Relatório de Análise e Justificativa de Design

Escalonador de Processos iCEVOS

Disciplina: Algoritmos e Estruturas de Dados
Professor: Prof. Dimmy Magalhães
Universidade: ICEV

Integrantes:

- David Carvalho - Matrícula: [0030544]
- Enzo Andrade - Matrícula: [0030932]

Data: Dezembro de 2024

---

1. Justificativa de Design

1.1 Escolha da Lista Ligada Simples

A escolha da **lista ligada simples** como estrutura de dados principal para implementar o comportamento do escalonador se justifica pelos seguintes aspectos:

Eficiência das Operações Críticas

- **Inserção no final (O(1))**: Essencial para reinserir processos após execução
- **Remoção do início (O(1))**: Crítica para seleção do próximo processo a executar
- **Acesso ao primeiro elemento (O(1))**: Necessário para verificar qual processo será executado

Adequação ao Comportamento FIFO por Prioridade

O escalonador implementa uma variação de **prioridade preemptiva com round-robin interno**:

- Dentro de cada nível de prioridade, os processos são executados em ordem FIFO
- A lista ligada preserva naturalmente a ordem de chegada
- Processos reexecutados vão para o final da fila, mantendo justiça

Simplicidade de Implementação

- Estrutura direta para o problema proposto

- Não requer complexidade adicional de árvores ou heaps
- Facilita debugging e visualização do estado do sistema

Gestão Dinâmica de Memória

- Não há desperdício de espaço (diferente de arrays fixos)
- Cresce e diminui conforme necessário
- Adequada para sistema com número variável de processos

1.2 Separação em Múltiplas Listas

A decisão de usar **quatro listas separadas** (alta, média, baixa prioridade e bloqueados) oferece:

- **Clareza conceptual**: Cada lista tem responsabilidade específica
- **Eficiência na seleção**: Verificação direta por ordem de prioridade
- **Facilidade de manutenção**: Estado visível e depurável

---

2. Análise de Complexidade (Big-O)

2.1 Operações da Lista Ligada

Operação	Complexidade	Justificativa
`adicionar(Processo p)`	**O(1)**	Inserção direta no final com referência `ultimo`
`removerPrimeiro()`	**O(1)**	Remoção direta do início com atualização de `primeiro`
`isEmpty()`	**O(1)**	Verificação simples de `primeiro == null`
`size()`	**O(1)**	Retorno de contador mantido incrementalmente
`peek()`	**O(1)**	Acesso direto ao primeiro elemento

2.2 Operações do Escalonador

Operação	Complexidade	Detalhamento
`adicionarProcesso()`	**O(1)**	Switch + inserção O(1) na lista apropriada
`escolherProximoProcesso()`	**O(1)**	Verificação sequencial de 3 listas em ordem fixa

`executarProcesso()`	**O(1)**	Operações aritméticas e inserção O(1)
`desbloquearProcesso()`	**O(1)**	Remoção O(1) + inserção O(1)
`bloquearProcesso()`	**O(1)**	Modificação de estado + inserção O(1)
`executarCiclo()`	**O(1)**	Composição de operações O(1)

2.3 Operações de Sistema

Operação	Complexidade	Observação
`carregarProcessos()`	**O(n)**	n = número de linhas no arquivo
`executarCompleto()`	**O(m)**	m = número total de ciclos executados
Sistema completo	**O(n + m)**	Linear no tamanho da entrada e execução

2.4 Análise de Espaço

- **Espaço por processo**: O(1) - armazenamento direto em nó
- **Espaço total das listas**: O(n) - proporcional ao número de processos
- **Espaço auxiliar**: O(1) - apenas variáveis de controle
- **Espaço total**: **O(n)** onde n é o número de processos

---

3. Análise da Anti-Inanição

3.1 Mecanismo Implementado

O sistema implementa **prevenção de inanição por contagem de ciclos**:

```
if (contadorCiclosAltaPrioridade >= limiteAltaPrioridade) {
    // Força execução de média ou baixa prioridade
    contadorCiclosAltaPrioridade = 0; // Reset após execução
}
```

3.2 Como Garante Justiça

Limite Determinístico

- Após exatamente **5 execuções consecutivas** de alta prioridade
- Sistema **obrigatoriamente** busca processo de menor prioridade
- Garante que processos de baixa prioridade executem periodicamente

Ordem de Busca Justa

1. Tenta **média prioridade** primeiro
2. Se não há média, tenta **baixa prioridade**
3. Apenas se não há nenhuma das duas, continua com alta

Reset Adequado

- Contador é **zerado** após qualquer execução de não-alta prioridade
- Permite que alta prioridade volte a dominar por mais 5 ciclos
- Balanceia responsividade com justiça

3.3 Riscos sem Anti-Inanição

Inanição Completa (Starvation)

- Processos de baixa prioridade **nunca** executariam
- Sistema seria injusto para tarefas menos críticas
- Possível violação de deadlines de processos de background

Degradação de Performance do Sistema

- Acúmulo indefinido de processos de baixa prioridade
- Consumo crescente de memória sem liberação
- Sistema eventualmente ficaria sobrecarregado

Comportamento Imprevisível

- Tempos de resposta não determinísticos para processos não-críticos
- Dificuldade de planejamento de capacidade do sistema
- Usuário perderia confiança na responsividade

---

4. Análise do Bloqueio

4.1 Ciclo de Vida - Processo com Recurso DISCO

Vamos acompanhar um processo exemplo:
`P2(EditorTexto,pri:2,ciclos:4,DISCO)`

Fase 1: Carregamento
Estado: Processo criado e adicionado à lista_media_prioridade
Lista Média: [P2(EditorTexto,pri:2,ciclos:4)]
Lista Bloqueados: []

Fase 2: Primeira Seleção

Ciclo X: Scheduler seleciona P2 para execução
Verificação: precisaDisco() retorna TRUE
- recursoNecessario = "DISCO" ✓
- !jaPediuRecurso = true ✓
Ação: Processo é BLOQUEADO (não executa)

Fase 3: Bloqueio

Estado após bloqueio:
Lista Média: [] (P2 removido)
Lista Bloqueados: [P2(EditorTexto,pri:2,ciclos:4)] (P2 adicionado)
P2.jaPediuRecurso = true (flag marcada)

Fase 4: Aguardo

Ciclos Y, Y+1, Y+2...: P2 permanece na lista de bloqueados
Outros processos executam normalmente
A cada ciclo, P2 continua na fila de bloqueados

Fase 5: Desbloqueio

Ciclo Z: No início do ciclo
Ação: desbloquearProcesso() remove P2 do início de lista_bloqueados
Estado: P2 retorna ao FINAL de lista_media_prioridade
Lista Média: [...outros processos..., P2(EditorTexto,pri:2,ciclos:4)]

Fase 6: Execução Normal

Ciclo Z+K: P2 é selecionado novamente
Verificação: precisaDisco() retorna FALSE
- jaPediuRecurso = true ✗ (não bloqueia mais)
Ação: P2 executa normalmente, ciclos: 4→3

Fase 7: Reexecuções

P2 continua executando normalmente até ciclos chegarem a 0
Cada execução: volta para o final da lista_media_prioridade
Nunca mais é bloqueado (jaPediuRecurso permanece true)

4.2 Propriedades do Sistema de Bloqueio

Bloqueio Único

- Processo é bloqueado **apenas na primeira** solicitação

- Flag `jaPediuRecurso` impede bloqueios subsequentes
- Simula "alocação de recurso" permanente

FIFO para Desbloqueio

- Processos são desbloqueados em ordem de chegada à fila
- Evita favoritism entre processos bloqueados
- Garante justiça temporal no acesso ao recurso

Retorno à Lista Original

- Processo desbloqueado volta à mesma lista de prioridade
- Preserva características de prioridade originais
- Mantém coerência do sistema de prioridades

---

5. Ponto Fraco e Proposta de Melhoria

5.1 Principal Gargalo de Performance

O principal gargalo identificado é a **ausência de otimização na busca por processos prontos**:

Problema Atual

```
public boolean temProcessosProntos() {
    return !listaAltaPrioridade.isEmpty() ||
           !listaMediaPrioridade.isEmpty() ||
           !listaBaixaPrioridade.isEmpty();
}
```

- **Custo**: 3 verificações O(1) a cada ciclo
- **Frequência**: Chamado em todo `executarCiclo()`
- **Ineficiência**: Verificação redundante quando listas estão constantemente vazias

5.2 Limitações Adicionais

Falta de Priorização Dinâmica

- Prioridades são fixas durante toda execução
- Não há envelhecimento (aging) de processos
- Processos não podem ter prioridade ajustada por comportamento

Recurso Único e Simplista

- Apenas um tipo de recurso ("DISCO") suportado
- Não há diferentes tipos de bloqueio

- Ausência de deadlock detection

Sem Métricas de Performance

- Não calcula tempo médio de espera
- Não monitora throughput do sistema
- Ausência de estatísticas de fairness

5.3 Proposta de Melhoria Teórica

Otimização 1: Cache de Estado

```
public class Scheduler {
   private boolean temProcessosCache = false;
   private int processosTotais = 0;

   private void invalidarCache() {
      // Recalcula apenas quando necessário
      temProcessosCache = (processosTotais > 0);
   }
}
```

Benefícios:

- Reduz verificações redundantes de O(3) para O(1)
- Atualização apenas quando listas mudam
- Melhoria especialmente em sistemas com muitos ciclos vazios

Otimização 2: Prioridade Dinâmica com Aging

```
public class ProcessoComAging extends Processo {
   private int tempoEspera = 0;
   private int prioridadeEfetiva;

   public void envelhecer() {
      tempoEspera++;
      if (tempoEspera > LIMITE_AGING) {
         prioridadeEfetiva = Math.max(1, prioridade - 1);
      }
   }
}
```

Benefícios:

- Reduz inanição de forma mais natural
- Processos antigos "sobem" de prioridade automaticamente
- Sistema mais responsivo para processos com longa espera

Otimização 3: Múltiplos Recursos

```
public class GerenciadorRecursos {
    private Map<String, FilaBloqueados> recursos;
    private Map<String, Integer> limitesRecursos;

    public boolean tentarAlocarRecurso(Processo p, String recurso) {
        return recursos.get(recurso).tentarAlocar(p);
    }
}
```

Benefícios:

- Suporte a CPU, DISCO, MEMORIA, REDE simultaneamente
- Controle de concorrência por tipo de recurso
- Detecção de deadlocks potenciais

5.4 Impacto das Melhorias

Otimização	Complexidade Atual	Complexidade Melhorada	Ganho
Cache de Estado	O(3) por ciclo	O(1) por ciclo	3x mais rápido
Aging	Sem aging	O(1) por processo/ciclo	Maior justiça
Múltiplos Recursos	1 recurso fixo	N recursos dinâmicos	Flexibilidade

---

6. Conclusões

6.1 Adequação da Solução

A implementação atual atende satisfatoriamente aos requisitos propostos:

- **Funcionalidade completa**: Todas as especificações foram implementadas
- **Eficiência adequada**: Operações críticas em O(1)
- **Robustez**: Tratamento de erros e casos extremos
- **Clareza**: Código legível e bem estruturado

6.2 Pontos Fortes

1. **Simplicidade elegante**: Solução direta sem complexidade desnecessária
2. **Corretude**: Algoritmo implementa fielmente as regras especificadas
3. **Eficiência**: Complexidade linear no melhor caso possível
4. **Manutenibilidade**: Estrutura clara facilita modificações futuras

6.3 Lições Aprendidas

- **Estruturas simples podem ser mais efetivas** que soluções complexas

- **Separação de responsabilidades** facilita debugging e manutenção
- **Prevenção de casos extremos** (anti-inanição) é essencial em sistemas reais
- **Tratamento robusto de entrada** é crucial para sistemas de produção

6.4 Aplicabilidade Prática

A solução desenvolvida demonstra princípios fundamentais de:

- **Sistemas Operacionais**: Escalonamento preemptivo com prioridades
- **Estruturas de Dados**: Uso eficiente de listas ligadas
- **Algoritmos**: Balance entre eficiência e justiça
- **Engenharia de Software**: Código limpo e bem documentado

O projeto serve como base sólida para compreensão de schedulers reais em sistemas operacionais modernos.
