Relatório de Análise e Justificativa de Design

1. Justificativa de Design

A estrutura escolhida para representar os processos foi a Lista Duplamente Encadeada, pois ela permite inserções e remoções eficientes em qualquer ponto da lista com complexidade O(1), desde que haja referência ao nó. Esse comportamento é adequado para um escalonador, que constantemente precisa mover processos entre diferentes filas (pronto, bloqueado, execução). Além disso, a lista duplamente encadeada facilita a implementação de políticas de escalonamento que exigem navegação tanto para frente quanto para trás.

A decisão por essa estrutura reflete uma análise cuidadosa das necessidades do escalonador. Em sistemas operacionais, como o modelo implementado na classe Scheduler, a gestão de processos exige flexibilidade para adicionar ou remover tarefas em tempo real, especialmente em um scheduler multilevel feedback queue (MLFQ). A Lista Duplamente Encadeada, implementada na classe ListaDuplamenteEncadeada, suporta essa flexibilidade ao manter ponteiros para os nós anteriores e posteriores, permitindo operações bidirecionais. Comparada a uma lista simplesmente encadeada, que só permite navegação unidirecional, ou a um array, que teria custo O(n) para inserções no meio, a estrutura escolhida otimiza o desempenho em cenários onde processos mudam de estado frequentemente (ex.: de "pronto" para "bloqueado" ao solicitar "DISCO"). No código, isso é evidente no método addFim e removeInicio, que garantem eficiência em filas separadas por prioridade (alta, média, baixa) e bloqueados. Essa escolha também alinha-se com práticas reais, como as runqueues do kernel Linux, que utilizam estruturas encadeadas para gerenciar processos, reforçando a relevância acadêmica e prática do design.

2. Complexidade (Big-O)

- **Inserção em uma lista**: O(1) quando feita no início ou fim.
- **Remoção de um processo específico**: O(1), caso se tenha o ponteiro para o nó; caso contrário, a busca é O(n).
- **Busca de processo por atributo (ex.: ID ou nome)**: O(n).
- **Troca de listas (ex.: pronto → bloqueado)**: O(1).

Portanto, o custo total de operações comuns do escalonador é eficiente, adequado para simulação de múltiplos processos.

Para uma análise mais detalhada, a complexidade reflete o comportamento assintótico em relação ao número de processos (n). A inserção em O(1) ocorre porque o método addFim apenas ajusta ponteiros, sem necessidade de realocação ou busca, como seria em um array dinâmico. A remoção em O(1) depende de acesso direto ao nó, como no removeInicio, mas uma busca por ID ou nome exige percorrer a lista linearmente, resultando em O(n). Esse trade-off é aceitável em um escalonador onde a maioria das operações é direta (ex.: mover processos entre filas), mas pode se tornar limitante em cenários com muitas buscas. A troca entre listas, como de "pronto" para "bloqueado" no método executarCiclo, é O(1) devido à independência das filas, refletindo um design modular. Em simulações com n pequeno (ex.: < 100 processos), o impacto do O(n) é mínimo, mas em sistemas maiores, como servidores com milhares de tarefas, alternativas como árvores de busca poderiam ser consideradas. Essa análise é fundamentada no código, onde a classe ListaDuplamenteEncadeada demonstra essas propriedades, garantindo um equilíbrio entre eficiência e simplicidade para o contexto acadêmico.

3. Análise da Anti-Inanição

A lógica do escalonador garante justiça ao evitar que processos de menor prioridade nunca sejam executados. Isso pode ser feito através de mecanismos como envelhecimento (aging), onde a prioridade de processos que estão esperando aumenta gradualmente, garantindo que eventualmente eles serão escolhidos. Sem essa regra, haveria o risco de inanição: processos de baixa prioridade poderiam ficar indefinidamente sem CPU.

A anti-inanição é um princípio essencial em escalonamento, especialmente em sistemas com múltiplas prioridades. No código, o mecanismo é parcialmente implementado pelo contador contAltaSeguidas, que limita execuções consecutivas de alta prioridade a 5 ciclos, forçando a alternância para filas de média ou baixa prioridade. Isso simula um aging básico, onde processos de baixa prioridade ganham "tempo de vida" relativo ao tempo de espera. Em termos teóricos, poderíamos definir uma função de envelhecimento como prioridade_nova = prioridade_original + (tempo_espera / constante_aging), promovendo processos de forma dinâmica. Sem essa intervenção, um influxo constante de tarefas de alta prioridade poderia levar à inanição, um problema clássico em algoritmos como Priority Scheduling estrito. Comparado a métodos como Round-Robin, que garantem execução por turnos, o design atual prioriza eficiência, mas depende do aging para justiça. Em cenários reais, como no Linux CFS, ajustes dinâmicos de prioridade são comuns, e essa análise sugere que uma extensão com timers por processo poderia aprimorar a robustez, alinhando-se aos objetivos do trabalho.

4. Análise do Bloqueio

Quando um processo solicita acesso ao DISCO, ele é movido para a lista de bloqueados. O ciclo de vida ocorre assim:

1. Pronto → Execução: processo ganha a CPU.
2. Execução → Bloqueado: solicita recurso de E/S (disco).
3. Bloqueado → Pronto: ao término da operação de disco, o processo retorna à fila de prontos.

Esse ciclo mostra a importância da estrutura de listas independentes (pronto, bloqueado), que organizam de forma clara o estado de cada processo.

O ciclo de bloqueio reflete o manejo de processos I/O-bound, comuns em sistemas operacionais. No método executarCiclo da classe Scheduler, a transição para "bloqueado" ocorre quando precisaDisco() retorna verdadeiro e jaBloqueou é falso, com o processo sendo adicionado à fila bloqueados via addFim. O desbloqueio, simulado por desbloqueiaUmSeTiver, move o processo de volta à fila original (baseada em prioridade), modelando a conclusão assíncrona de E/S. Essa separação em listas independentes, como visto nas variáveis alta, media, baixa e bloqueados, facilita o rastreamento de estados, evitando confusão em uma única estrutura. Em termos práticos, isso reflete sistemas como o Windows NT, que usa filas por estado. Para simulações mais realistas, poderíamos adicionar tempos de bloqueio variáveis (ex.: baseado em tamanho de dados lidos/escritos), mas a abordagem atual é didática, destacando transições de estado (finite state machine) e a eficiência de O(1) nas movimentações, como visto no código.

5. Ponto Fraco

O principal gargalo do scheduler atual está na busca linear em listas, que tem complexidade O(n). Isso pode se tornar um problema em cenários com muitos processos, aumentando o tempo de decisão do escalonador. Melhoria teórica: substituir a lista por uma fila de prioridade (Priority Queue baseada em heap), o que permitiria selecionar o processo de maior prioridade em O(log n).

A busca linear O(n) surge quando é necessário localizar um processo por ID ou nome, como em cenários onde o escalonador precisa verificar atributos específicos. Com o aumento de n (número de processos), esse custo pode degradar o desempenho, especialmente em sistemas de alta carga, como servidores web. A proposta de uma Priority Queue baseada em heap (ex.: Binary Heap) reduziria a seleção do próximo processo para O(log n), mantendo inserções em O(log n) ou O(1) em heaps otimizados (ex.: Fibonacci Heap). No Java, isso poderia ser implementado com PriorityQueue, ajustando comparadores por prioridade e tempo de espera para anti-inanição. Outra limitação é a ausência de preemptividade total: o scheduler é non-preemptive por ciclo, o que poderia ser melhorado com um quantum de tempo, interrompendo processos longos. Uma extensão teórica seria integrar Round-Robin dentro de prioridades iguais, evitando monopolização. Essas melhorias, inspiradas em escalonadores como o Completely Fair Scheduler (CFS) do Linux, elevaram a escalabilidade, sendo uma sugestão valiosa para o trabalho acadêmico.

6. Considerações Finais e Possíveis Extensões

Em resumo, o design adotado equilibra simplicidade e eficiência, servindo como base sólida para entender conceitos de escalonamento. Para extensões futuras, poderia-se incorporar suporte a múltiplos recursos de E/S (ex.: rede, impressora), modelando filas específicas por dispositivo, ou implementar métricas de desempenho como throughput, turnaround time e waiting time para avaliações quantitativas. Testes com arquivos de input variados (ex.: misturando I/O-bound e CPU-bound) validariam a robustez, e integrações com threads reais em Java simulariam concorrência paralela. Essa análise reforça a importância de estruturas

de dados adequadas em sistemas operacionais, destacando trade-offs entre complexidade e performance, e oferece um caminho para aprofundamento no contexto da disciplina.