1. Justificativa de Design: Por que a estrutura de dados utilizada é uma estrutura eficiente para implementar o comportamento do escalonador?

A estrutura de dados utilizada no projeto é uma **fila circular** (FilaCircularDeProcessos), implementada com nós (Node.java) que contêm objetos Processo. Essa escolha é eficiente para o escalonador por várias razões:

- **Acesso cíclico e ordenado**: A fila circular permite um acesso contínuo e cíclico aos processos, o que é ideal para um escalonador que precisa processar tarefas repetidamente em ordem, como em um sistema de escalonamento por prioridades. Isso garante que todos os processos na fila sejam eventualmente atendidos.
- **Eficiência em inserção e remoção**: Inserções no final e remoções no início da fila têm complexidade O(1) em uma fila circular bem implementada, o que é crucial para gerenciar processos rapidamente durante a execução de ciclos.
- **Separação por prioridades**: O uso de três filas distintas (Alta, Média, Baixa) mais uma fila de bloqueados permite organizar processos por prioridade sem a necessidade de reordenar constantemente, reduzindo o custo computacional.
- **Flexibilidade para bloqueio/desbloqueio**: A fila circular facilita mover processos entre filas (por exemplo, da fila de alta prioridade para a fila de bloqueados) sem grandes reorganizações, já que os nós podem ser facilmente realocados.

Essa estrutura é adequada porque suporta as operações frequentes do escalonador (adicionar, remover, executar e bloquear processos) de forma eficiente e mantém a lógica de priorização clara.

---

2. Análise da Complexidade (Big-O) das Operações na Implementação

Com base na descrição do projeto, as principais operações na implementação do escalonador (FilaCircularDeProcessos e Scheduler) têm as seguintes complexidades:

- **Adicionar um processo (adicionarProcesso)**: O(1)
    - Inserir um processo no final da fila circular é uma operação constante, pois apenas ajusta os ponteiros do último nó.
- **Remover um processo (removerProcesso)**: O(n)
    - Para remover um processo específico por ID, é necessário percorrer a fila até encontrá-lo, o que leva O(n) no pior caso, onde n é o número de processos na fila.
- **Atualizar um processo**: O(n)
    - Similar à remoção, atualizar um processo requer buscar pelo ID na fila, o que tem complexidade O(n).
- **Buscar um processo (buscarProcesso)**: O(n)
    - A busca por ID na fila circular exige percorrer a lista, resultando em O(n).
- **Executar um ciclo (executarCiclo)**: O(1) para seleção, O(n) para manipulação
    - Selecionar o próximo processo da fila de alta prioridade (ou média/baixa, se aplicável) é O(1), pois acessa o início da fila. Porém, se o processo for

movido para a fila de bloqueados ou concluído, pode haver uma busca ou remoção, elevando a complexidade a O(n) em alguns casos.
- **Ver listas**: O(n)
    - Listar todos os processos de uma fila requer percorrer todos os nós, com complexidade O(n).
- **Salvar/Carregar processos (RepositorioProcessos)**: O(n)
    - Ler ou escrever processos no arquivo Processo.txt exige processar cada processo, resultando em O(n).

A complexidade geral é dominada por operações de busca e remoção (O(n)), mas operações frequentes como adicionar e executar ciclos são otimizadas (O(1)).

---

3. Análise da Anti-Inanição: Como a Lógica Garante a Justiça no Escalonamento e Qual o Risco se Essa Regra Não Existisse?

A lógica do escalonador implementa uma estratégia anti-inanição ao alternar entre filas de prioridade após 5 ciclos consecutivos de execução de processos de alta prioridade. Isso garante justiça no escalonamento da seguinte forma:

- **Mecanismo de justiça**: O escalonador prioriza processos de alta prioridade, mas, após 5 ciclos, verifica as filas de média e baixa prioridade. Isso impede que processos de menor prioridade sejam completamente ignorados, garantindo que todos eventualmente sejam executados.
- **Rotação de prioridades**: A alternância entre filas assegura que processos de média e baixa prioridade tenham oportunidade de execução, mesmo em cenários com muitos processos de alta prioridade.
- **Prevenção de inanição**: Sem essa regra, processos de baixa prioridade poderiam nunca ser executados se a fila de alta prioridade estivesse constantemente recebendo novos processos. Isso levaria à **inanição** (starvation), onde processos menos prioritários ficam indefinidamente esperando.

Risco sem a regra de anti-inanição:

- Processos de média e baixa prioridade seriam negligenciados, causando atrasos indefinidos ou falhas em sistemas onde todos os processos são importantes (mesmo que com prioridades diferentes).
- Isso poderia levar a um sistema injusto, onde apenas processos de alta prioridade são atendidos, comprometendo a eficiência global e a experiência do usuário em aplicações que dependem de todos os processos.

---

4. Análise do Bloqueio: Ciclo de Vida de um Processo que Precisa do "DISCO"

O ciclo de vida de um processo que precisa do recurso "DISCO" no escalonador segue estas etapas:

1. **Criação e Adição**: O processo é criado com o atributo recurso_necessario = "DISCO" e adicionado à fila correspondente à sua prioridade (Alta, Média ou Baixa) via FilaCircularDeProcessos.
2. **Seleção para Execução**: Durante um ciclo (executarCiclo), o processo é selecionado da sua fila de prioridade (começando pela Alta, depois Média e Baixa, conforme a lógica de priorização).
3. **Verificação do Recurso**: O escalonador verifica se o processo requer "DISCO". Como o recurso "DISCO" causa bloqueio, o processo é movido da sua fila de prioridade para a **fila de bloqueados**.
4. **Estado Bloqueado**: Na fila de bloqueados, o processo aguarda até que o escalonador decida desbloqueá-lo (geralmente no próximo ciclo, dependendo da implementação).
5. **Desbloqueio**: Em um ciclo subsequente, o escalonador verifica a fila de bloqueados e move o processo de volta para sua fila de prioridade original (Alta, Média ou Baixa), mantendo sua ordem relativa.
6. **Execução**: Uma vez desbloqueado, o processo é novamente selecionado para execução. Seu número de ciclos necessários é decrementado. Se chegar a zero, o processo é concluído e removido; caso contrário, retorna ao final da sua fila de prioridade.
7. **Conclusão ou Repetição**: O processo repete o ciclo de execução (e potencial bloqueio, se "DISCO" for necessário novamente) até que todos os ciclos necessários sejam executados, quando então é removido do sistema.

Essa jornada garante que processos que requerem "DISCO" sejam temporariamente suspensos, mas não perdidos, e retornem à execução assim que possível.

---

5. Ponto Fraco: Principal Gargalo de Performance e Melhoria Teórica

Principal gargalo de performance:

O maior gargalo no escalonador está nas operações que exigem **busca por ID** nas filas circulares, como removerProcesso, atualizarProcesso e buscarProcesso, que têm complexidade **O(n)**. Isso ocorre porque a fila circular, implementada como uma lista encadeada, requer percorrer todos os nós para localizar um processo específico. Em sistemas com muitos processos, isso pode degradar significativamente o desempenho.

Impacto:

- Em cenários com filas longas, operações como remover ou atualizar processos tornam-se lentas.
- A busca repetitiva por ID durante a execução de ciclos (por exemplo, para verificar ou mover processos) aumenta o custo computacional.

Melhoria teórica:

Para mitigar esse gargalo, uma solução seria **integrar uma tabela de hash** (como um HashMap em Java) para indexar processos por ID, mantendo a fila circular para a lógica de escalonamento. A proposta seria:

- **Implementação**:
    - Adicionar um HashMap<Integer, Node> no Scheduler ou FilaCircularDeProcessos, onde a chave é o ID do processo e o valor é o nó correspondente na fila circular.
    - Ao adicionar um processo, atualizar o HashMap com o ID e o nó.
    - Para operações como busca, remoção ou atualização, usar o HashMap para acessar o nó diretamente em O(1), em vez de percorrer a fila.
- **Benefícios**:
    - **Busca e remoção em O(1)**: Acessar um processo por ID no HashMap reduz a complexidade de O(n) para O(1).
    - **Manutenção da lógica de escalonamento**: A fila circular continua gerenciando a ordem de execução, enquanto o HashMap otimiza operações baseadas em ID.
    - **Escalabilidade**: A melhoria é especialmente eficaz em sistemas com muitos processos, reduzindo o impacto de operações frequentes.
- **Desvantagens**:
    - Aumenta o uso de memória devido ao HashMap.
    - Requer manutenção adicional para sincronizar o HashMap com a fila circular (por exemplo, atualizar o mapa ao mover processos entre filas).

Essa melhoria preservaria a eficiência da fila circular para escalonamento, mas eliminaria o gargalo de busca, tornando o escalonador mais rápido e escalável.