Relatório de Análise e Justificativa de Design

relatorio_analise.pdf

1. Justificativa de Design

A implementação utiliza listas encadeadas e filas por prioridade para modelar o escalonador. Cada fila (alta, média, baixa e bloqueados) é representada por uma ListaDeProcessos simples, implementada como lista simplesmente encadeada com operações addFirst/addLast/removeFirst em O(1). Essa estrutura é eficiente para o comportamento do escalonador porque permite inserir e remover processos rapidamente nas extremidades (filas FIFO) sem custo de deslocamento de elementos — o acesso sequencial é natural ao modelo de filas de pronto e de bloqueio em sistemas operacionais. Além disso a separação por filas de prioridade reflete diretamente a política de escalonamento por prioridade, mantendo a lógica clara e de fácil verificação.

2. Complexidade (Big-O) das operações

Operações principais e suas complexidades na implementação atual: - adicionarProcesso: O(1) — adiciona no final da fila da prioridade apropriada (addLast). - remover próximo para executar (removeFirst): O(1) — retirar do início da fila. - refileirarNoFimDaSuaPrioridade: O(1) — addLast. - procura por id (findById): O(n) no tamanho da lista (linear). - removeById: O(n) no pior caso. - executarCicloDeCPU (por ciclo): as operações dominantes são O(1) cada (remoção, inserção, teste de recurso), logo um ciclo tem custo amortizado O(1) além de eventuais varreduras adicionais se chamadas; a exibição/serialização (toString) faz iteração e é O(m) onde m é o número de processos presentes no sistema no momento. Portanto as operações de escalonamento crítico (enfileirar/desenfileirar) são O(1), adequado para alto desempenho.

3. Análise da Anti-Inanição

A anti-inanição implementada controla um contador contador_ciclos_alta_prioridade que incrementa sempre que um processo de prioridade alta é executado e é zerado quando um processo de prioridade média ou baixa é escolhido. Se o contador atingir 5, o scheduler força a escolha de processos das prioridades inferiores (média, depois baixa) antes de escolher alta novamente — garantindo que processos de menor prioridade não fiquem indefinidamente sem CPU. Sem essa regra, cargas contínuas de processos de alta prioridade poderiam causar inanicação (starvation) de processos de prioridades inferiores: eles nunca seriam escolhidos e poderiam esperar indefinidamente. O risco inclui degradação de serviços, violação de SLAs e comportamento não justo da máquina.

4. Análise do Bloqueio (ciclo de vida de um processo que precisa do DISCO)

Exemplo: o processo P que necessita de 'DISCO' entra inicialmente na sua fila de prioridade (alta/média/baixa). Quando P atinge o início da fila e é selecionado para execução, o scheduler verifica precisaDeDiscoPelaPrimeiraVez. Se for a primeira solicitação, P tem seu flag jaSolicitouRecurso definido para true e é movido para lista_bloqueados (addLast). Ele permanece em bloqueados até o início de um ciclo subsequente, quando a rotina de desbloqueio retira um processo da lista_bloqueados com removeFirst e o refileira no fim da fila correspondente à sua prioridade. Assim o ciclo (simplificado) é: PRONTO → EXECUTANDO → BLOQUEADO (se pedir DISCO na 1ª vez) → (quando desbloqueado) PRONTO (reem colocado no fim) → EXECUTANDO → TERMINA. Observação: a política de desbloqueio atual retira apenas 1 processo bloqueado por ciclo (FIFO na lista_bloqueados), o que modela latência de I/O mas também causa potencial acúmulo se muitos processos demandarem DISCO simultaneamente.

5. Ponto Fraco e Melhoria Proposta

Principal gargalo: a lista_bloqueados é processada um elemento por ciclo (removeFirst no início do ciclo), o que pode criar um acúmulo quando muitos processos solicitam o mesmo recurso (DISCO). Além disso, a estratégia atual é single-threaded e não modela a concorrência real de I/O ou paralelismo de dispositivos. Melhoria teórica: usar uma estrutura que rastreie tempos de término de I/O (por exemplo, uma fila de eventos baseada em timestamp ou um min-heap de eventos) para desbloquear processos apenas quando o evento de I/O terminar — permitindo modelar latências variadas e desbloquear múltiplos processos conforme seus tempos expirarem. Outra melhoria é permitir

desbloqueio de múltiplos processos por ciclo ou simular um dispositivo com paralelismo (N acessos concorrentes), reduzindo gargalo. Para desempenho, também se poderia manter um mapa auxiliar (HashMap id→Node) quando remoções arbitrárias por id forem frequentes, reduzindo removeById para O(1) no custo de memória adicional.

6. Regras de Colaboração, Entrega e Apresentação (resumo requerido)

6.1 Formação dos Grupos e Linguagem: - Grupos até 3 pessoas. Qualquer linguagem que respeite as regras da disciplina. 6.2 Uso Obrigatório do Git e Repositório Público: - Código versionado em repositório Git público. - Histórico de commits individual é componente da nota; commits devem ser frequentes e atômicos. 6.3 Documentação (README e Código): - README.md profissional com: nome da disciplina/professor, nomes e matrículas dos integrantes, link do repositório, descrição do projeto e instruções claras de compilação/execução. - Código bem comentado e organizado.

7. Observações Finais

O design proposto privilegia clareza e correspondência direta com as políticas de escalonamento descritas no enunciado. Para cenários reais de sistemas operacionais, adicionar simulação de tempo, modelagem de latência de I/O e estruturas de eventos traria mais fidelidade. A implementação em Java, com listas encadeadas simples, é adequada para fins educacionais e para demonstrar conceitos de escalonamento, anti-inanição e bloqueio.

Apêndice: Observação sobre o código-fonte

O PDF contém uma descrição resumida do código. Incluí aqui declarações e resumo dos arquivos. Para o código-fonte completo, organizado e com histórico de commits, publique o repositório Git público conforme as regras do trabalho e forneça o link no README.