

Pontifícia Universidade Católica de Campinas Faculdade de Engenharia de Computação - FECOMP

Sistemas Operacionais A – Relatório Experimento 5

Beatriz Morelatto Lorente RA: 18071597

Cesar Marrote Manzano RA: 18051755

Fabricio Silva Cardoso RA: 18023481

Pedro Ignácio Trevisan RA: 18016568

**Sumário**

**1.Introdução.......................................................................................................3**

**2.Respostas das perguntas..............................................................................4**

**3.Resultados da execução do primeiro programa..........................................5**

**4.Resultados da execução do segundo programa.........................................9**

**5.Análise dos Resultados..................................................................................0**

**6.Conclusão........................................................................................................0**

**Introdução**

O experimento realizado permitiu o melhor entendimento de todos os conceitos adquiridos ao longo do semestre, como: criação de processos filhos, fila de mensagens, memória compartilhada, semáforos e threads. Foram feitos dois programas diferentes, visando a resolução do problema clássico do barbeiro dorminhoco.

No primeiro programa, era necessário o uso de processos filhos, fila de mensagens e memória compartilhada. O número de cadeiras disponíveis era igual a sete, e era necessário inicializar dois processos barbeiros e vinte processos clientes. Um processo cliente, envia uma mensagem ao barbeiro, com uma string de tamanho e números variáveis (entre 2 e 1023) e o barbeiro devolvia a mensagem com o resultado do serviço prestado. Caso o cliente não fosse atendido, era necessário mostrar o seu número. Caso contrário, era necessário imprimir o resultado da string que seria, de sua respectiva ordenação e o tempo de demora para ser atendido, ou seja, para iniciar o corte de cabelo.

No segundo programa, era necessário trocar os processos filhos por threads, mutex para exclusão mútua e semáforos ao invés de fila de mensagens. O número de cadeiras mantém em sete, o de barbeiros vai para três e o de clientes para vinte e sete. A lógica e as impressões necessárias, funcionam de maneira similar ao primeiro programa.

**Respostas das perguntas**

***Pergunta 1: Qual é o recurso comum que necessita de exclusão mútua ?***

**Resposta:** O número de cadeiras disponíveis será um recurso comum do barbeiro e do cliente, portanto será necessário a exclusão mútua dessa variável.

***Pergunta 2: De que maneira (leitura, escrita, ambos) barbeiros e clientes vão acessar o recurso comum ?***

**Resposta:** No primeiro programa é necessário implementar um semáforo que trava o acesso ao recurso comum, que trava tanto o barbeiro quanto o cliente. No segundo programa isso é feito com o uso de mutex.

***Pergunta 3: Como os números foram colocados na String ?***

**Resposta:** Os números foram convertidos para string com o auxílio da função ‘sprintf()’. A cada número convertido, era adicionado um espaço em branco, para facilitar a conversão de string para número novamente.

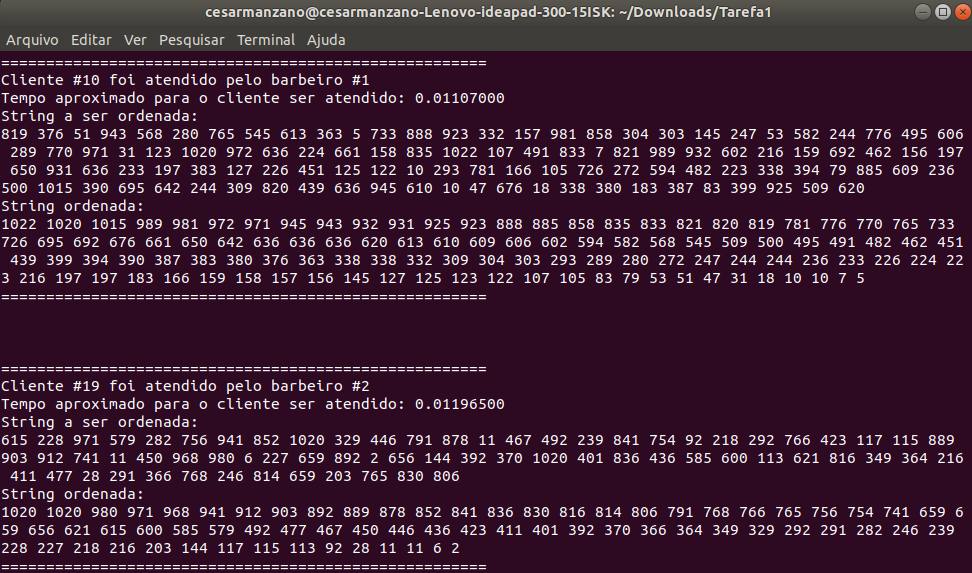
***Pergunta 4: Como o barbeiro vai ter acesso aos valores a serem ordenados ?***

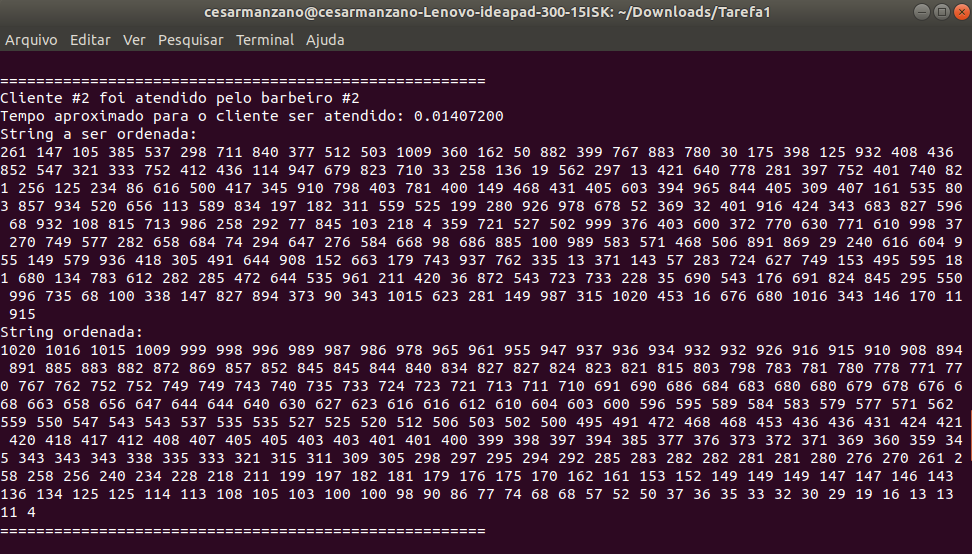
**Resposta:** No primeiro programa, o cliente passa uma mensagem com a string para ser ordenada, é necessário converter em um vetor de inteiros e então ordenar em forma decrescente. No segundo programa, é implementado um vetor de struct, no qual o barbeiro acessa e pega a string do cliente.

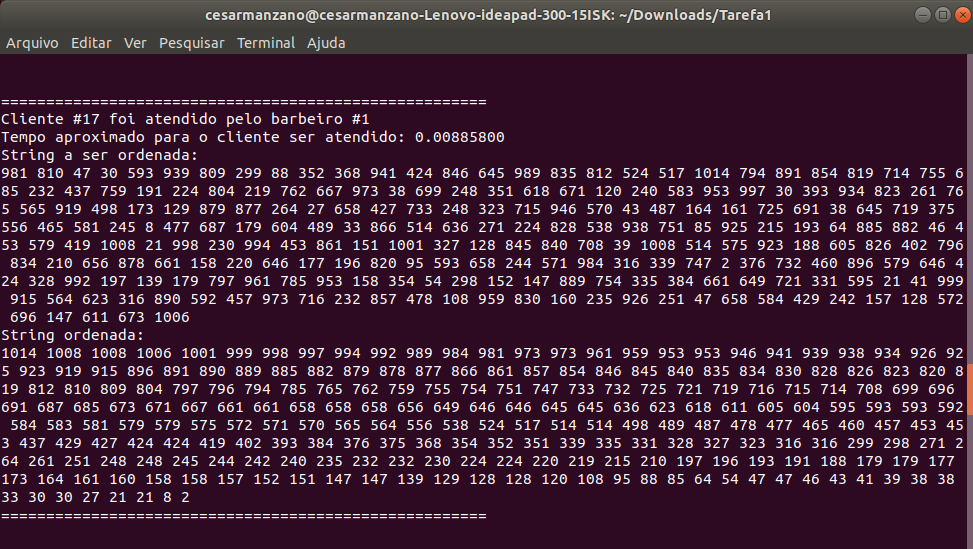
***Pergunta 5: Como o cliente vai ter acesso aos resultados?***

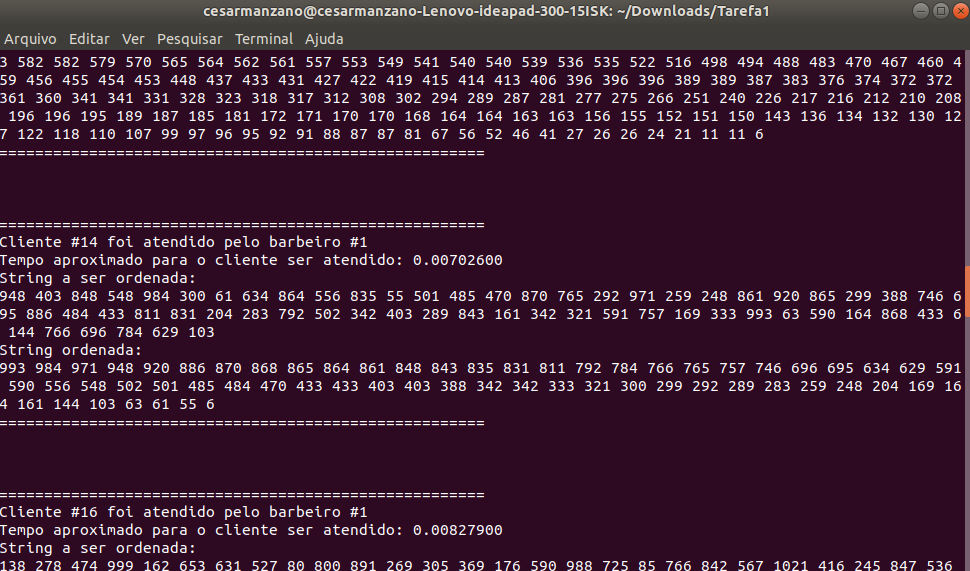
**Resposta:** No primeiro programa, por meio de uma mensagem enviada pelo barbeiro. No segundo programa, o cliente acessa a struct após o barbeiro inserir as informações necessárias, para assim imprimir de maneira correta.

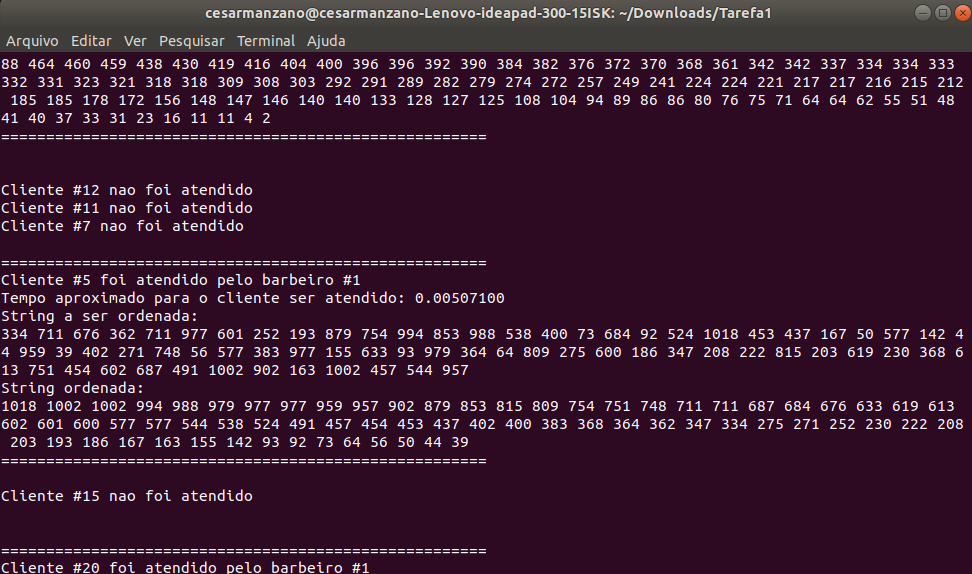
**Resultados da execução do primeiro programa**

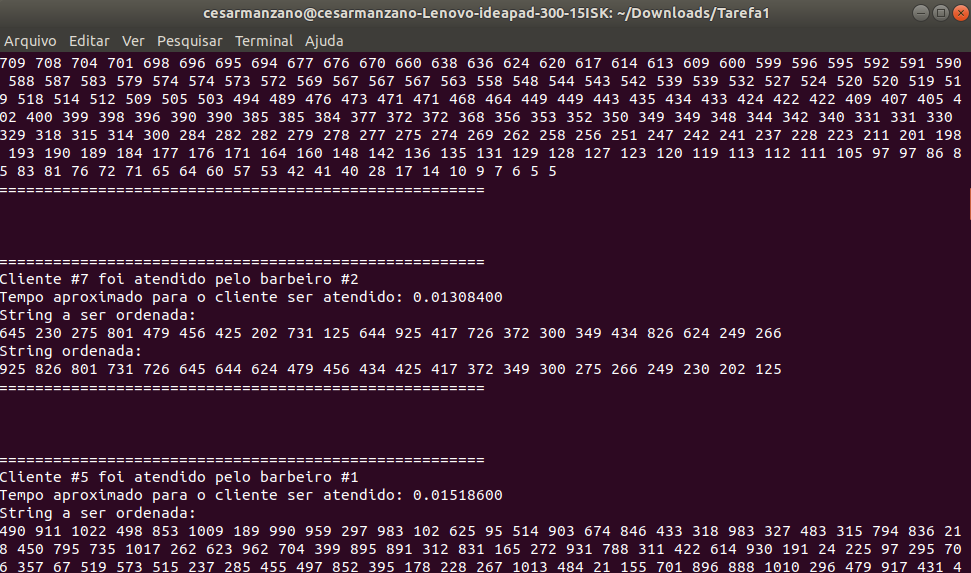


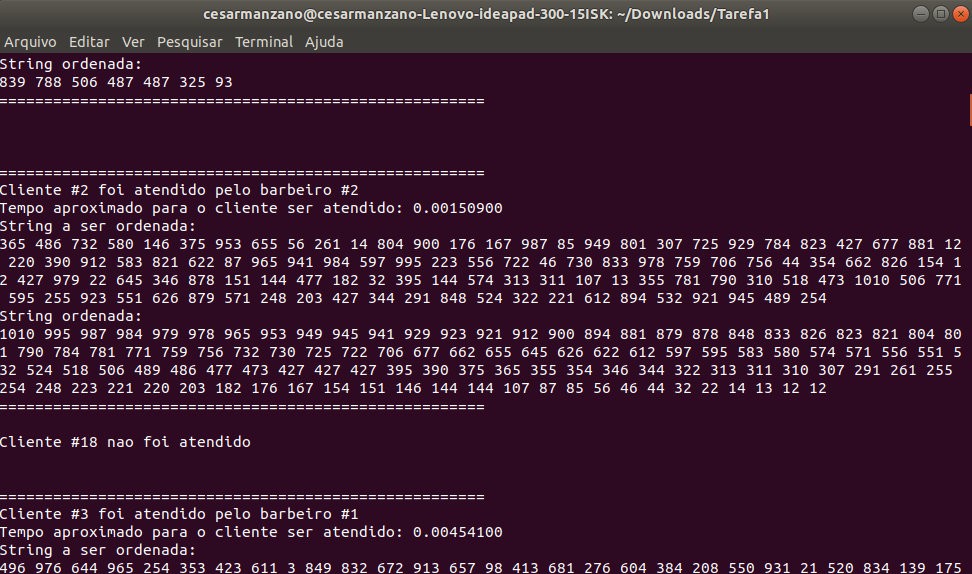




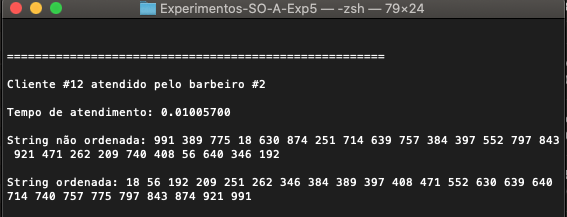


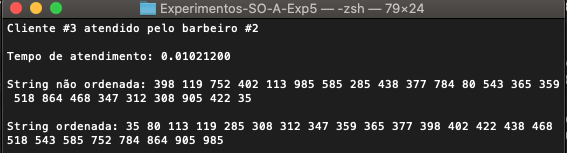


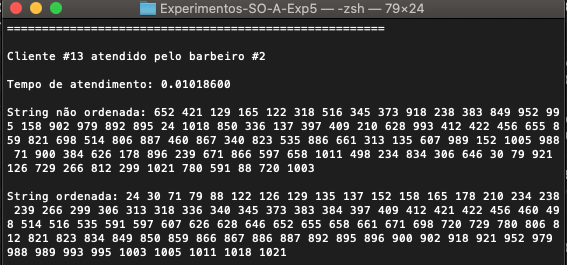


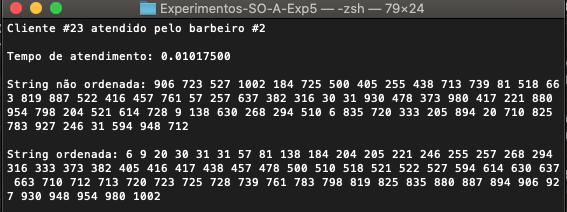


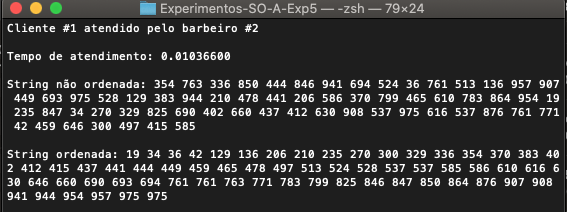
**Resultados da execução do segundo programa**

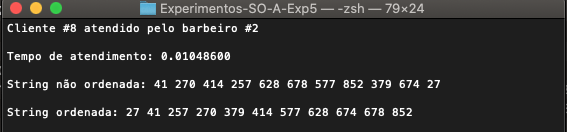


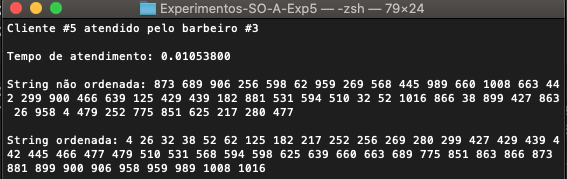


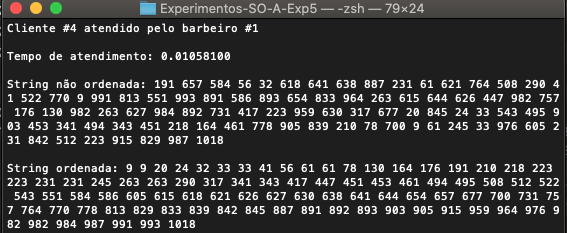


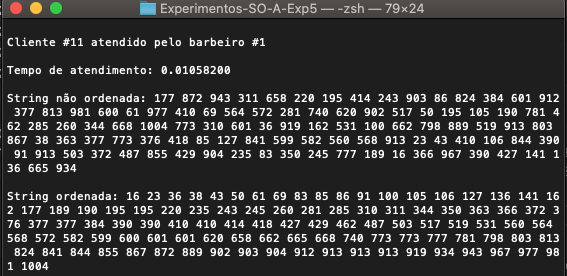


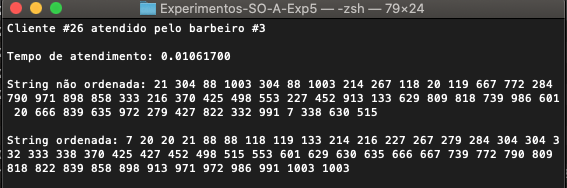


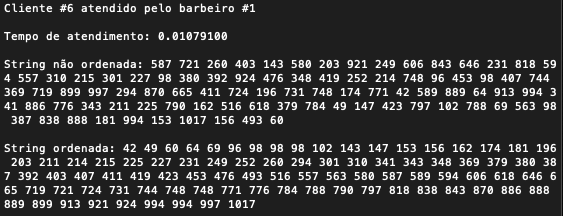


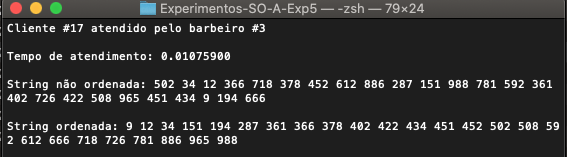


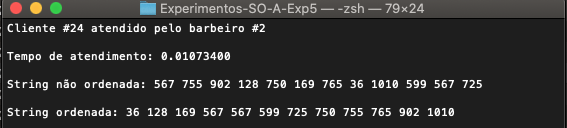


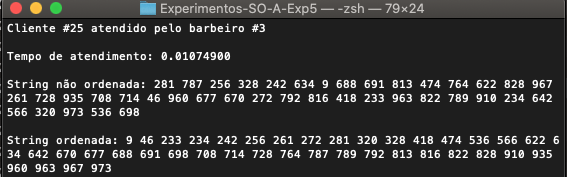


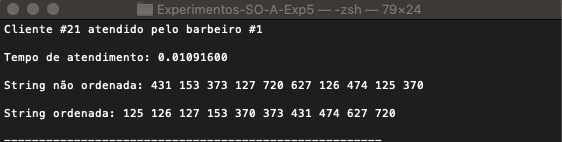


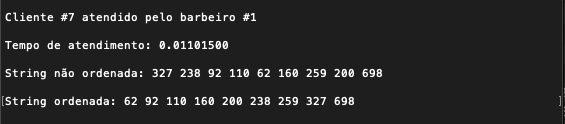


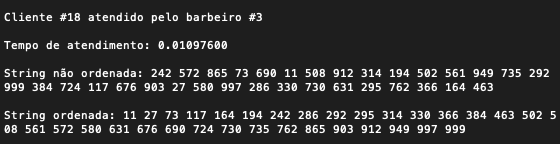


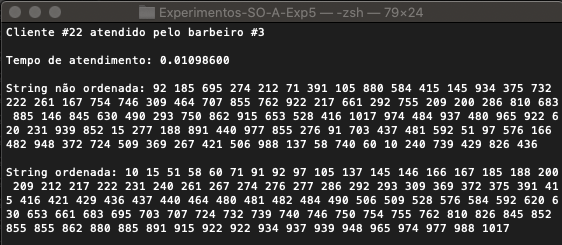


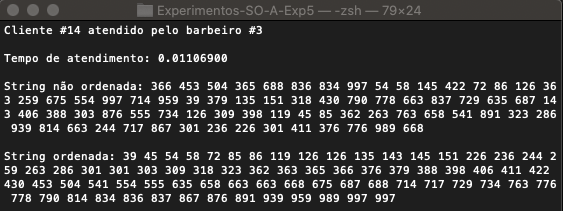


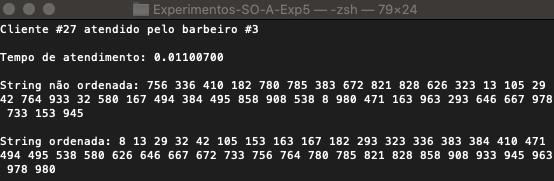


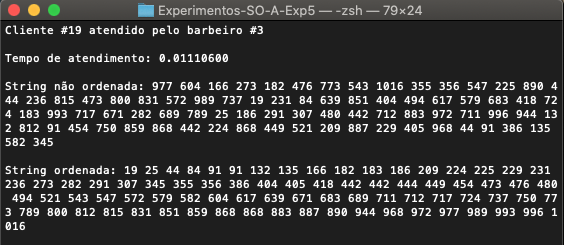


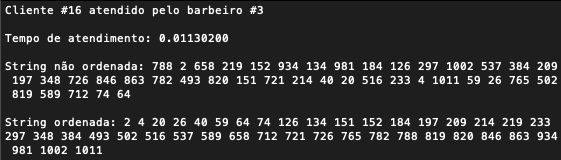


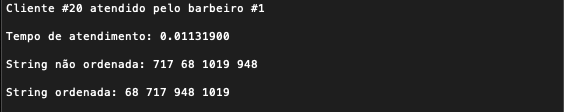


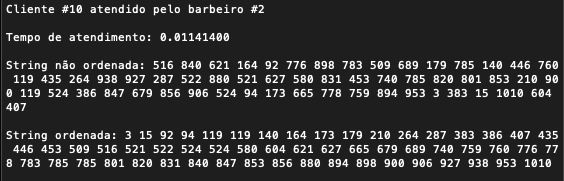


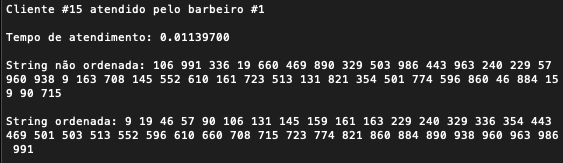


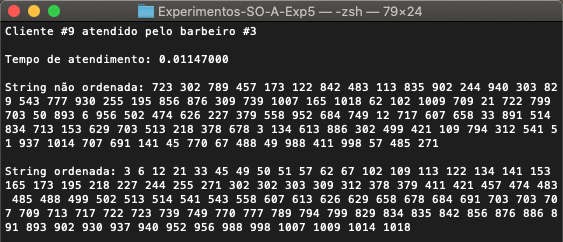


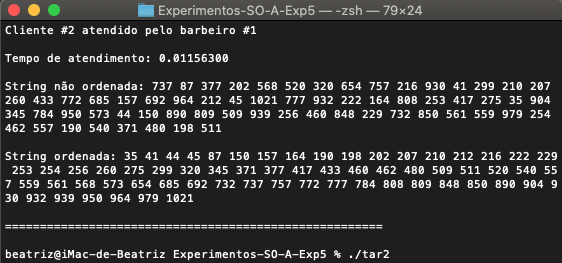








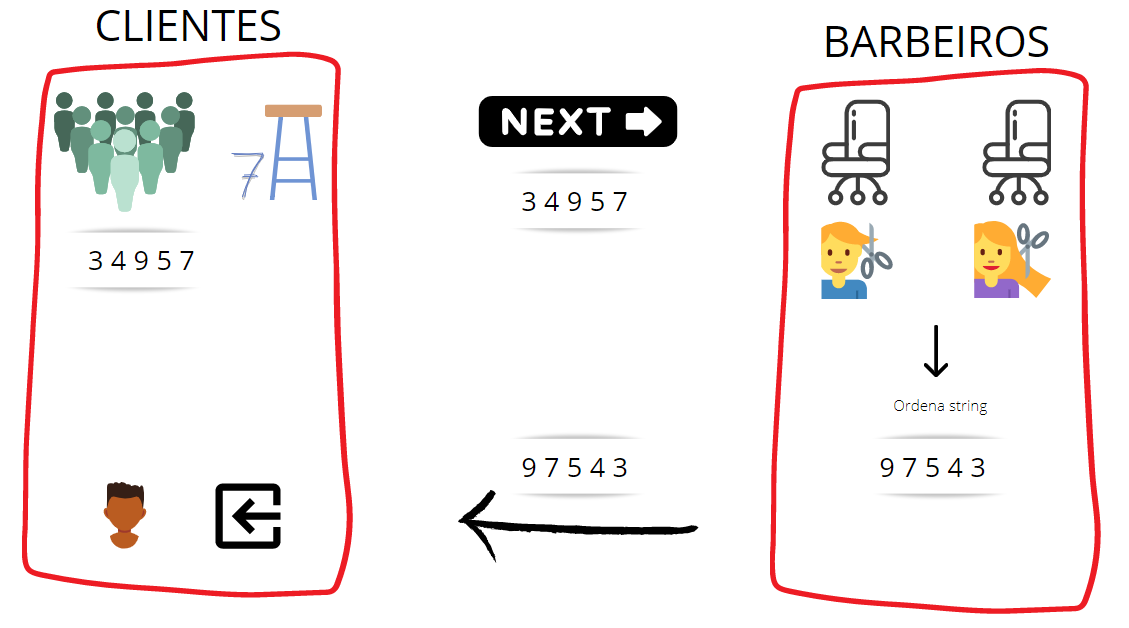


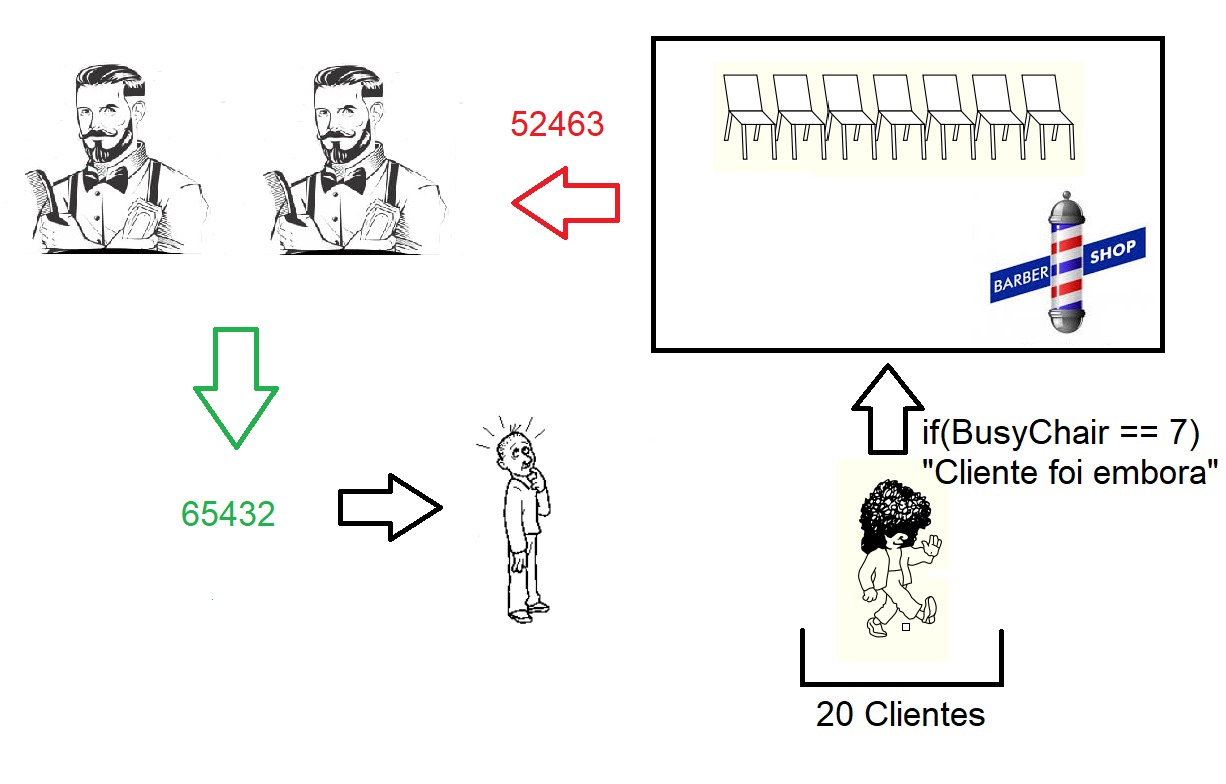


**Análise dos resultados**

**Programa 1**

Inicialmente mostraremos dois esquemas para exemplificar a implementação do primeiro programa do barbeiro dorminhoco.





São criados vinte processos clientes e dois processos barbeiros. Os processos clientes verificavam se havia alguma cadeira de espera disponível. Caso não houvesse, o cliente saia da barbearia. Caso contrário, o processo gerava um vetor de inteiros com números aleatórios entre 2 e 1023, com tamanho máximo de 300 (a seed era diferente para cada processo, gerando valores difentes para cada cliente). Após isso era necessário converter esse vetor de inteiros em um vetor de caracteres, ou seja, uma string, que seria enviada ao barbeiro. O barbeiro por sua vez, pegava a mensagem enviada pelo cliente, convertia a string para um vetor de inteiros novamente, para assim poder ser ordenado de forma decrescente. Para o ordenação foi escolhido o método bubblesort, uma vez que a quantidade de elementos a serem ordenados não era alta, causando um baixo impacto no tempo de execução do programa. Após a ordenação era necessário converter o vetor de inteiros para string novamente, para poder enviar outra mensagem ao cliente. O cliente por sua vez, ao receber uma mensagem, imprimia as informações necessárias: número do barbeiro que o atendeu, seu próprio número, a string a ser ordenada, a string ordenada e o tempo de atendimento. Para termos certeza de que cada barbeiro devolveria a mensagem ao cliente certo, utilizou-se dos tipos de mensagens da fila. O tipo de mensagem de para o envio ao barbeiro era o tipo 1 e do envio do barbeiro para o cliente utiliza-se o número do cliente mais 50 (numero do cliente + 50) com tipo de mensagem (a informação do número do cliente era passada na struct de envio). Dessa forma garantíamos que cada barbeiro enviaria a mensagem para o cliente correto.

A respeito dos resultados, podemos observar que esta parte do experimento foi realizada com sucesso. Verifica-se que a string de cada processo filho foi ordenada de forma decrescente, como esperado e que alguns clientes também não foram atendidos.

Foram feitas 4 rodadas de teste (execução) para podemos observar melhor alguns aspectos do programa e para uma melhor análise da tarefa, além de alguns gráficos mostrados abaixo:



Tabela 1 - Tempo de atendimento por cliente na segunda execução

Tabela 2 - Tempo de atendimento por cliente na segunda execução

Tabela 3 - Tempo de atendimento por cliente na terceira execução

Tabela 4 - Tempo de atendimento por cliente na quarta execução

Nos gráficos acima podemos ver que a ordem dos clientes não afetou o seu tempo, ou seja, os primeiros clientes não necessariamente foram os primeiros a serem atendidos. Isso se deve ao fato de não sabermos como o SO escalona os processos. Os primeiros processos escalonados pelo SO serão os primeiros a entrarem na fila de mensagens e, portanto, os primeiros a serem escalonados. Alguns outros fatores como programas externos rodando em background, podem interferir nos resultados, atrasando ou adiantando o atendimento dos clientes.

Também podemos verificar nos gráficos que nem todos os vinte clientes foram atendidos, como mostrado na análise abaixo:

Tabela 5 - Números de clientes não atendidos a cada rodada de execução

Neste gráfico podemos verificar a quantidade de clientes que não foram atendidos a cada execução. Na média foram 2,75 clientes não atendidos por execução. Isso mostra que em alguns casos, alguns clientes (processos) não são terminados em tempo suficiente para outros serem atendidos.

Um único processo cliente precisa enviar uma mensagem ao barbeiro que, por sua vez, ordenará a string e enviará outra mensagem ao cliente e este imprime as informações necessárias. Portanto, a quantidade de tarefas que um único processo precisa realizar é muito alta e por isso, alguns clientes não são atendidos.

**Programa 2**

**Conclusão**

Como citado anteriormente, o experimento possibilitou o uso de todos os mecanismos vistos no semestre. Foi importante para que os conceitos fossem melhor compreendidos pelos alunos.

No primeiro programa, foi possível a revisão de funções relacionadas a fila de mensagens e memória compartilhada como: shmget(), shmat(), msgget(), entre outras. No segundo programa, todas as funções relacionadas a threads, mutex e semáforos também puderam ser revisadas, contribuindo para a conclusão do experimento