Relatório de Análise - Escalonador de Processos

Disciplina: Estrutura de Dados
Sistema Operacional: iCEVOS
Componente: Escalonador de Processos

Grupo: Adriano Batista, Eduardo Oliveira, João Marcos Nogueira

Turma Allen

1. Justificativa de Design

Estrutura de Dados Escolhida: Lista Circular Ligada

A **Lista Circular Ligada** foi escolhida como estrutura base para implementar as filas de processos do escalonador iCEVOS pelas seguintes razões técnicas:

1.1 Eficiência nas Operações Principais

- **Inserção no final**: O(1) - Mantemos uma referência à cauda, permitindo inserção direta.
- **Remoção do início**: O(1) - A cabeça sempre aponta para o próximo processo a ser executado.
- **Rotação natural**: A natureza circular permite que processos retornem automaticamente ao final da fila

1.2 Adequação ao Modelo Round-Robin

O escalonador de processos implementa um sistema baseado em prioridades com características de Round-Robin dentro de cada nível. A lista circular é perfeita para este modelo porque:

- **Rotação contínua**: Processos que não terminam retornam naturalmente ao final da fila.
- **Acesso sequencial**: Sempre executamos o processo no início da fila.

- **Prevenção de inanição**: A estrutura facilita a implementação da regra de anti-inanição.

1.3 Gerenciamento de Memória

- **Alocação dinâmica**: Não precisamos definir tamanho máximo das filas.
- **Sem desperdício**: Aloca apenas o espaço necessário para os processos ativos.
- **Flexibilidade**: Suporta qualquer quantidade de processos por fila.

2. Análise de Complexidade (Big-O)

2.1 Operações da Lista Circular

```
// Complexidade das operações principais
public void adicionarNoFinal(Processo processo)     // O(1)
public Processo removerDoInicio()                   // O(1)
public boolean estaVazia()                          // O(1)
public Processo verInicio()                         // O(1)
public void imprimirLista()                         // O(n)
public int getTamanho()                             // O(1)
```

2.2 Operações do Escalonador

```
// Análise detalhada do ciclo de CPU
public void executarCicloDeCPU() {
    // Desbloqueio: O(1)
    desbloquearProcesso();

    // Seleção: O(1)
    Processo processo = selecionarProximoProcesso();

    // Execução: O(1)
    executarProcesso(processo);

    // Total por ciclo: O(1)
}
```

2.3 Complexidade Geral do Sistema

- **Por ciclo de CPU**: O(1) - Todas as operações são de tempo constante.

- **Para n processos totais**: O(n) - Cada processo deve ser executado até completar.
- **Para p processos simultâneos**: O(1) - A quantidade de processos nas filas não afeta a complexidade por ciclo.
- **Espaço**: O(p) - Armazena apenas os processos ativos nas filas.

2.4 Análise de Cenários

Melhor Caso: Todos os processos têm 1 ciclo → O(n)
Caso Médio: Processos com k ciclos médios → O(n × k)
Pior Caso: Processo com muitos ciclos + bloqueios → O(n × k × b)

- n = número de processos
- k = ciclos médios por processo
- b = fator de bloqueio

3. Análise da Anti-Inanição

3.1 Problema da Inanição

Em sistemas de prioridade pura, processos de **baixa prioridade** podem sofrer **inanição** quando:

- Há fluxo constante de processos de alta prioridade
- Processos importantes nunca conseguem executar
- Sistema torna-se injusto e pode travar processos críticos

3.2 Solução Implementada

```
// Regra de anti-inanição implementada
if (contadorCiclosAltaPrioridade >= 5) {
    // Força execução de processo de menor prioridade
    if (!listaMediaPrioridade.estaVazia()) {
        return listaMediaPrioridade.removerDoInicio();
    } else if (!listaBaixaPrioridade.estaVazia()) {
        return listaBaixaPrioridade.removerDoInicio();
    }
}
```

3.3 Garantias de Justiça

Tempo Máximo de Espera

- **Processo de média prioridade**: Máximo 5 ciclos de espera
- **Processo de baixa prioridade**: Máximo 5 ciclos + tempo da fila média
- **Starvation completa**: Impossível com esta implementação

Análise Matemática

Para um processo de baixa prioridade P:

- **Pior caso**: 5 processos alta + todos os processos média
- **Tempo máximo**: 5 + |FilaMedia| ciclos
- **Garantia**: P sempre executará eventualmente

4. Análise do Bloqueio de Recursos

4.1 Ciclo de Vida - Processo com Recurso "DISCO"

```
graph TD
    A[Processo Criado] --> B[Fila de Prioridade]
    B --> C[Selecionado para Execução]
    C --> D{Precisa de DISCO?}
    D -->|Sim| E[Bloqueado - Lista de Bloqueados]
    D -->|Não| F[Execução Normal]
    F --> G{Terminou?}
    G -->|Não| B
    G -->|Sim| H[Processo Finalizado]
    E --> I[Próximo Ciclo: Desbloqueio]
    I --> B
```

4.2 Estados Detalhados

Estado 1: PRONTO

- Processo está na fila de sua prioridade
- Aguarda sua vez de executar
- **Transição**: Quando selecionado pelo escalonador

Estado 2: EXECUTANDO

- Processo está usando a CPU
- **Se precisa DISCO**: vai para BLOQUEADO
- **Se não precisa**: executa e volta para PRONTO ou FINALIZADO

Estado 3: BLOQUEADO

- Processo solicitou recurso DISCO
- Movido para `listaBloqueados`
- **FIFO**: Primeiro bloqueado é primeiro desbloqueado

Estado 4: DESBLOQUEADO

- A cada ciclo, um processo é desbloqueado
- Retorna para sua fila de prioridade original
- **Importante**: Não precisa mais do recurso DISCO

4.3 Política de Desbloqueio

- **FIFO (First In, First Out)**: Processo mais antigo é desbloqueado primeiro, ou seja o primeiro que entra na fila é o primeiro que sai.
- **Um por ciclo**: Apenas um processo é desbloqueado por ciclo
- **Justiça**: Evita que processos fiquem bloqueados indefinidamente, e por tanto não correm o risco de nunca serem executados.

5. Análise de Performance - Ponto Fraco

5.1 Principal Gargalo Identificado

Problema: Desbloqueio Sequencial

O maior gargalo do sistema atual é o **desbloqueio de apenas um processo por ciclo**:

```
// Gargalo atual
if (!listaBloqueados.estaVazia()) {
    Processo processoDesbloqueado =
listaBloqueados.removerDoInicio(); // Apenas 1
    adicionarProcesso(processoDesbloqueado);
}
```

exemplo:

- 50 processos solicitam DISCO simultaneamente
- Todos ficam bloqueados
- Demora 50 ciclos para desbloqueá-los completamente
- **Throughput reduzido drasticamente**

5.2 Análise de Impacto

Métricas de Performance Afetadas:

- **Throughput**: Redução de ~30-60% com muitos bloqueios
- **Tempo de resposta**: Aumento linear com número de bloqueados
- **Utilização de CPU**: Desperdício de ciclos com filas vazias

5.3 Proposta de Melhoria Teórica

Solução: Desbloqueio Inteligente por Lotes

```
// Melhoria proposta
public void desbloquearProcessosInteligente() {
    int maxDesbloqueios = Math.min(3, listaBloqueados.getTamanho());
    int processosAltaEsperando = listaAltaPrioridade.getTamanho();

    // Se há poucos processos de alta prioridade, desbloqueia mais
    if (processosAltaEsperando < 2) {
        maxDesbloqueios = Math.min(5, listaBloqueados.getTamanho());
    }

    for (int i = 0; i < maxDesbloqueios; i++) {
        if (!listaBloqueados.estaVazia()) {
            Processo p = listaBloqueados.removerDoInicio();
            adicionarProcesso(p);
        }
    }
}
```

Benefícios Esperados:

- **Throughput**: Aumento de 40-80% em cenários com bloqueios
- **Tempo de resposta**: Redução proporcional ao lote de desbloqueio

- **Utilização**: Melhor aproveitamento da CPU
- **Adaptabilidade**: Sistema se adapta à carga de trabalho

5.4 Trade-offs da Solução Proposta

Vantagens:

- Performance significativamente melhor
- Sistema mais responsivo
- Melhor utilização de recursos

Desvantagens:

- Maior complexidade de implementação
- Possível favorecimento excessivo de alta prioridade
- Necessidade de balanceamento fino dos parâmetros