedimento `initialize_mutexes` e a inicialização dos semáforos no procedimento `initialize_semaphores`. As filas são inicializadas com `createQueue`.

4.2 Semáforos

Foram utilizados dois semáforos gerais, e cada cliente possui um semáforo próprio.

O semáforo `sofa` faz controle dos lugares disponíveis do sofá e `barber` indica que há barbeiro disponível para cortar.

O semáforo do cliente serve para toda a comunicação do barbeiro com o cliente: chamar para cortar o cabelo, sinalizar que o corte finalizou e autorizar o pagamento no caixa.

4.3 *Mutexes*

São utilizados dois *mutexes*, o `door` controla a entrada e saída de clientes na barbearia, impedindo que entre mais que a capacidade do estabelecimento. Já o `cashRegister` controla a caixa registradora, garantindo que haja barbeiro para aceitar o pagamento e que o barbeiro não fique ocioso quando não há cliente esperando para pagar.

4.4 Filas

As filas utilizadas são adaptações do site *GeeksforGeeks* [2]. É usada uma fila, a `sofa_queue` que é a fila para os clientes sentarem no sofá.

4.5 Cliente

4.5.1 *Thread* principal

Inicialmente, o cliente verifica a capacidade da loja, e entra caso haja espaço. Caso contrário, a *thread* apenas finaliza. Se o cliente conseguir entrar, o contador de clientes (protegido pelo *mutex* `door`) é incrementado.

Então, o cliente fica em pé aguardando espaço no sofá e, quando consegue, se coloca na fila para cortar o cabelo (`sofa_queue`). Após isso, o cliente sinaliza que está

[2] `https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/`. Acesso em: 22 out 2020.

aguardando pela disponibilidade de um barbeiro, seguido de sinalizar que está esperando ser chamado. Quando é chamado, libera o espaço no sofá e sinaliza que está aguardando o corte terminar. Então, aguarda que possa pagar. Quando consegue pagar, decrementa o contador, ainda protegido, e a *thread* retorna.

4.5.2 `make_customer`

Essa função cria e retorna um cliente novo. São inicializados o id e o semáforo associado ao cliente.

4.5.3 `destroy_customer`

Esse procedimento apenas destrói e libera os recursos utilizados pelo cliente.

4.5.4 `get_id`

Essa função retorna o id do cliente.

4.6 Barbeiro

4.6.1 *Thread* principal

Nessa implementação, cada barbeiro recebe uma fila própria para os pagamentos. Ou seja, os clientes que terminarem de cortar o cabelo com determinado barbeiro irão direto para a fila de caixa desse barbeiro.

Caso haja algum cliente na fila do caixa desse barbeiro esperando para pagar E o caixa estiver livre, o barbeiro ocupa esse caixa e recebe os pagamentos dos clientes nessa fila até que não haja mais clientes para pagar naquele momento.

Caso não haja clientes na fila de pagamento do barbeiro ou o caixa não esteja livre, o barbeiro verifica se há algum cliente no sofá esperando para cortar o cabelo. Se isso acontecer, o barbeiro sinaliza que está livre, retira o cliente da fila e realiza o corte. Após isso, coloca o cliente em sua fila de pagamento, que pode estar vazia ou não nesse momento.

Por fim, se nenhuma dessas condições for verdadeira, o barbeiro apenas aguarda e verifica tudo novamente após um tempo.

4.6.2 `make_barber`

Essa função cria e retorna um barbeiro novo. São inicializados o id e a fila de caixa do barbeiro.

4.6.3 `destroy_barber`

Esse procedimento apenas destrói e libera os recursos utilizados pelo barbeiro.

5 Análise dos resultados

Para a análise dos resultados, foi utilizado o comando `time` do Linux. Com ele, podemos ter o tempo de execução do programa dividido em três partes. São elas:

- **Real**: o tempo decorrido entre a execução e a conclusão do comando;
- **User**: o tempo de usuário;
- **Sys**: o tempo de kernel.

Ambas as soluções foram testadas com a barbearia contendo 3 barbeiros e 3 cadeiras, 4 espaços no sofá, lotação para 20 pessoas e 25 clientes querendo entrar na loja. Cada teste foi rodado três vezes e a média dos *outputs* foram colocadas em tabelas para melhor visualização.

5.1 Primeira implementação

Para esta implementação, foram feitos três testes.

5.1.1 Primeiro teste

No primeiro teste, é feita a execução do código com alguns `sleep` arbitrários que foram implementados anteriormente.

Tabela 1 – Teste com o código original

Modo	Tempo
real	0m58.370s
user	0m0.000s
sys	0m0.005s

Fonte: Elaborada a partir dos dados no terminal do Linux.

Como pode ser observado, o tempo total de execução foi de cerca de 58.370 segundos, o tempo de usuário foi de 0.000 segundos e as chamadas de sistemas duraram cerca de 0.005 segundos. O `sleep` torna a execução do programa mais lenta, mas seu uso tem exatamente essa intenção.

5.1.2 Segundo teste

Para o segundo teste, os `sleep` foram retirados. Com isso, esperamos que o tempo total de execução reduza drasticamente.

Tabela 2 – Teste sem `sleep`

Modo	Tempo
real	0m0.018s
user	0m0.000s
sys	0m0.016s

Fonte: Elaborada a partir dos dados no terminal do Linux.

O tempo total de execução foi de cerca de 0.018 segundos, o tempo de usuário foi de 0.000 segundos e as chamadas de sistemas duraram cerca de 0.016 segundos.

5.1.3 Terceiro teste

Para o terceiro teste, foi retirado do código, além dos `sleep`, todos os *outputs* das funções `printf`.

Tabela 3 – Teste sem `sleep` e sem `printf`

Modo	Tempo
real	0m0.015s
user	0m0.000s
sys	0m0.016s

Fonte: Elaborada a partir dos dados no terminal do Linux.

Nesse caso, o tempo real de execução foi de 0.015 segundos, o tempo de usuário foi de 0.000 segundos e as chamadas de sistemas duraram 0.016 segundos.

5.2 Segunda implementação

Para esta implementação, também foram feitos três testes.

5.2.1 Primeiro teste

No primeiro teste, é feita a execução do código da mesma maneira, mantendo os `sleep` arbitrários que foram implementados na solução.

Tabela 4 – Teste com o código original

Modo	Tempo
real	0m10.021s
user	0m0.000s
sys	0m0.011s

Fonte: Elaborada a partir dos dados no terminal do Linux.

Como pode ser observado, o tempo total de execução foi de cerca de 10.021 segundos, o tempo de usuário foi de 0.000 segundos e as chamadas de sistemas duraram cerca de

0.011 segundos. É claro que não há comparação cabível com o primeiro teste da outra implementação ou qualquer outro teste, já que os `sleep` são completamente arbitrários em ambas as soluções.

5.2.2 Segundo teste

Para o segundo teste, os `sleep` foram retirados.

Tabela 5 – Teste sem `sleep`

Modo	Tempo
real	0m0.020s
user	0m0.011s
sys	0m0.016s

Fonte: Elaborada a partir dos dados no terminal do Linux.

O tempo real de execução foi de cerca de 0.020 segundos, o tempo de usuário foi de 0.011 segundos e as chamadas de sistemas duraram cerca de 0.016 segundos.

5.2.3 Terceiro teste

Para o terceiro teste, foi novamente retirado todos os `sleep` e `printf`.

Tabela 6 – Teste sem `sleep` e sem `printf`

Modo	Tempo
real	0m0.018s
user	0m0.000s
sys	0m0.016s

Fonte: Elaborada a partir dos dados no terminal do Linux.

Nesse caso, o tempo real de execução foi de 0.018 segundos, o tempo de usuário foi de 0.000 segundos e as chamadas de sistemas duraram 0.016 segundos. Nota-se a redução de cerca de 0.545 segundos do tempo total de execução entre o teste anterior e este.

5.3 Comparação

No segundo teste, a primeira implementação possui um tempo real médio de 0.018 segundos e a segunda implementação de 0.020 segundos. O tempo de usuário médio foi de 0 segundos e 0.011 segundos, respectivamente. A média do tempo de sistema foi igual para ambas implementações, a 0.016 segundos. Para o tempo real, houve uma diferença de 0.002 segundos, o que representa uma melhora de 10% de velocidade para a primeira implementação.

No segundo teste, a primeira implementação possui um tempo real médio de 0.015 segundos e a segunda implementação de 0.018 segundos. O tempo de usuário médio foi

igual para ambas, a 0 segundos. A média do tempo de sistema também foi igual, a 0.016 segundos. Houve uma diferença de 0.003 para o tempo real, dessa vez representando uma melhora de cerca de 16,67% para a primeira implementação.

6 Conclusão

O grupo concluiu que problemas de comunicação de processos podem ser abordados de inúmeras maneiras, mas que muitas precauções devem ser tomadas a fim de evitar impasses e inanição dos processos. O problema da Barbearia de Hilzer traz desafios como o gerenciamento de filas e o afunilamento para o único caixa disponível, no final da rotina dos clientes. Mesmo assim, esse problema pode ser atacado com o uso, no conceito base, apenas de semáforos, *mutexes* e uma estrutura de fila.

Por fim, o grupo também concluiu que as decisões tomadas nas implementações têm um impacto no tempo de execução final e, portanto, o projeto deve ser bem pensado, o que felizmente foi a preocupação principal desde o começo.

7 Referências Bibliográficas

DOWNEY, Allen B. **The Little Book of Semaphores**. 2. ed. [S. L.]: Green Tea Press, 2016. 279 p. Disponível em: http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf. Acesso em: 22 out. 2020.

TANENBAUM, Andrew s; BOS, Herbert. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Pearson Education do Brasil, 2016. 758 p. ISBN 978-85-4301-818-8.