Relatório de Análise e Justificativa de Design

Professor: Dimmy Magalhães

Instituição: ICEV

Turma: Allen- 2° Periodo

Alunos:
Murilo Souza de Andrade

Erick Daniel Pereira Brito

Guilherme De Morais Cunha Rodrigues

Relatório de Análise e Justificativa de Design

Scheduler (iCEVOS)

Este relatório descreve e justifica o design da estrutura de dados utilizada no scheduler fornecido, analisa complexidades (Big-O) das operações implementadas, explica a estratégia de anti-inanição (fairness), descreve o ciclo de vida de um processo que precisa do recurso "DISCO", identifica o principal ponto fraco de performance e propõe uma melhoria teórica. Ao final, incluímos o código-fonte das classes entregues.

1) Justificativa de Design

A estrutura de dados utilizada no scheduler foi projetada para refletir as listas clássicas de um escalonador: filas (listas encadeadas) para processos prontos, bloqueados e/ou em execução, além de nós que representam cada processo. Usar listas encadeadas é eficiente para este contexto pelos motivos a seguir:

- Inserções e remoções frequentes: escalonadores movem processos entre filas (pronto, bloqueado, esperando I/O). Em listas encadeadas, inserir ou remover um nó quando já se tem a referência é O(1), sem necessidade de realocar memória como em arrays.

- Ordenação por prioridade/quantum: se o scheduler implementar filas por prioridade, combina-se listas

separadas por nível de prioridade ou se mantêm filas por classes.

- Simplicidade sem overhead de estruturas complexas: para um trabalho acadêmico a implementação com listas deixa o comportamento do escalonador explícito e fácil de inspeção, depuração e extensão.

2) Complexidade (Big-O) das operações

A análise abaixo considera a implementação típica com listas encadeadas simples e as operações observáveis no código fornecido:

- Inserir processo na fila de prontos: O(1) se inserido no fim ou início com ponteiros mantidos; O(n) se for necessário buscar uma posição por prioridade sem estruturas auxiliares.

- Remover processo da cabeça (escalonar): O(1).

- Mover processo entre listas (pronto → bloqueado → pronto): O(1) dado o nó.

- Buscar processo por PID (se houver): O(n) na pior caso, pois precisa percorrer a lista.

- Atualizar tempo/quantum a cada tick: O(1) por processo (total O(p) por tick, onde p é número de processos ativos).

3) Análise da Anti-Inanição (Fairness)

A anti-inanição visa garantir que nenhum processo seja permanentemente preterido. Uma estratégia comum, que pode ser identificada no design, inclui:

- Envelhecimento (aging): aumentar a prioridade de processos que esperam há muito tempo, movendo-os progressivamente para filas de maior prioridade.

- Quantum justo: garantir que cada processo receba fatias de CPU regulares.

Como a implementação usa filas encadeadas, o envelhecimento pode ser implementado através de contadores de espera em cada processo; ao ultrapassar um limiar, o processo é promovido para uma fila de maior prioridade ou colocado à frente da fila atual.

Risco se essa regra não existisse:

- Processos de baixa prioridade poderiam nunca ser escalonados se existirem processos de alta prioridade que chegam continuamente (inanição/starvation).

- Isso causa má utilização de recursos e viola requisitos de justiça do sistema.

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

1. Criação: o processo é criado e inserido na fila de prontos.

2. Execução: o processo é escalonado e recebe CPU; ao necessitar acessar o DISCO, ele faz uma requisição de I/O e é movido para a lista de bloqueados (ou espera por I/O).

3. Espera por I/O: na lista de bloqueados, o processo aguarda o término da operação de disco. O sistema de I/O ou uma interrupção notifica o scheduler quando o I/O termina.

4. Desbloqueio: ao receber o sinal de completude do DISCO, o processo é movido de volta para a fila de prontos (normalmente no final da fila).

5. Re-escalonamento: o processo volta a concorrer por CPU e continuará sua execução a partir do ponto em que bloqueou. No código, isso corresponde a removê-lo da lista de prontos e inseri-lo na lista de bloqueados (e vice-versa) — operações que, com listas encadeadas, são O(1) se a referência ao nó for conhecida.

5) Ponto Fraco (gargalo) e melhoria proposta

Principal gargalo:

- o código possui como maior defeito principalmente uma deficiência quanto a exibição feita sem pausa dos processos o que torna poluído no terminal e mais dificil de compreender e tambem o scheduler precisa frequentemente

localizar processos arbitrários, percorrer listas encadeadas torna-se O(n) por operação, degradando performance com muitos processos.

Melhoria teórica proposta:

Utilizar uma função que executa uma pausa para a verificação mais facilitada e para a melhora de performance o ideal seria utilizar as bibliotecas do java como ArrayList, LinkedList, Queue e entre outras estruturas de dados

Conclusão

A implementação baseada em listas encadeadas é adequada e eficiente para operações típicas de escalonadores (inserção e remoção frequentes), mantendo simplicidade e claridade. Para cenários com muitos processos e necessidades de buscas arbitrárias ou prioridades dinâmicas, recomenda-se híbridos (listas + hashmap + estruturas de prioridade) para melhorar complexidade e desempenho.