PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

ENGENHARIA DE COMPUTAÇÃO

SISTEMAS OPERACIONAIS A

Carlos Henrique Vieira Marques – 18720367

Erick Matheus Lopes Pacheco – 18711630

Hiago Silva Fernandes – 18726455

João Henrique Pereira – 18712919

Leonardo Sanavio – 18054395

EXPERIMENTO #5

BARBEIRO DORMINHOCO, IPC, THREAD, MUTEX E MEMÓRIA COMPARTILHADA

CAMPINAS

MAIO 2020

SUMÁRIO

1. INTRODUÇÃO

O experimento visa solidificar os conhecimentos previamente adquiridos propondo uma problemática e uma solução que envolve mutuamente – e não separadamente como nos experimentos anteriores – os mecanismos de processos e memória, tal como a troca de mensagens entre processos, os threads, o uso de memória compartilhada, semáforos e etc.

Considerando isso, em ambas as partes do experimento será implementado o algoritmo do “Barbeiro Dorminhoco”, que consiste nas seguintes condições:

- Se a barbearia não possui clientes, o barbeiro dorme;
- Se a cadeira do barbeiro estiver livre, um cliente pode ser atendido;
- O cliente aguarda o barbeiro em caso de uma cadeira de espera livre;
- Se não houver onde sentar, o cliente vai embora.

Na primeira parte é implementada a problemática do Barbeiro Dorminhoco utilizando memória compartilhada e fila de mensagens (IPC) através de processos filhos. Sendo a fila bloqueante, não é necessário a implantação de semáforos para tratar o problema de race condition entre clientes e barbeiros. O uso de semáforo ocorre apenas para fila de clientes que é compartilhada entre todos os processos do programa.

Na parte dois a implementação ocorre por meio do uso de threads, onde essa será a substituta da fila e a utilização de mutex para que não haja problemas de *race conditions.*

2. PERGUNTAS

2.1. REFERENTES AOS SISTEMAS UNIX

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

O recurso comum para uso de exclusão mútua (semáforos) serve para garantir que o Barbeiro não durma para sempre (deadlocks) ou que 1 barbeiro não atenda mais de 1 cliente ao mesmo tempo (race condition).

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

recurso comum?

Os clientes e os barbeiros irão acessar o recurso comum através de Memória compartilhada, visa-se a implementação de uma Fila Simples, permitindo atendimento FIFO (First In, First Out -primeiro que chega, primeiro que sai) para que ocorra um sincronismo entre origem e destino.

Pergunta 3: Como os números foram colocados na string?

Sabemos que cada posição de uma string (vetor de chars) possui um byte de tamanho alocado na memória, e temos uma limitação de 8096 bytes por mensagem da fila de mensagens. Considerando que devemos armazenar 1024 números distintos, poderíamos criar um vetor de char com 1024 * sizeof(int) posições, armazenar os números e lê-los utlizando cast para int, porém estouraria o tamanho da mensagem por termos que armazenar dois destes (um não ordenado e outro ordenado). Por definição, com um byte podemos representar 256 (2^8) caracteres, sendo assim, podemos armazenar os números de 0 a 255 em 1 byte de uma posição de vetor unsigned sem obter problemas para obtê-los. Como armazenar 1024? Compressão. Números maiores que 255 nós comprimimos com as seguintes lógicas matemáticas: Armazenamos Número % 256 = Resto no vetor de chars visto que o resto sempre será menor que 256, e armazenamos Número / 256 = Quociente num vetor de inteiros, para "descomprimir" o número, basta utilizarmos da expressão

Número = (Quociente * 256) + Resto que voltamos ao número original, isso permite utilizarmos um vetor de chars de 1024 posições, ou seja, conseguimos representar 1024 números distintos utilizando apenas 1KB.

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

Parte1: O Barbeiro terá acesso aos valores através do comando msgrcv() presente na função cuthair(), que irá acessar o conteúdo da fila que está na memória compartilhada. A fila foi alimentada pelo cliente que gerou a string a ser organizada e enviou para esta através do comando msgnd().

Parte 2: Na parte 2 o Barbeiro utiliza a mesma função cuthair(), porém está agora possui uma struct contendo os dados, que serão compartilhados entre as threads, e um semáforo para exclusão mútua.Sendo assim, o Barbeiro adquire e trata a string gerada pelo cliente.

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

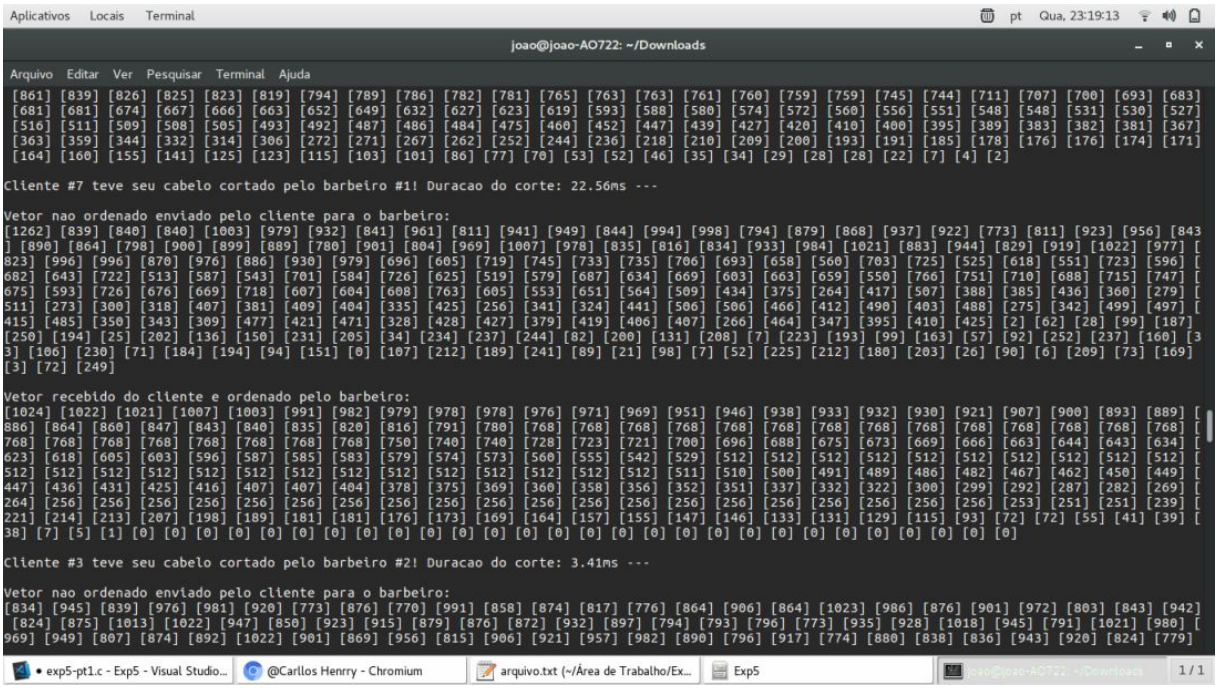
Parte1:Através da função apreciate_hair() o cliente usará o comando mgsrcv() para receber os dados (tempo de corte, número do cliente, qual barbeiro o atendeu, string gerado e por fim a string convertida e organizada) pertencentes a Fila de mensagem.

Parte2: Também utiliza a função apreciate_hair(), porém com mudanças para receber os dados. Através do uso de semáforo e a implementação de uma struct, os dados foram compartilhados entre os threads permitindo que os clientes tivessem acesso ao resultado final do processo.

3. RESULTADOS

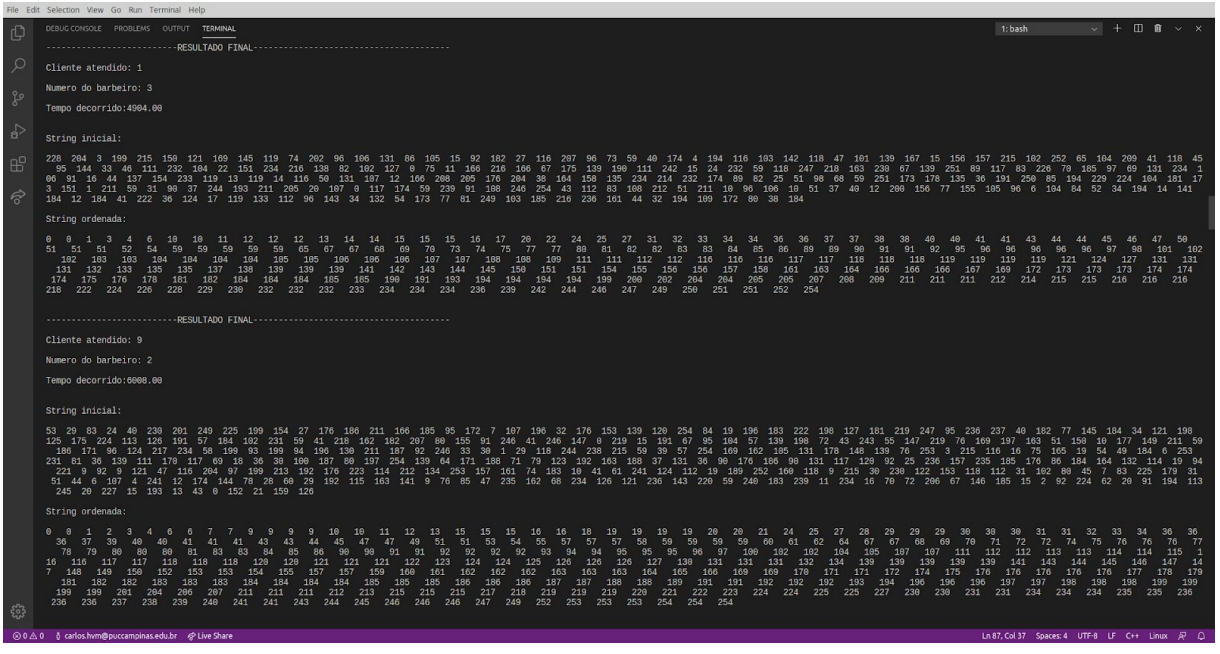
A seguir, os resultados obtidos de ambas as partes do experimento, contendo informações detalhadas das saídas.

3.1. REFERENTES A PARTE 1



(Imagem retratando o resultado da compilação da parte 1, apresenta o número do cliente, número do barbeiro que atendeu o cliente, a string gerada pelo cliente e o vetor de inteiro organizado pelo Barbeiro.)

3.2. REFERENTES A PARTE 2



(Imagem retratando o resultado da compilação da parte 2, apresenta o número do cliente, número do barbeiro que atendeu o cliente, a string gerada pelo cliente e o vetor de inteiro organizado pelo Barbeiro.)
OBS.: Um arquivo em .txt foi anexado juntamente com os demais arquivos, com todo o resultado.

4. ANÁLISE DOS RESULTADOS

4.1. REFERENTE A PARTE 1

Na primeira parte, os resultados revelam o conjunto das características apresentadas pelos experimentos anteriores, nele é possível perceber: a meticulosa ação do mecanismo de escalonamento do SO, a eficácia do sistema de semáforos para impedir race conditions no segmento de memória compartilhada e a atuação da ferramenta de espera bloqueada utilizada pelo sistema de fila de mensagens IPC do System-V.

Todos estes mecanismos trabalhando em conjunto impactam diretamente no tempo em que um cliente leva desde entrar na região crítica – e sentar-se na cadeira de espera e sinalizar isso – a até enviar a mensagem (no caso, uma estrutura que contabiliza a medição do tempo juntamente com um vetor de tamanho rand com números aleatórios) para o barbeiro, que, por sua vez, retornará essa mensagem com o vetor ordenado para o cliente que, ao receber, entra novamente numa região crítica que exibe os resultados.

Então temos que:

• Os clientes "disputam" entre si para sentarem numa cadeira livre e sofrem influência direta do escalonamento de CPU;

• O cliente que chega "primeiro" é responsável pela exclusão mútua travando o mutex, de modo a impedir que outros sentem na mesma cadeira de espera;

• Enquanto um cliente senta na cadeira de espera, outro pode, paralelamente, entrar na fila de mensagens que será atendida (recebida) por ordem de chegada pelo barbeiro, através de uma espera bloqueada;

• Para exibir os resultados, um segundo mutex é utilizado pelos clientes que recebem a mensagem de volta do barbeiro, de modo a impedir que os prints sejam interrompidos e saiam ilegíveis;

4.2. REFERENTE A PARTE 2

A parte 2 do experimento em relação à parte 1, sofreu mudanças, como a implementação da threads, o uso de mutex com threads e semáforos para clientes e barbeiros.

O programa gerou diversos debates e muita dificuldade para equipe, ainda mais pela interpretação do comportamento dos threads e semáforos. Inicialmente usamos 1 semáforo para clientes e 1 para barbeiro, porém ocorria problemas em que a geração da string ficava travada ou era atropelada por outra thread, devido a ter só um semáforo para os clientes. Então, foi alterado para um vetor de semáforos de clientes, solucionando o problema. A string de [char] que era o logicamente o cabelo gerado pelo próprio cliente, possui valores variando entre **0 a 255** e seu tamanho variando entre **0 a 1023,** sua conversão é feita através de um cast em *unsigned char* (**unsigned->** para evitar número negativos) para um *int*.

A organização do vetor foi realizada através do algoritmo Selection-Sort. O Selection-Sort foi escolhido devido a sua consistência no gasto de tempo para organizar o vetor. (O Quicksort poderia ser usado, porém devido o comportamento dos threads e dos processos não serem padronizados, o tempo de resposta do quick poderia ser muito aleatório). Por fim, temos o cliente retornando seu tempo no salão, o Barbeiro que o atendeu, a string que ele gerou (cabelo) e a string convertida em inteiro e organizada.

5. CONCLUSÃO

A primeira parte do experimento permitiu a consolidação do aprendizado adquirido nos experimentos anteriores, reforçando o funcionamento da espera bloqueada e da fila de mensagens, bem como da exclusão mútua. A equipe teve problemas nessa parte do projeto que não foi possível resolver devido às complexidades para debugar e executar por etapas um programa que executa muitos processos o que prejudicou no desenvolver do projeto e compreensão desta parte.

Assim como na parte 1, a parte 2 do experimento também proporcionou alguns desafios. Foi implementado um buffer para armazenar os dados que seriam gerados e impressos pelo programa, como o tempo gasto no corte, o vetor ordenado, número do cliente e número do barbeiro.

A grande dificuldade, foi estabelecer a sincronia entre os threads. Garantir a associação correta entre o barbeiro e o cliente, e que os dados a serem gerados e impressos pelos clientes não se misturassem, além da execução correta do programa, passando pela *generate_hair(), cut_hair() e apreciate_hair* nessa ordem. Devido a isso, diversas impressões ficavam confusas. Era difícil distinguir se o erro decorria de uma má geração e ordenação dos valores, ou se a concorrência dos threads no momento de armazenar e imprimir, que comprometia os dados. Algo a ser também ponderado foi a geração dos valores no cliente. Foi usado a função *rand()* e *srand(time(NULL))* para gerar valores aleatórios. Entretanto, dado a velocidade das threads, as sementes geradas na *srand(time(NULL))* eram as mesmas, provocando repetitividade nos valores gerados, aparentando a princípio que os valores eram equivocados.

Uma alteração simples em relação a parte 1, foi garantir que o cliente voltasse mais tarde caso a barbearia estivesse cheia, sendo feita a partir de um laço *while()* na função *customer().*