Universidade de São Paulo
Escola de Artes, Ciências e Humanidades (EACH)

Sistemas Operacionais
Exercício-Programa 1: Escalonador Round-Robin

Dérick dos Santos Arriado – T94 – 15636042
Higor Gabriel de Freitas – T04 – 15575879
Hugo Cardoso Ferreira de Araújo – T04 – 15459500
João Henrique Kuroki Cominato – T94 – 15462870
Karina Yang Chen – T94 – 15466658

São Paulo
Entrega para 12 de Agosto de 2025

1 Introdução

Este trabalho tem como objetivo a implementação de um escalonador de processos para um sistema de time-sharing em uma máquina virtual com arquitetura simplificada, utilizando o algoritmo Round Robin. O sistema deve gerenciar a execução concorrente de múltiplos processos em um único processador, com controle de estados (Executando, Pronto, Bloqueado), gestão de trocas de contexto e tratamento de operações de entrada/saída, seguindo as especificações do enunciado.

Nnossa abordagem de desenvolvimento seguiu uma estrutura modular, organizando o sistema em componentes: BCP (Bloco de Controle de Processo) para o controle individual de processos (contexto), Tabela de Processos para o gerenciamento global, e módulos de IO para carregamento de programas e geração de *logs*.

Nosso foco será na implementação do algoritmo Round Robin propriamente dito, com controle de quantum, filas de prontos e bloqueados, e mecanismos de preempção e bloqueio por E/S, usando-se um tempo de espera igual a *dois quantum*.

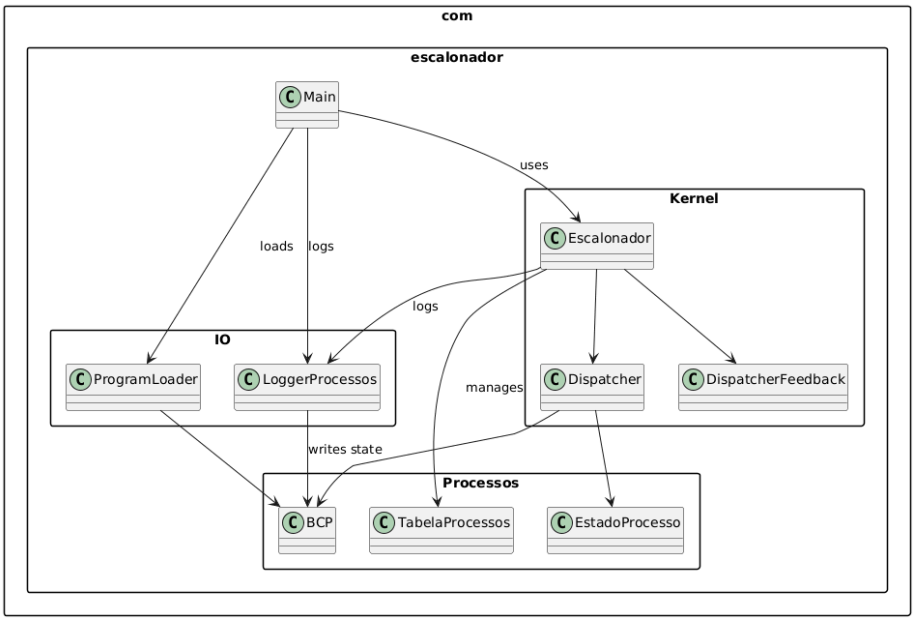

Figura 1: Diagrama da estrutura de pacotes do projeto.

1.1 Implementação

O projeto foi implementando na linguagem de programação Java e requer o JDK superior a versão 24 para ser rodado. Além do código, foi colocado um ".jar‘ para execução facilitada do programa. É necessário que, para rodar o programa, a pasta "programas‘ esteja no mesmo diretório que o arquivo ".jar".

Os programas devem estar dispostos na pasta programas, porém somente é possível rodar 10 por vez por padrão, e para aumentar esse limite deve ser alterado a constante **NUMERO_DE_PROGRAMAS** dentro do código da Main().

O sistema foi estruturado seguindo princípios de modularização e organização em pacotes, dividindo as responsabilidades em componentes especializados. A arquitetura implementada compreende quatro pacotes principais: Processos para o gerenciamento de processos, *Kernel* para o núcleo do escalonador, IO para operações de entrada e saída, e o pacote principal que coordena a execução.

```
public enum EstadoProcesso {
    EXECUTANDO,
    BLOQUEADO,
    PRONTO
}
```

No pacote Processos, foram implementadas as estruturas fundamentais para o controle de execução. A classe BCP (Bloco de Controle de Processo) constitui o elemento central, armazenando todas as informações necessárias para o gerenciamento de processos, incluindo contador de programa, estado atual (definido pelo *enum* EstadoProcesso com valores ***EXECUTANDO, BLOQUEADO e PRONTO***), valores dos registradores X e Y em BCP, o código do programa em formato de lista de strings, e o nome do programa. A classe incorpora validações robustas para transições de estado, prevenindo mudanças inválidas como a transição direta de BLOQUEADO para EXECUTANDO.

```
public class BCP {
    private int contadorDePrograma;
    private EstadoProcesso estadoProcesso;
    private int registradorX;
    private int registradorY;
    private final ArrayList<String> codigo;
    private final String nomePrograma;
    private int tempoEspera;
    ...
}
```

A TabelaProcessos atua como repositório para todos os BCPs ativos, embora sua implementação seja básica, funcionando principalmente como um contêiner de processos.

```
public class TabelaProcessos {
    private Queue<BCP> processos_prontos = new LinkedList<>();
    private Queue<BCP> processos_bloqueados = new LinkedList<>();
    private List<BCP> processos = new ArrayList<>();
```

```
5
6       public void adicionaProcessos(BCP bcp) {
7           if (bcp != null) {
8               processos.add(bcp);
9           }
10      }
11
12      public void excluiProcessos(BCP bcp) {
13          if (bcp != null) {
14              processos.remove(bcp);
15          }
16      }
17  }
```

O pacote IO contém componentes completos para operações de entrada e saída. A classe *ProgramLoader* é responsável pelo carregamento de programas a partir de arquivos texto, lendo o nome do programa na primeira linha e as instruções subsequentes, criando instâncias de BCP adequadamente inicializadas. Também gerencia o carregamento do valor do *quantum* do arquivo *quantum.txt*.

A classe *LoggerProcessos* implementa todo o sistema de *logging* exigido, gerando arquivos de saída com formatação precisa conforme especificado, incluindo registros de carregamento, execução, interrupção, operações de E/S, término de processos e estatísticas finais do sistema.

```
1   public class ProgramLoader {
2       public BCP carregarPrograma(String pathPrograma) {
3           ArrayList<String> codigo = new ArrayList<>();
4           try {
5               BufferedReader br = new BufferedReader(new FileReader(
                    pathPrograma));
6               String nomePrograma = br.readLine();
7               String linha;
8               while ((linha = br.readLine()) != null) {
9                   codigo.add(linha.trim());
10              }
11
12              return new BCP(codigo, nomePrograma);
13          }
14          ...
15  }
```

O pacote *Kernel*, contém as classes *Escalonador* e *Dispatcher*, é responsável pela lógica central do algoritmo de Round Robin, mecanismos de troca de contexto, controle de quantum e gerenciamento das filas de processos prontos e bloqueados. O Dispatcher atua como interpretador de instruções, processando comandos de atribuição, E/S e SAIDA, e sinaliza ao Escalonador as necessidades de mudança de estado, que são então gerenciadas pela TabelaProcessos.

```
1   public DispatcherFeedback Rodar(BCP processo) {
2       if (processo.getEstadoProcesso() != EstadoProcesso.EXECUTANDO)
            {
3           processo.setEstadoProcesso(EstadoProcesso.EXECUTANDO);
4       }
5
```

```
    String linhaDeCodigo = processo.getCodigo().get(processo.
        getContadorDePrograma());

    // Pegando o contexto (Simbolicamente, como se fosse o
        Dispatcher obtendo o contexto)
    int X = processo.getRegistradorX();
    int Y = processo.getRegistradorY();

    if (linhaDeCodigo.contains("X")) {
        String[] partes = linhaDeCodigo.split("=");
        int valorRegX = Integer.parseInt(partes[1].trim());
        processo.setRegistradorX(valorRegX);
        processo.incrementarContadorDePrograma();
        return DispatcherFeedback.NADA;
    }

    if (linhaDeCodigo.contains("Y")) {
        String[] partes = linhaDeCodigo.split("=");
        int valorRegY = Integer.parseInt(partes[1].trim());
        processo.setRegistradorY(valorRegY);
        processo.incrementarContadorDePrograma();
        return DispatcherFeedback.NADA;
    }

    if (linhaDeCodigo.contains("COM")) {
        processo.incrementarContadorDePrograma();
        return DispatcherFeedback.NADA;
    }
    ...
}
```

O Escalonador implementa um ciclo básico de execução por quantum, implementando todas as funcionalidades essenciais do algoritmo *Round Robin*, incluindo o tratamento correto de operações de *E/S* com tempo de bloqueio fixo, decremento automático de tempos de espera na fila de bloqueados, e transições de estado precisas entre executando, pronto e bloqueado.

```
while (!tabelaProcessos.getProcessos().isEmpty()) {
    diminuiTempoEspera();

    if(!tabelaProcessos.getProcessos_prontos().isEmpty()){
        BCP processo = tabelaProcessos.getProcessos_prontos().poll
            ();
        loggerProcessos.executaProcesso(processo);

        for (int i = 0; i < quantum; i++) {
            DispatcherFeedback SaidaProcesso = dispatcher.Rodar(
                processo);

            if (SaidaProcesso == DispatcherFeedback.ES) {
                tabelaProcessos.bloqueiaProcessos(processo,
                    TEMPO_BLOQUEIO);
                loggerProcessos.interrompeProcesso(processo, i+1);
                somaDeQuantumRodou += i+1;
                numeroDeTrocas++;
                break;
            }
```

```
18
19              if (SaidaProcesso == DispatcherFeedback.NADA) {
20                  if (i == quantum - 1) {
21                      tabelaProcessos.trocaExecucao(processo);
22                      loggerProcessos.interrompeProcesso(processo, i
                            + 1);
23                      somaDeQuantumRodou += i + 1;
24                      numeroDeTrocas++;
25                  }
26
27                  continue;
28              }
29
30              if (SaidaProcesso == DispatcherFeedback.FEITO) {
31                  tabelaProcessos.excluiProcessos(processo);
32                  somaDeQuantumRodou += i + 1;
33                  numeroDeTrocas++;
34
35                  loggerProcessos.terminaProcesso(processo);
36                  TempoFinalizacaoSoma += numeroDeTrocas;
37
38                  break;
39              }
40
41              numeroInteracoes++;
42          ...
```

2 Análise de Resultados

2.1 Metodologia de Teste

Para uma análise abrangente do sistema de escalonamento, foram conduzidos testes com diferentes valores de quantum, variando de 1 a 21 instruções, cobrindo tanto cenários com quanta curta (máxima preempção e overhead) quanto longos (mínimas trocas de contexto). Cada configuração foi executada múltiplas vezes para garantir a consistência dos resultados, utilizando os mesmos 10 programas de entrada para manter a comparabilidade.

3 Métricas de Avaliação

1. Número médio de trocas de processo por processo: representa a frequência média com que um processo é interrompido (por fim de quantum, E/S, ou término) e retornam à fila de prontos;

2. Número médio de instruções executadas por quantum: indica a eficiência na utilização do tempo de CPU alocado, sendo a média de instruções executadas até alguma troca de processo ocorrer.

Tabela 1: Relação entre Quantum, Média de Trocas, Média de Instruções e variações percentuais.

Quantum	Média de Trocas	Média de Instruções	Ganho % de MI	Perda % de MT
1	13,90	1,0000	NAN	NAN
2	7,90	1,7595	75,95%	-43,17%
3	5,90	2,3559	33,90%	-25,32%
4	5,20	2,6731	13,46%	-11,86%
5	4,40	3,1591	18,18%	-15,38%
6	4,10	3,3902	7,32%	-6,82%
7	3,80	3,6579	7,90%	-7,32%
8	3,80	3,6579	0,00%	0,00%
9	3,70	3,7568	2,70%	-2,63%
10	3,70	3,7568	0,00%	0,00%
11	3,50	3,9714	5,71%	-5,41%
12	3,50	3,9714	0,00%	0,00%
13	3,50	3,9714	0,00%	0,00%
14	3,40	4,0882	2,94%	-2,86%
15	3,40	4,0882	0,00%	0,00%
16	3,40	4,0882	0,00%	0,00%
17	3,40	4,0882	0,00%	0,00%
18	3,40	4,0882	0,00%	0,00%
19	3,40	4,0882	0,00%	0,00%
20	3,40	4,0882	0,00%	0,00%
21	3,30	4,2121	3,03%	-2,94%

3.1 Análise da tabela 1

Do quantum pequeno 1 a 4 temos uma melhoria acentuada, o sistema tem uma alta média de trocas, e ao aumentar do quantum 1 para o 2 temos um salto de 75,95% um salto enorme na eficiência, depois tem uma perda de aproximadamente 43% de eficiência. Isso acontece pois com o quantum maior os processos conseguem executar mais vezes antes de serem interrompidos. A partir do quantum com valor 5, o ganho de MI (Média de Instruções) se torna extremamente pequeno, tornando irrelevante quando o quantum chega a 7, além de apresentar uma estagnação na quantidade de instruções média a partir desse valor. Do quantum 8 a 20 fica com estabilidade, isso demonstra que existe um tamanho ideal para os processos, pois nesses quantums não há perdas ou ganhos consideráveis No final, no quantum 21 tem uma pequena melhora.

Tabela 2: Impacto de processos bloqueados e throughput

Quantum	Processo bloqueados simultaneamente (média)	Throughput
1	1,655	0,0719
3	3,898	0,1695
5	0,5227	0,2273
7	0,6053	0,2632
9	0,6216	0,2703
11	0,6571	0,2857
13	0,6571	0,2857
15	0,6765	0,2941
17	0,6765	0,2941
19	0,6765	0,2941
21	0,6970	0,3030

3.2 Análise da tabela 2

Para valores de quantum entre 1 e 5, observa-se um aumento acentuado na eficiência do sistema, com a métrica analisada evoluindo de 0.0719 para 0.2273. Nesta faixa, o sistema apresenta alta ineficiência em Q=1 devido ao excessivo overhead de trocas de contexto, enquanto em Q=3 e Q=5 há significativa melhoria no aproveitamento da CPU. No intervalo de Q=7 a Q=13, o crescimento se estabiliza progressivamente, atingindo o pico de eficiência em Q=11 (0.2857), onde o sistema alcança sua máxima capacidade de processamento. Para quanta iguais ou superiores a 15, os ganhos adicionais tornam-se marginais, não justificando o aumento do quantum devido ao comprometimento da responsividade do sistema e ao crescimento gradual na média de processos bloqueados, que passa de 0.5227 em Q=5 para 0.697 em Q=21, indicando maior tempo de espera para liberação dos processos em estado bloqueado.

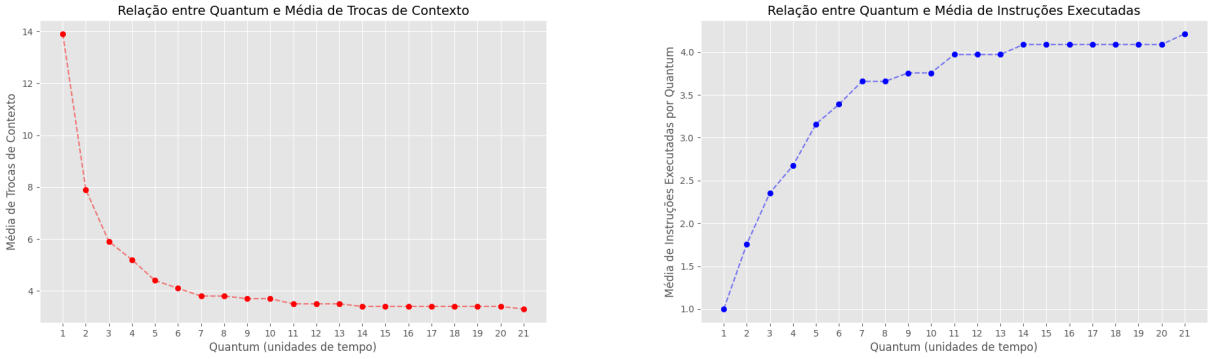

Figura 2: Média de Trocas de Contextos e Média de Instruções Executadas, por quantum.

3.3 Gráfico de Análise para os Quantum

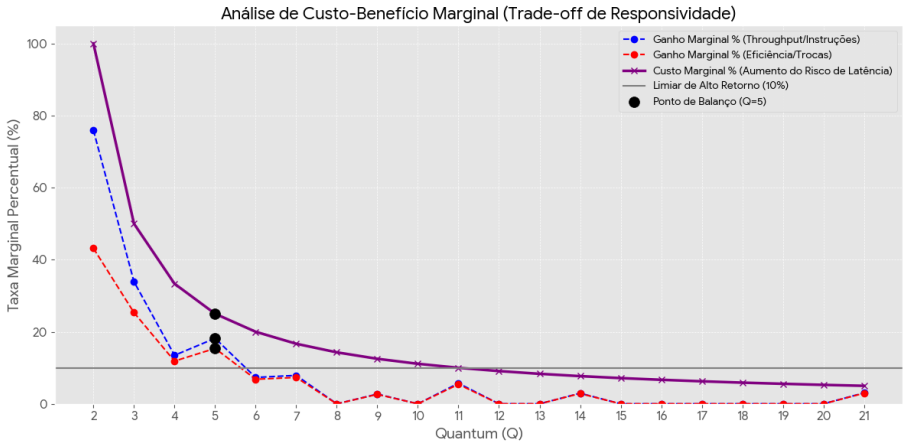

Figura 3: Ganhos marginais e Custos marginais

Conforme observado no gráfico, traçando-se um limiar de ganho de eficiência superior a 10%, ou seja, qualquer aumento no quantum que resulte em um ganho marginal inferior a 10% é considerado um "ponto de saturação"para o investimento em eficiência, pois o custo potencial para a responsividade (aumento da latência de fila) não justifica o pequeno retorno. Pode-se observar que o quantum Q=5 traz o melhor valor de ganho marginal sem comprometer a interatividade do sistema. Essa observação considera que o custo de troca é nulo.

De fato, com Q=5 o escalonador já capturou 89,62% da redução total possível da sobrecarga e 67,22% do aumento total possível do throughput. Um outro ponto de possível melhoria seria o quantum Q=7, que diminuiria relativamente a média de trocas para 3,8 e teria uma média de instruções de 3,65.

Entretanto, devido desconsideramos o custo por troca de contexto nesse projeto, entendemos que um valor menor poderia proporcionar um escalonamento mais "Justo", já que haveria mais variação de processos na CPU. Em uma situação na qual o tempo de troca de contexto fosse levado em conta, provavelmente Q = 7 se tornaria uma opção mais cível, contudo não foi nosso foco analisar essa hipótese.

4 Conclusão

Realizando uma análise sobre os gráficos referentes a eficiência do código, optaríamos e recomendaríamos um quantum igual a 5. Isso porque, além de ser o valor que inicia a saturação entre a média de instruções por quantum (como visualizado no gráfico) ele também é um valor que permite uma quantidade de trocas adequadas e sem ter excesso de overhead, mantendo um equilíbrio ideal

entre throughput e tempo de resposta do sistema.

Por exemplo, com quantum igual 5 tivemos uma média de trocas de 4,4 e média de instruções de 3,1591, que são ótimos valores. A escolha deste valor de quantum (5) representa o ponto ótimo onde os benefícios da redução de trocas de contexto igualam-se aos custos da diminuição da responsividade, se tornando balanceado entre eficiência e tempo de resposta.

Referências

1. O código do projeto está disponível no GitHub: `https://github.com/TheHugoHypothesis/Escalonador`

2. OPENAI. ChatGPT [inteligência artificial]. Versão GPT-5. Disponível em: `https://chat.openai.com/` Acesso em: 10 out. 2025. Para melhorar a coesão do texto previamente escrito, pois apesar de estar rico em conteúdo, existiam muitos erros básicos de português e coesão.