Diário de bordo: Trabalho Prático 2

Projeto Simulador de Caixa de Supermercado com Fila Prioritária

Equipe 4:

- Clarissa Eri Morita
- Giovanna Dornelles Barrichello
- Isabella Vicente
- João Victor Mota
- Peterson Fontinhas

Vídeo demonstrando as funcionalidades do sistema:

[Acesse o vídeo através desse link aqui]

Semana 1: Estrutura Básica e Funções Essenciais

Objetivo da Semana: Estabelecer as estruturas de dados fundamentais (Cliente e Fila), e implementar as operações básicas de fila (criação, inserção, atendimento simples e impressão de uma única fila), além da função de cálculo de tempo.

Giovanna - Semana 1

Minhas Responsabilidades: Nesta primeira semana, foquei na base do nosso sistema de filas. Minhas principais responsabilidades foram a definição e implementação da estrutura principal da Fila e das operações mais elementares: criarFila() e inserirClienteFila().

Dificuldades Encontradas e Resolução: A maior dificuldade foi garantir a robustez de inserirClienteFila(), especialmente ao lidar com o caso de uma fila vazia. Se o primeiro e ultimo ponteiros não fossem atualizados corretamente quando o primeiro cliente era inserido, teríamos problemas sérios. A solução foi adicionar uma verificação explícita if(fila->primeiro == NULL) para tratar esse cenário inicial, onde o novo nó se torna ambos, primeiro e ultimo. Para os demais casos, a inserção no final foi mais direta.

Código Comentado:

```
void inserirClienteFila(Fila *fila, Cliente *cliente){

    Nodecliente *novo = malloc(sizeof(Nodecliente));
```

```
    if (novo != NULL){
        novo->cliente =
        cliente; novo->proximo
        = NULL;
        // Verifica se a fila está
        vazia. if(fila->primeiro ==
        NULL){
            // Se estiver vazia, o novo nó se torna tanto o primeiro
quanto o último.
            fila->primeiro =
            novo; fila->ultimo =
            novo;
        }
        // Se a fila NÃO estiver vazia.
        else{
            // O próximo do nó que era o último agora aponta para o
novo nó.
            fila->ultimo->proximo = novo;
            // O novo nó se torna o novo último da
            fila. fila->ultimo = novo;
        }
        // Incrementa o tamanho da
        fila. fila->tam++;
    }
    else {
        printf("\nErro ao alocar memória para o nó do cliente!\n");
    }
}
```

A função inserirClienteFila() adiciona um novo cliente ao final de uma fila. Ela aloca memória para um novo nó, associa o cliente a esse nó e o insere no final da fila, mantendo a ordem FIFO (First-In, First-Out). Se a fila estiver vazia, o novo nó se torna tanto o primeiro quanto o último. Após a inserção, o tamanho da fila é incrementado.

Clarissa Morita - Semana 1

Minhas Responsabilidades: Minha função principal nesta semana foi implementar como os clientes seriam "atendidos" (removidos da fila) e como poderíamos visualizar o estado da fila. Para isso, desenvolvi atenderCliente() e imprimirFila(), e preparei a imprimirAmbasFilas() para quando tivéssemos a estrutura de duas filas.

Dificuldades Encontradas e Resolução: A maior dificuldade inicial em atenderCliente() foi entender o fluxo de liberação de memória. Se eu liberasse o Cliente* logo após removê-lo da fila, não teríamos como usá-lo para as estatísticas futuras. A solução foi um acordo com o João Victor (que faria a lista de atendidos): atenderCliente só liberaria o Nodecliente (o invólucro), e o ponteiro Cliente* seria repassado para a lista de atendidos. Além disso, garantir que imprimirFila funcionasse corretamente para filas vazias e não vazias exigiu um bom controle de ponteiros.

Código comentado:

```
Nodecliente* atenderCliente(Fila *fila, ListaAtendidos
*listaAtendidos){

    Nodecliente *atendido = NULL;

    if(fila->primeiro != NULL){

        atendido = fila->primeiro;

        fila->primeiro = atendido->proximo;

        fila->tam--;

        atendido->cliente->tempo =
calcularTempo(atendido->cliente->itens);

        printf("\nCliente %s atendido.\nTempo de atendimento: ",
atendido->cliente->nome);

        exibirTempoFormatado(atendido->cliente->tempo);
```

```
        printf(".\n");

        adicionarClienteAtendido(listaAtendidos, atendido->cliente);

        free(atendido);

        return NULL;

    }

    // Se a fila estiver vazia.

    else {

        printf("\nFila vazia!\n");

    }

    return atendido;

}
```

A função atenderCliente() simula a remoção de um cliente da fila para atendimento. Ela verifica se a fila não está vazia, remove o primeiro cliente (seguindo a lógica FIFO), atualiza o tamanho da fila, calcula o tempo de atendimento do cliente com base em seus itens, e exibe essa informação formatada. O cliente atendido é então adicionado a uma lista separada para futuras estatísticas. Por fim, a memória do nó da fila é liberada, mas não os dados do cliente, que são transferidos para a lista de atendidos.

Peterson Fontinhas - Semana 1

Minhas Responsabilidades: Minha tarefa inicial foi essencial para as futuras estatísticas e a lógica de atendimento: criar a função calcularTempo(). Também colaborei na definição da estrutura Cliente em clientes.h, pois ela seria a base para o cálculo.

Dificuldades Encontradas e Resolução: A calcularTempo em si é bem direta, mas o desafio foi pensar em como esse valor seria armazenado e utilizado. Inicialmente, pensei em retornar o tempo, mas percebemos que ele precisava ser um atributo do Cliente para persistir e ser coletado para estatísticas. A resolução veio com a inclusão do campo int tempo; na struct

Cliente e a coordenação com a Clarissa para que atenderCliente atribuísse esse valor.

Semana 2: Expansão com Duas Filas, Cadastro Inteligente e Coleta de Dados

Objetivo da Semana: Refatorar o sistema para suportar duas filas (comum e preferencial), implementar a lógica de cadastro que direciona o cliente à fila correta, e iniciar o armazenamento de clientes atendidos para as futuras estatísticas.

Isabella Luiza - Semana 2

Minhas Responsabilidades: Esta semana foi crucial para a lógica de negócio do supermercado. Fui responsável por cadastrarCliente() (a versão interativa) para direcionar os clientes à fila correta, criar uma versão de demonstração cadastrarClienteDemo() e começar a estrutura de gerarEstatisticas(), focando na coleta inicial dos dados.

Dificuldades Encontradas e Resolução: Em cadastrarCliente(), o maior desafio foi o tratamento da entrada do usuário, especialmente a mistura de scanf e fgets, que podia deixar o \n no buffer e causar problemas na próxima leitura. Solucionei isso usando getchar() para limpar o buffer e strcspn() para remover o \n do fgets. A lógica de if (cliente->prioridade) foi a chave para enviar o cliente à fila certa. Para gerarEstatisticas(), o foco foi percorrer a nova ListaAtendidos e somar os tempos e contar os clientes por tipo, o que exigiu atenção aos ponteiros e tipos de variáveis para evitar overflow.

João Victor - Semana 2

Minhas Responsabilidades: Com a necessidade de coletar dados para estatísticas, minha tarefa foi criar uma nova estrutura para armazenar os clientes já atendidos. Implementei criarListaAtendidos() e adicionarClienteAtendido(). Além disso, para a apresentação dos tempos, desenvolvi exibirTempoFormatado().

Dificuldades Encontradas e Resolução: A criação de ListaAtendidos e adicionarClienteAtendido foi um desafio semelhante ao da fila, mas com a particularidade de que os clientes já vêm de uma fila. Garanti que o ponteiro do Cliente* fosse corretamente transferido e que a nova lista funcionasse de forma independente. exibirTempoFormatado exigiu atenção aos detalhes para

a formatação e os plurais (minuto/minutos, segundo/segundos). A solução foi usar a matemática simples de divisão e módulo, e uma condicional para o plural.

Código Comentado:

```
void adicionarClienteAtendido(ListaAtendidos *lista, Cliente *cliente)
{

    NodeAtendido* novoNo = malloc(sizeof(NodeAtendido));

    if (novoNo != NULL) {

        // Associa o cliente ao novo

        nó. novoNo->cliente = cliente;

        // O próximo ponteiro do novo nó é

        NULL. novoNo->proximo = NULL;



        // Verifica se a lista de atendidos está vazia.

        if (lista->primeiro == NULL) {

            // Se estiver vazia, o novo nó se torna o primeiro e o
último.

            lista->primeiro =

            novoNo; lista->ultimo =

            novoNo;

        } else {

            // Se não estiver vazia, adiciona o novo nó ao final da
lista.

            lista->ultimo->proximo =

            novoNo; lista->ultimo = novoNo;

        }

        // Incrementa o contador total de clientes

        atendidos. lista->total_clientes_atendidos++;
```

```
    } else {

        printf("\nErro ao alocar memória para o nó de cliente
atendido!\n");

    }

}
```

A função adicionarClienteAtendido() insere um cliente recém-atendido em uma lista específica de clientes já processados. Ela aloca um novo nó para a lista de atendidos, atribui o cliente a esse nó e o adiciona ao final da lista. Se a lista estiver vazia, o novo nó se torna o primeiro e o último. Um contador de clientes atendidos é incrementado a cada adição.

Semana 3: Lógica de Atendimento Avançada e Gestão de Memória

Objetivo da Semana: Implementar a lógica complexa de atendimento alternado, garantir a correta liberação de memória e realizar os testes finais de integração do sistema.

Peterson Fontinhas - Semana 3

Minhas Responsabilidades: Esta foi a semana mais desafiadora. Fui o principal responsável por dar vida à lógica de atendimento alternado (2 comuns / 1 preferencial) na função handleOpcaoDois(). Além disso, tive a crítica e fundamental tarefa de implementar handleOpcaoZero() para garantir que todo o programa liberasse a memória alocada dinamicamente, prevenindo vazamentos.

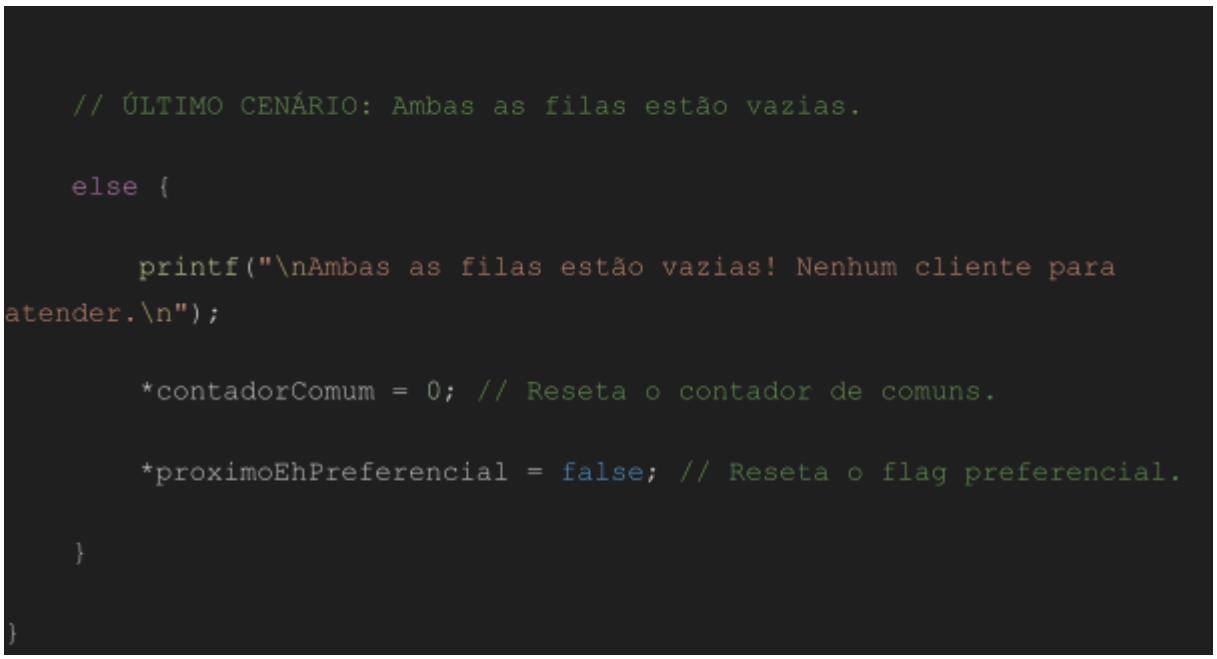
Dificuldades Encontradas e Resolução: Em handleOpcaoDois(), a complexidade veio dos múltiplos cenários: a vez do preferencial com fila preferencial vazia, a vez do comum com fila comum vazia, e a necessidade de forçar um preferencial se o limite de comuns fosse atingido. A solução foi construir uma sequência de if-else if aninhados, com contadorComum e proximoEhPreferencial como variáveis de estado. A liberação de memória em handleOpcaoZero() também foi um ponto de atenção, pois precisei garantir que cada Nodecliente, NodeAtendido e, o mais importante, cada Cliente* fosse liberado **exatamente uma vez** para evitar duplas liberações ou

vazamentos. Percorrer cada lista e liberar seus componentes um a um, e por último a estrutura da fila/lista em si, foi a estratégia.

Código comentado:

```
void handleOpcaoDois(Fila *filaComum, Fila *filaPreferencial,
ListaAtendidos *clientesAtendidos, int *contadorComum, const int
LIMITE_COMUM, bool *proximoEhPreferencial) {

    // Lógica de atendimento alternado: 2 comuns, 1 preferencial

    // PRIMEIRO CENÁRIO: Tentar atender um cliente preferencial se for
a vez dele E houver clientes na fila preferencial.

    if (filaPreferencial->tam > 0 && *proximoEhPreferencial) {

        printf("\nAtendendo cliente da Fila Preferencial...\n");

        atenderCliente(filaPreferencial, clientesAtendidos);

        *contadorComum = 0;

        *proximoEhPreferencial = false;

    }



    // SEGUNDO CENÁRIO: Tentar atender um cliente comum se NÃO for a
vez do preferencial E o limite de comuns não foi atingido E houver
clientes na fila comum.

    else if (filaComum->tam > 0 && *contadorComum < LIMITE_COMUM) {

        printf("\nAtendendo cliente da Fila Comum...\n");

        atenderCliente(filaComum, clientesAtendidos);

        (*contadorComum)++;

        // Verifica se o limite de clientes comuns para o ciclo foi
atingido.

        if (*contadorComum == LIMITE_COMUM) {

            *proximoEhPreferencial = true; // Se atingiu o limite, o
próximo atendimento DEVE ser preferencial.
```

```
        }

    }

    // TERCEIRO CENÁRIO: Se o limite de comuns foi atingido OU não há mais clientes comuns disponíveis,

    // mas há clientes preferenciais esperando. Força o atendimento de um preferencial.

    else if (filaPreferencial->tam > 0) {

        printf("\nAtendendo cliente da Fila Preferencial (prioridade ou limite de comuns atingido)...\n");

        atenderCliente(filaPreferencial, clientesAtendidos);

        *contadorComum = 0; // Reseta o contador de comuns.

        *proximoEhPreferencial = false; // Volta a priorizar comuns no próximo ciclo.

    }

    // QUARTO CENÁRIO: Se não há clientes preferenciais (ou não é a vez deles), mas há clientes comuns esperando.

    // Continua atendendo clientes comuns.

    else if (filaComum->tam > 0) {

        printf("\nAtendendo cliente da Fila Comum (sem preferenciais disponíveis)...\n");

        atenderCliente(filaComum, clientesAtendidos);

        (*contadorComum)++; // Incrementa o contador de comuns.

        // Verifica se o limite de clientes comuns foi atingido.

        if (*contadorComum == LIMITE_COMUM) {

            *proximoEhPreferencial = true; // Se atingiu o limite, o próximo DEVE ser preferencial (quando houver).

        }

    }
```



```
    // ÚLTIMO CENÁRIO: Ambas as filas estão vazias.
    else {
        printf("\nAmbas as filas estão vazias! Nenhum cliente para
atender.\n");
        *contadorComum = 0; // Reseta o contador de comuns.
        *proximoEhPreferencial = false; // Reseta o flag preferencial.
    }
}
```

A função handleOpcaoDois() implementa a complexa lógica de atendimento alternado do supermercado: a cada 2 clientes comuns atendidos, 1 cliente preferencial deve ser atendido. Ela verifica a disponibilidade de clientes em ambas as filas (filaComum e filaPreferencial) e o estado atual do ciclo de atendimento (contadorComum e proximoEhPreferencial). A função prioriza o atendimento de clientes preferenciais quando é a vez deles ou quando não há mais clientes comuns disponíveis no ciclo. Se há clientes comuns e o limite de 2 ainda não foi atingido, atende-se um comum. Os contadores são atualizados para manter a alternância correta. Caso ambas as filas estejam vazias, uma mensagem é exibida.