Instituto Federal de Minas Gerais - Campus Ouro Branco
Curso: Sistema de Informação
Matéria: Sistemas Operacionais
Professora: Suelen Mapa de Paula

1º Trabalho Prático de Sistemas Operacionais

Alunos:

Felipe Leal Vieira RA: 0034372

João Victor Dutra Martins Silva RA: 0076873

Kaiky Pires de Souza Cunha RA: 0076893

1. Descrição Geral do Sistema:

1.1 Objetivo do trabalho:

A base do trabalho e seu objetivo principal é implementar dois algoritmos de escalonamento de processos, um preemptivo e outro não-preemptivo, para analisar e comparar seus comportamentos e eficiência em diferentes métricas, como tempo médio de execução (Turnaround Time), tempo médio de espera (Waiting Time) e número de trocas de contexto. Além disso, o projeto busca proporcionar uma interface interativa que permita ao usuário configurar parâmetros relevantes e visualizar os resultados de forma clara e didática.

Outro ponto importante, é aprofundar o aprendizado na área de Sistemas Operacionais, compreendendo melhor os conceitos teóricos de escalonamento de processos e sua aplicação prática. Essa abordagem possibilita o desenvolvimento de habilidades na implementação e análise de algoritmos, promovendo um entendimento mais sólido sobre como sistemas operacionais gerenciam recursos e otimizam o desempenho de processos.

Por sua vez, os algoritmos de escalonamento são de muita importância em sistemas operacionais é destacada pela sua função crítica na alocação eficiente de processadores, garantindo que os processos sejam executados de forma ordenada e justa. Esses algoritmos são fundamentais para maximizar a utilização dos recursos, minimizar os tempos de espera e execução, e oferecer uma experiência de uso eficiente e responsiva, especialmente em ambientes multitarefa.

1.2 Algoritmos escolhidos e justificativa para sua seleção:

1.2.1 Round-Robin:

Sendo Round-Robin um algoritmo preemptivo, isso significa que um tipo de escalonamento ou abordagem em sistemas operacionais onde um processo em execução pode ser interrompido antes de ser concluído para que outro processo tenha acesso ao processador. Este algoritmo foi escolhido por sua ampla aplicação em sistemas multitarefa modernos e suas vantagens. Ele funciona atribuindo a cada processo um tempo fixo (quantum) para execução, garantindo que todos os processos tenham a chance de utilizar o processador em intervalos regulares.

Suas principais vantagens e proporciona um tempo de resposta rápido para processos interativos e assegura equidade na distribuição do tempo de CPU. É ideal para sistemas onde há muitas tarefas concorrentes. Uma grande dificuldade, foi a sua implementação, onde foi desafiador ao criar uma interface gráfica para visualização em tempo real. Além disso, a escolha do quantum é crítica, pois um valor inadequado pode afetar a eficiência do algoritmo, além de enquanto estava sendo feita a implementação, sofremos uma dificuldade para implementar o quantum e fazer com que funcione adequadamente.

1.2.2 Shortest Job First:

Sendo Shortest Job First um algoritmo não preemptivo, este algoritmo foi selecionado por sua eficiência em minimizar o tempo médio de espera. Ele organiza os processos na fila de prontos com base na duração de suas execuções, priorizando os mais curtos.

Sua principal vantagem reduz significativamente o tempo médio de espera e é uma solução ótima em cenários onde a duração dos processos é conhecida previamente. Sendo a maior dificuldade de implementação identificar a duração dos processos de forma precisa para simulação pode ser desafiador, especialmente se há uma interface interativa para o usuário configurar os parâmetros. Além disso, a natureza não preemptiva do algoritmo pode resultar em longos períodos de espera para processos com maior duração.

2. Detalhes de Implementação:

2.1 Pseudocódigo dos algoritmos:

2.1.1 Pseudocódigo do RoundRobin MonoThread:

```
função executarRoundRobinMono(resultado: texto, processosRR: lista de Processo)
    filaProntos <- criarFilaVazia()
    ordenar(processosRR, porTempoChegada)
    tempoAtual <- 0
    totalTurnaroundTime <- 0
    totalWaitTime <- 0
    indiceProcesso <- 0

    enquanto (não filaProntos.vazia() ou indiceProcesso < tamanho(processosRR))
        enquanto (indiceProcesso < tamanho(processosRR) e processosRR[indiceProcesso].tempoChegada <= tempoAtual)
            filaProntos.adicionar(processosRR[indiceProcesso])
            indiceProcesso <- indiceProcesso + 1
        fimenquanto

        se filaProntos.vazia() e indiceProcesso < tamanho(processosRR) então
            tempoAtual <- processosRR[indiceProcesso].tempoChegada
            continuar
        fimse

        processoAtual <- filaProntos.remover()
        tempoExecucao <- mínimo(processoAtual.duracao, quantum)
        resultado <- resultado + "Tempo " + tempoAtual + ": Processo " + processoAtual.id + " executando por " + tempoExecucao + " unidades."
        processoAtual.reduzirDuracao(tempoExecucao)
        tempoAtual <- tempoAtual + tempoExecucao

        enquanto (indiceProcesso < tamanho(processosRR) e processosRR[indiceProcesso].tempoChegada <= tempoAtual)
            filaProntos.adicionar(processosRR[indiceProcesso])
            indiceProcesso <- indiceProcesso + 1
        fimenquanto

        se processoAtual.duracao > 0 então
            filaProntos.adicionar(processoAtual)
        senão
            turnaround <- tempoAtual - processoAtual.tempoChegada
            waiting <- turnaround - processoAtual.duracaoOriginal
            totalTurnaroundTime <- totalTurnaroundTime + turnaround
            totalWaitTime <- totalWaitTime + máximo(waiting, 0)
            resultado <- resultado + "Processo " + processoAtual.id + " finalizado no tempo " + tempoAtual
        fimse
    fimenquanto

    turnaroundTimeRR <- totalTurnaroundTime / tamanho(processosRR)
    waitingTimeRR <- totalWaitTime / tamanho(processosRR)
fimfunção
```

2.1.2 Pseudocódigo do RoundRobin MultiThread:

```
função executarRoundRobinMulti(resultado: texto, processosRR: lista de Processo)
    filaProntos <- criarFilaVazia()
    ordenar(processosRR, porTempoChegada)
    tempoAtual <- 0
    totalTurnaroundTime <- 0
    totalWaitTime <- 0
    indiceProcesso <- 0

    enquanto (não filaProntos.vazia() ou indiceProcesso < tamanho(processosRR))
        enquanto (indiceProcesso < tamanho(processosRR) e processosRR[indiceProcesso].tempoChegada <= tempoAtual)
            filaProntos.adicionar(processosRR[indiceProcesso])
            indiceProcesso <- indiceProcesso + 1
        fimenquanto

        se filaProntos.vazia() e indiceProcesso < tamanho(processosRR) então
            tempoAtual <- processosRR[indiceProcesso].tempoChegada
            continuar
        fimse

        processosParaExecutar <- selecionarProcessosParaExecucao(filaProntos, numProcessadores)
        para cada processoAtual em processosParaExecutar faça
            tempoExecucao <- mínimo(processoAtual.duracao, quantum)
            resultado <- resultado + "Tempo " + tempoAtual + ": Processo " + processoAtual.id + " executando por " + tempoExecucao + " unidades."
            processoAtual.reduzirDuracao(tempoExecucao)
        fimpara

        tempoAtual <- tempoAtual + quantum
        para cada processoAtual em processosParaExecutar faça
            se processoAtual.duracao > 0 então
                filaProntos.adicionar(processoAtual)
            senão
                turnaround <- tempoAtual - processoAtual.tempoChegada
                waiting <- turnaround - processoAtual.duracaoOriginal
                totalTurnaroundTime <- totalTurnaroundTime + turnaround
                totalWaitTime <- totalWaitTime + máximo(waiting, 0)
                resultado <- resultado + "Processo " + processoAtual.id + " finalizado no tempo " + tempoAtual
            fimse
        fimpara
    fimenquanto

    turnaroundTimeRR <- totalTurnaroundTime / tamanho(processosRR)
    waitingTimeRR <- totalWaitTime / tamanho(processosRR)
fimfunção
```

2.1.3 Pseudocódigo do SJF MonoThread:

```
função executarSJFMono(resultado: texto, processosSJF: lista de Processo)
    ordenar(processosSJF, porTempoChegada)
    filaProntos <- criarListaVazia()
    tempoAtual <- 0
    totalTurnaroundTime <- 0
    totalWaitTime <- 0

    enquanto (existeProcessosPendentes(processosSJF))
        para cada processo em processosSJF faça
            se processo.tempoChegada <= tempoAtual e não processo.executado
                filaProntos.adicionar(processo)
            fimse
        fimpara

        ordenar(filaProntos, porDuracao)

        se não filaProntos.vazia() então
            processoAtual <- filaProntos.removerPrimeiro()
            tempoExecucao <- processoAtual.duracao
            resultado <- resultado + "Tempo " + tempoAtual + ": Processo " + processoAtual.id + " executando por " + tempoExecucao + " unidades."
            tempoAtual <- tempoAtual + tempoExecucao
            turnaround <- tempoAtual - processoAtual.tempoChegada
            waiting <- turnaround - processoAtual.duracaoOriginal
            totalTurnaroundTime <- totalTurnaroundTime + turnaround
            totalWaitTime <- totalWaitTime + waiting
            processoAtual.executado <- verdadeiro
        senão
            tempoAtual <- tempoAtual + 1
        fimse
    fimenquanto

    turnaroundTimeSJF <- totalTurnaroundTime / tamanho(processosSJF)
    waitingTimeSJF <- totalWaitTime / tamanho(processosSJF)
fimfunção
```

2.1.4 Pseudocódigo do SJF MultiThread:

```
função executarSJFMulti(resultado: texto, processosSJF: lista de Processo)
    ordenar(processosSJF, porTempoChegada)
    filaProntos <- criarListaVazia()
    tempoAtual <- 0
    totalTurnaroundTime <- 0
    totalWaitTime <- 0

    enquanto (existeProcessosPendentes(processosSJF))
        para cada processo em processosSJF faça
            se processo.tempoChegada <= tempoAtual e não filaProntos.contem(processo) e processo.duracao > 0
                filaProntos.adicionar(processo)
            fimse
        fimpara

        ordenar(filaProntos, porDuracao)
        processosParaExecutar <- criarListaVazia()

        para i de 1 até numProcessadores faça
            se não filaProntos.vazia() então
                processosParaExecutar.adicionar(filaProntos.removerPrimeiro())
            fimse
        fimpara

        maiorTempoExecucao <- 0
```

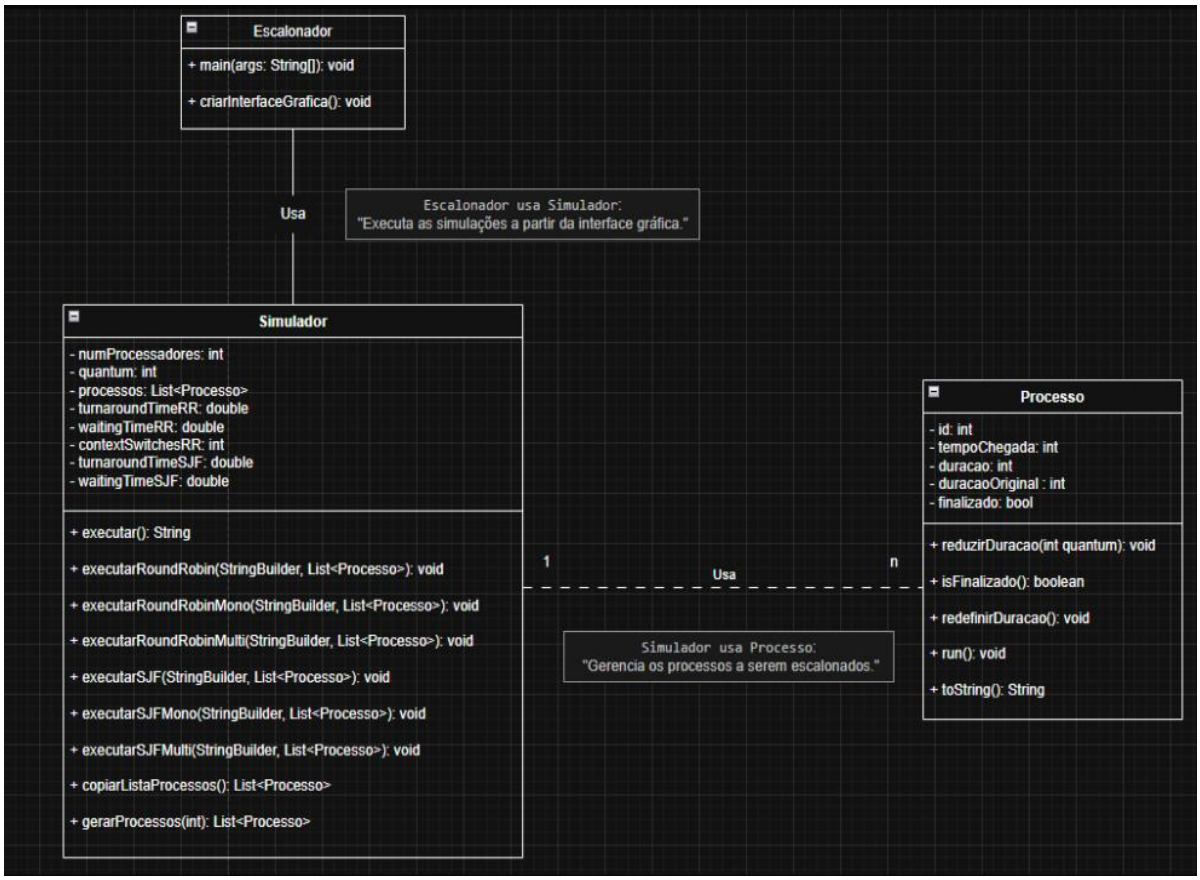
```
        para cada processoAtual em processosParaExecutar faça
            tempoExecucao <- processoAtual.duracao
            resultado <- resultado + "Tempo " + tempoAtual + ": Processo " + processoAtual.id + " executando por " + tempoExecucao + " unidades."
            processoAtual.reduzirDuracao(tempoExecucao)
            maiorTempoExecucao <- máximo(maiorTempoExecucao, tempoExecucao)
        fimpara

        tempoAtual <- tempoAtual + maiorTempoExecucao

        para cada processoAtual em processosParaExecutar faça
            se processoAtual.duracao == 0 então
                turnaround <- tempoAtual - processoAtual.tempoChegada
                waiting <- turnaround - processoAtual.duracaoOriginal
                totalTurnaroundTime <- totalTurnaroundTime + turnaround
                totalWaitTime <- totalWaitTime + máximo(waiting, 0)
                resultado <- resultado + "Processo " + processoAtual.id + " finalizado no tempo " + tempoAtual
            fimse
        fimpara
    fimenquanto

    turnaroundTimeSJF <- totalTurnaroundTime / tamanho(processosSJF)
    waitingTimeSJF <- totalWaitTime / tamanho(processosSJF)
fimfunção
```

2.1.3 Diagrama UML:



2.2 Explicação detalhada do código:

2.2.1 Processo.java:

A classe Processo foi projetada para representar um processo em um sistema de escalonamento, simulando sua execução e comportamento em um ambiente multitarefa. Ela implementa a interface Runnable, permitindo que os objetos dessa classe sejam executados em threads separadas.

```
package escalonamento;

public class Processo implements Runnable {
    private final int id; //identificador unico do processo
    private final int tempoChegada; //tempo em que o processo chega na fila de prontos
    private int duracao; //duracao atual do processo
    private final int duracaoOriginal; //duracao inicial do processo (inalterada)
    private boolean finalizado; //indica se o processo foi concluido
```

Os atributos da classe incluem id, que identifica o processo de forma única; tempoChegada, que registra o momento em que o processo entra na fila de prontos; duracao, que controla o tempo restante para a conclusão do processo; e duracaoOriginal, que armazena a duração inicial para referência futura. Há também o atributo finalizado, que indica se o processo foi concluído.

```
    //construtor para inicializar os atributos do processo
    public Processo(int id, int tempoChegada, int duracao) {
        this.id = id; //inicializa o identificador unico do processo
        this.tempoChegada = tempoChegada; //define o tempo de chegada
        this.duracao = duracao; //define a duracao inicial
        this.duracaoOriginal = duracao; //armazena a duracao inicial para referencia futura
        this.finalizado = false; //inicialmente, o processo nao esta finalizado
    }

    //construtor de copia para criar uma copia do processo
    public Processo(Processo outro) {
        this.id = outro.id; //copia o identificador unico
        this.tempoChegada = outro.tempoChegada; //copia o tempo de chegada
        this.duracao = outro.duracao; //copia a duracao atual
        this.duracaoOriginal = outro.duracaoOriginal; //copia a duracao original
        this.finalizado = false; //a copia do processo nao esta finalizada inicialmente
    }
```

O construtor principal da classe inicializa todos os atributos com valores fornecidos, garantindo que o processo comece com o estado correto. Já o construtor de cópia permite criar uma réplica de outro processo, mantendo as características, mas com o estado finalizado reiniciado.

```java
//metodo para obter o id do processo
public int getId() {
    return id; //retorna o identificador unico do processo
}

//metodo para obter a duracao atual do processo
public synchronized int getDuracao() {
    return duracao; //retorna a duracao atual
}

//metodo para reduzir a duracao do processo em um valor especificado
public synchronized void reduzirDuracao(int quantum) {
    this.duracao -= quantum; //reduz a duracao pelo valor do quantum
    if (this.duracao <= 0) {
        this.finalizado = true; //marca o processo como finalizado se a duracao for zero ou menor
        this.duracao = 0; //garante que a duracao nao seja negativa
    }
}
```

A classe oferece métodos para acessar e manipular os atributos. Por exemplo, o método getId retorna o identificador do processo, enquanto getDuracao e getTempoChegada fornecem informações sobre a duração atual e o tempo de chegada, respectivamente. Para modificar o estado do processo, o método reduzirDuracao permite diminuir o tempo restante, marcando o processo como finalizado caso o tempo atinja zero ou menos. Além disso, o método redefinirDuracao restaura a duração para o valor original e reinicia o estado do processo.

```java
//metodo para obter o tempo de chegada do processo
public int getTempoChegada() {
    return tempoChegada; //retorna o tempo de chegada
}

//metodo para obter a duracao original do processo
public int getDuracaoOriginal() {
    return duracaoOriginal; //retorna a duracao inicial armazenada
}

//metodo para redefinir a duracao para o valor original
public void redefinirDuracao() {
    this.duracao = this.duracaoOriginal; //redefine a duracao para o valor original
    this.finalizado = false; //marca o processo como nao finalizado
}
```

O método isFinalizado verifica se o processo foi concluído. Já o método toString fornece uma representação legível do processo, mostrando informações como o identificador, o tempo de chegada e a duração atual e original.

```java
//metodo para verificar se o processo foi finalizado
public synchronized boolean isFinalizado() {
    return finalizado; //retorna true se o processo estiver finalizado
}
```

A implementação do método run simula a execução do processo em um thread separado. Enquanto o processo não for concluído, ele insere uma pausa de 100 milissegundos, simulando o tempo de processamento. Caso a thread seja interrompida, a execução é encerrada de forma segura.

```java
//metodo para executar o processo em uma thread (simula a execucao)
@Override
public void run() {
    try {
        while (!isFinalizado()) { //enquanto o processo nao estiver finalizado
            Thread.sleep(millis:100); //simula o tempo de execucao do processo
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt(); //interrompe a thread em caso de excecao
    }
}

//metodo para representar o processo como uma string legivel
@Override
public String toString() {
    return "Processo " + id + " - Chegada: " + tempoChegada + " - Duracao Atual: " + duracao +
           " (Original: " + duracaoOriginal + ")"; //retorna uma representacao legivel do processo
}
}
```

Por fim, a classe foi projetada para ser segura em ambientes multithread, utilizando o modificador synchronized nos métodos que manipulam atributos compartilhados, como duracao e finalizado. Isso garante que operações realizadas por diferentes threads não causem inconsistências nos dados. A classe também é extensível, permitindo sua integração com outras partes de um sistema de escalonamento, como algoritmos de planejamento ou gerenciamento de filas.

2.2.2 Simulador.java:

A classe Simulador foi desenvolvida para realizar simulações de algoritmos de escalonamento de processos, permitindo comparar o desempenho do Round-Robin (RR) e do Shortest Job First (SJF). Sua estrutura permite gerenciar múltiplos processadores, manipular listas de processos e calcular métricas importantes como tempos médios de turnaround, tempos médios de espera e o número de trocas de contexto.

```java
public class Simulador {
    private final int numProcessadores; // numero de processadores disponiveis na simulacao
    private final int quantum; // quantum de tempo utilizado no algoritmo Round-Robin
    private final List<Processo> processos; // lista de processos a serem simulados
    private double turnaroundTimeRR; // tempo medio de turnaround no Round-Robin
    private double waitingTimeRR; // tempo medio de espera no Round-Robin
    private int contextSwitchesRR; // numero de trocas de contexto no Round-Robin
    private double turnaroundTimeSJF; // tempo medio de turnaround no Shortest Job First (SJF)
    private double waitingTimeSJF; // tempo medio de espera no SJF

    // construtor para inicializar o simulador com parametros especificos
    public Simulador(int numProcessadores, int quantum, List<Processo> processosPersonalizados, int maxProcessos) {
        this.numProcessadores = numProcessadores; // inicializa o numero de processadores
        this.quantum = quantum; // inicializa o quantum de tempo
        if (processosPersonalizados != null && !processosPersonalizados.isEmpty()) {
            this.processos = processosPersonalizados; // utiliza processos personalizados se fornecidos
        } else {
            this.processos = gerarProcessos(maxProcessos); // gera processos aleatorios caso nao sejam fornecidos
        }
    }
```

A simulação começa com a inicialização do simulador, que pode receber uma lista personalizada de processos ou gerar uma lista de processos aleatórios. O método de geração utiliza valores aleatórios para determinar o tempo de chegada e a duração de cada processo, garantindo que cada processo tenha um identificador único. O construtor também configura o número de processadores e o quantum, que define o tempo máximo de execução de cada processo no algoritmo Round-Robin.

```java
    // metodo para gerar uma lista de processos aleatorios
    private List<Processo> gerarProcessos(int quantidade) {
        List<Processo> lista = new ArrayList<>(); // cria uma nova lista de processos
        Random random = new Random(); // instancia o gerador de numeros aleatorios

        for (int i = 0; i < quantidade; i++) {
            int tempoChegada = random.nextInt(bound:10); // gera um tempo de chegada aleatorio
            int duracao = random.nextInt(bound:10) + 1; // gera uma duracao aleatoria (de 1 a 10)
            lista.add(new Processo(i + 1, tempoChegada, duracao)); // adiciona o processo gerado na lista
        }

        return lista; // retorna a lista de processos gerados
    }
```

O método principal de execução, executar, organiza toda a simulação. Ele exibe informações gerais, como o número de processadores e o quantum, e lista os processos que serão simulados. Em seguida, os algoritmos de escalonamento são executados.

```java
// metodo principal para executar a simulacao
public String executar() {
    StringBuilder resultado = new StringBuilder(); // usado para construir o resultado da simulacao
    resultado.append(str:"=== Simulacao de Escalonamento ===\n");
    resultado.append(str:"Numero de processadores: ").append(numProcessadores).append(str:"\n"); // exibe o numero de
                                                                                                 // processadores
    resultado.append(str:"Quantum para Round-Robin: ").append(quantum).append(str:" unidades\n"); // exibe o quantum
    resultado.append(str:"\n");

    resultado.append(str:"=== Lista de Processos ===\n");
    for (Processo p : processos) {
        resultado.append(String.format(format:"ID: %d | Chegada: %d | Duracao Atual: %d\n",
                p.getId(), p.getTempoChegada(), p.getDuracao())); // exibe os detalhes de cada processo
    }
    resultado.append(str:"\n");

    resultado.append(str:"=== Execucao Round-Robin ===\n");
    executarRoundRobin(resultado, copiarListaProcessos()); // executa o algoritmo Round-Robin
    resultado.append(str:"\n");

    resultado.append(str:"=== Execucao Shortest Job First (SJF) ===\n");
    executarSJF(resultado, copiarListaProcessos()); // executa o algoritmo SJF
    resultado.append(str:"\n");

    resultado.append(str:"=== Resumo ===\n");
    resultado.append(String.format(
            format:"Round-Robin -> Tempo Medio de Turnaround: %.2f | Tempo Medio de Espera: %.2f | Trocas de Contexto: %d\n",
            turnaroundTimeRR, waitingTimeRR, contextSwitchesRR)); // exibe os resultados do Round-Robin
    resultado.append(String.format(format:"SJF         -> Tempo Medio de Turnaround: %.2f | Tempo Medio de Espera: %.2f\n",
            turnaroundTimeSJF, waitingTimeSJF)); // exibe os resultados do SJF
    resultado.append(str:"\n");

    return resultado.toString(); // retorna o resultado completo da simulacao
}
```

Para o Round-Robin, o simulador avalia se será utilizado um único processador ou múltiplos processadores. No caso mono-processador, os processos são organizados em uma fila circular e executados pelo quantum ou até sua conclusão, como mostrado no pseudocódigo ou apresentado no simulador.java. Se o processo não é finalizado, ele é reinserido na fila. Trocas de contexto são contabilizadas sempre que o controle passa de um processo para outro. No caso multiprocessador, o simulador utiliza um pool de threads para executar os processos em paralelo como apresentado no pseudocódigo. Cada thread executa um processo pelo quantum ou até a conclusão, enquanto métricas de desempenho são calculadas conforme os processos terminam.

```java
private void executarRoundRobin(StringBuilder resultado, List<Processo> processosRR) {
    // verifica se o escalonamento Round-Robin sera mono ou multiprocessador
    if (numProcessadores == 1) {
        executarRoundRobinMono(resultado, processosRR); // chama o metodo para execucao mono-processador
    } else {
        executarRoundRobinMulti(resultado, processosRR); // chama o metodo para execucao multi-processador
    }
}
```

Já no algoritmo Shortest Job First (SJF), os processos são ordenados de acordo com a duração restante, sendo o processo mais curto executado primeiro. No caso mono-processador, o processo com menor duração é executado até o fim antes de passar para o próximo. No modo multiprocessador, múltiplos processos com menor duração são atribuídos aos threads disponíveis em um pool, permitindo execução simultânea. Em ambos os casos, os tempos de turnaround e espera são calculados à medida que os processos terminam.

```java
private void executarSJF(StringBuilder resultado, List<Processo> processosSJF) {
    // verifica o numero de processadores e chama o metodo adequado para execucao
    // SJF
    if (numProcessadores == 1) {
        executarSJFMono(resultado, processosSJF);
    } else {
        executarSJFMulti(resultado, processosSJF);
    }
}
```

Durante a execução de ambos os algoritmos, o simulador registra métricas importantes. O tempo de turnaround é calculado como o intervalo entre o tempo de chegada e a conclusão do processo, enquanto o tempo de espera considera o tempo em que o processo permaneceu na fila antes de ser executado. No caso do Round-Robin, o número de trocas de contexto também é monitorado para avaliar o impacto do quantum.

```java
private List<Processo> copiarListaProcessos() {
    //cria uma nova lista para armazenar a copia dos processos
    List<Processo> copia = new ArrayList<>();
    //itera sobre todos os processos na lista original
    for (Processo p : processos) {
        //adiciona uma copia de cada processo na nova lista
        copia.add(new Processo(p));
    }
    //retorna a nova lista com os processos copiados
    return copia;
}

public DefaultCategoryDataset getDataset() {
    //cria um dataset para armazenar os dados de desempenho
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
    //adiciona o tempo de turnaround do Round-Robin ao dataset
    dataset.addValue(turnaroundTimeRR, "Round-Robin", "Tempo de Execução");
    //adiciona o tempo de espera do Round-Robin ao dataset
    dataset.addValue(waitingTimeRR, "Round-Robin", "Tempo de Espera");
    //adiciona o numero de trocas de contexto do Round-Robin ao dataset
    dataset.addValue(contextSwitchesRR, "Round-Robin", "Trocas de Contexto");
    //adiciona o tempo de turnaround do SJF ao dataset
    dataset.addValue(turnaroundTimeSJF, "SJF", "Tempo de Execução");
    //adiciona o tempo de espera do SJF ao dataset
    dataset.addValue(waitingTimeSJF, "SJF", "Tempo de Espera");
    //retorna o dataset preenchido
    return dataset;
}
```

Ao final da simulação, os resultados são organizados e exibidos de forma detalhada. As métricas de desempenho são apresentadas para ambos os algoritmos, destacando diferenças em eficiência e comportamento. Além disso, o simulador oferece uma funcionalidade para gerar um conjunto de dados estruturado (getDataset), permitindo a criação de gráficos para visualização das métricas de desempenho, como tempos médios de turnaround, tempos de espera e trocas de contexto.

A classe também possui funcionalidades auxiliares, como a capacidade de criar cópias da lista original de processos (copiarListaProcessos). Isso garante que as operações realizadas durante a simulação não afetem os dados originais. Além disso, o uso de estruturas sincronizadas e pools de threads permite que o simulador execute processos em ambientes paralelos sem riscos de inconsistência nos dados compartilhados.

2.2.3 Escalanador.java:

O código define uma classe chamada Escalonador, que gerencia a interface gráfica e inicializa a simulação de algoritmos de escalonamento de processos. Essa classe faz parte do pacote escalonamento.

```java
public class Escalonador {
    // Classe principal que gerencia a interface grafica e inicializa a simulacao de
    // escalonamento de processos

    Run | Debug
    public static void main(String[] args) {
        // Metodo principal que inicia a interface grafica da aplicacao
        SwingUtilities.invokeLater(() -> criarInterfaceGrafica());
    }

    private static void criarInterfaceGrafica() {
        // Metodo responsavel por configurar e criar a interface grafica do simulador
        JFrame frame = new JFrame(title:"Simulador de Escalonamento de Processos"); // Janela principal do simulador
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Define o comportamento ao fechar a janela
        frame.setSize(width:1200, height:900); // Define o tamanho da janela

        JPanel mainPanel = new JPanel(new BorderLayout()); // Painel principal com layout de borda
        JTabbedPane tabbedPane = new JTabbedPane(); // Abas para organizar as funcionalidades
```

O método main é responsável por inicializar a aplicação e criar a interface gráfica. Ele utiliza SwingUtilities.invokeLater para garantir que a interface seja configurada na thread correto e que a aplicação continue responsiva durante a execução.

A janela principal do simulador é criada utilizando JFrame. Ela possui um tamanho predefinido e é configurada para encerrar a aplicação ao ser fechada. O painel principal (mainPanel) utiliza o layout BorderLayout para organizar os componentes, e as abas são criadas com JTabbedPane.

```
JPanel simuladorPanel = new JPanel(new BorderLayout());

JPanel inputPanel = new JPanel(); // Painel para entrada de parametros
inputPanel.setLayout(new BoxLayout(inputPanel, BoxLayout.Y_AXIS)); // Organiza os componentes em uma coluna

// Obtem o numero maximo de threads disponiveis no sistema
int maxThreads = Runtime.getRuntime().availableProcessors();
Integer[] threadOptions = new Integer[maxThreads];
for (int i = 0; i < maxThreads; i++) {
    threadOptions[i] = i + 1; // Popula as opcoes de threads de 1 ate o maximo
}
JComboBox<Integer> numProcessadoresDropdown = new JComboBox<>(threadOptions); // Dropdown para selecionar numero
                                                                               // de processadores
JTextField numProcessosField = new JTextField(columns:10); // Campo de entrada para o numero de processos
JTextField quantumField = new JTextField(columns:10); // Campo de entrada para o valor do quantum

inputPanel.add(new JLabel(text:"Numero de Processadores:")); // Rotulo para o numero de processadores
inputPanel.add(numProcessadoresDropdown); // Adiciona o dropdown ao painel
inputPanel.add(new JLabel(text:"Numero de Processos na Fila de Prontos:")); // Rotulo para numero de processos
inputPanel.add(numProcessosField); // Adiciona o campo de texto ao painel
inputPanel.add(new JLabel(text:"Quantum (para Round-Robin):")); // Rotulo para o quantum
inputPanel.add(quantumField); // Adiciona o campo de texto ao painel
```

A aba de simulação é configurada para coletar parâmetros de entrada, exibir resultados e gráficos, e organizar os indicadores de desempenho. Ela é composta por um painel de entrada, uma área de texto para exibição de resultados e um painel inferior com gráficos e indicadores.

```
// Botao para iniciar a simulacao
JButton iniciarButton = new JButton(text:"Iniciar Simulacao");
inputPanel.add(iniciarButton); // Adiciona o botao ao painel
iniciarButton.setBackground(new Color(rgb:0xE30613));
iniciarButton.setForeground(Color.WHITE);
iniciarButton.setFont(new Font(name:"Arial", Font.BOLD, size:14));
simuladorPanel.add(inputPanel, BorderLayout.NORTH); // Adiciona o painel de entrada ao topo
// Painel para centralizar o botão
JPanel botaoPanel = new JPanel(new FlowLayout(FlowLayout.CENTER)); // Painel com layout de centralização
botaoPanel.add(iniciarButton); // Adiciona o botão ao painel
botaoPanel.setBorder(BorderFactory.createEmptyBorder(top:10, left:0, bottom:10, right:0)); // Adiciona uma margem superior e inferior

// Adiciona o painel do botão ao painel de entrada
inputPanel.add(botaoPanel);
// Area de texto para exibicao dos resultados da simulacao
JTextArea outputArea = new JTextArea(rows:20, columns:50);
outputArea.setEditable(b:false); // Define a area como nao editavel
JScrollPane scrollPane = new JScrollPane(outputArea); // Adiciona barra de rolagem
```

Um botão chamado "Iniciar Simulação" é configurado para iniciar a simulação com os parâmetros fornecidos. Abaixo dele, há uma área de texto (JTextArea) onde os resultados da simulação são exibidos de forma clara, com suporte para barra de rolagem.

```
// Painel para exibicao do grafico
JPanel graficoPanel = new JPanel();
graficoPanel.setPreferredSize(new Dimension(width:1000, height:300)); // Define o tamanho do painel
graficoPanel.setBorder(BorderFactory.createLineBorder(new Color(rgb:0x009639), thickness:2));
JScrollPane scrollPaneGrafico = new JScrollPane(graficoPanel); // Adiciona barra de rolagem ao grafico
bottomPanel.add(scrollPaneGrafico, BorderLayout.NORTH); // Adiciona o grafico na parte superior
```

Na parte inferior da aba de simulação, é configurado um painel que exibe gráficos gerados durante a simulação e indicadores de desempenho como tempo médio de execução e trocas de contexto. Os gráficos são exibidos no painel graficoPanel e os indicadores são organizados em uma tabela.

```
// Painel para criacao de processos personalizados
JPanel processosPanel = new JPanel(new BorderLayout());
DefaultTableModel processosTableModel = new DefaultTableModel(); // Modelo da tabela de processos
processosTableModel.addColumn(columnName:"ID"); // Coluna para ID do processo
processosTableModel.addColumn(columnName:"Chegada"); // Coluna para tempo de chegada
processosTableModel.addColumn(columnName:"Duracao"); // Coluna para duracao
processosTableModel.addColumn(columnName:"Prioridade"); // Coluna para prioridade

JTable tabelaProcessos = new JTable(processosTableModel); // Tabela baseada no modelo
JScrollPane scrollPaneProcessos = new JScrollPane(tabelaProcessos); // Adiciona barra de rolagem
scrollPaneProcessos.setBorder(BorderFactory.createLineBorder(new Color(rgb:0x009639), thickness:2));
JPanel addProcessoPanel = new JPanel(); // Painel para entrada de dados de processos
JTextField idField = new JTextField(columns:5); // Campo para ID do processo
JTextField chegadaField = new JTextField(columns:5); // Campo para tempo de chegada
JTextField duracaoField = new JTextField(columns:5); // Campo para duracao do processo
JTextField prioridadeField = new JTextField(columns:5); // Campo para prioridade
JButton addProcessoButton = new JButton(text:"Adicionar Processo"); // Botao para adicionar processo

addProcessoPanel.add(new JLabel(text:"ID:")); // Rotulo para ID
addProcessoPanel.add(idField); // Adiciona o campo para ID
addProcessoPanel.add(new JLabel(text:"Chegada:")); // Rotulo para tempo de chegada
addProcessoPanel.add(chegadaField); // Adiciona o campo para tempo de chegada
addProcessoPanel.add(new JLabel(text:"Duracao:")); // Rotulo para duracao
addProcessoPanel.add(duracaoField); // Adiciona o campo para duracao
addProcessoPanel.add(new JLabel(text:"Prioridade:")); // Rotulo para prioridade
addProcessoPanel.add(prioridadeField); // Adiciona o campo para prioridade
addProcessoPanel.add(addProcessoButton); // Adiciona o botao para adicionar

processosPanel.add(scrollPaneProcessos, BorderLayout.CENTER); // Adiciona a tabela ao centro
processosPanel.add(addProcessoPanel, BorderLayout.SOUTH); // Adiciona o painel de entrada na parte inferior

tabbedPane.add(title:"Criar Processos", processosPanel); // Adiciona a aba de criacao de processos
```

Essa aba permite ao usuário criar processos personalizados para a simulação. Os processos são organizados em uma tabela, e o usuário pode adicionar dados como ID, tempo de chegada, duração e prioridade.

```
// Painel para exibicao do historico de simulacoes
JPanel historicoPanel = new JPanel(new BorderLayout());
DefaultTableModel historicoTableModel = new DefaultTableModel(); // Modelo da tabela de historico
historicoTableModel.addColumn(columnName:"Numero de Processadores"); // Coluna para numero de processadores
historicoTableModel.addColumn(columnName:"Processos"); // Coluna para numero de processos
historicoTableModel.addColumn(columnName:"Quantum"); // Coluna para quantum
historicoTableModel.addColumn(columnName:"Tempo Medio de Execucao (RR)"); // Coluna para tempo medio de execucao no
                                                                          // Round-Robin
historicoTableModel.addColumn(columnName:"Tempo Medio de Espera (RR)"); // Coluna para tempo medio de espera no Round-Robin
historicoTableModel.addColumn(columnName:"Trocas de Contexto (RR)"); // Coluna para trocas de contexto no Round-Robin
historicoTableModel.addColumn(columnName:"Tempo Medio de Execucao (SJF)"); // Coluna para tempo medio de execucao no SJF
historicoTableModel.addColumn(columnName:"Tempo Medio de Espera (SJF)"); // Coluna para tempo medio de espera no SJF

JTable tabelaHistorico = new JTable(historicoTableModel); // Tabela baseada no modelo
JScrollPane scrollPaneHistorico = new JScrollPane(tabelaHistorico); // Adiciona barra de rolagem
historicoPanel.add(scrollPaneHistorico, BorderLayout.CENTER); // Adiciona a tabela ao centro do painel

tabbedPane.add(title:"Historico", historicoPanel); // Adiciona a aba de historico
```

A aba "Histórico" exibe os resultados de simulações anteriores. Os dados de cada execução são organizados em uma tabela para que o usuário possa analisar o desempenho dos algoritmos ao longo do tempo.

```
        mainPanel.add(tabbedPane, BorderLayout.CENTER); // Adiciona as abas ao painel principal
        frame.add(mainPanel); // Adiciona o painel principal a janela
        frame.setVisible(b:true); // Torna a janela visivel
    }
}
```

A janela principal é finalizada com a adição do painel principal e torna-se visível para o usuário.

2.3 Representações gráficas da interface do sistema:

Representação gráfica da home do simulador.

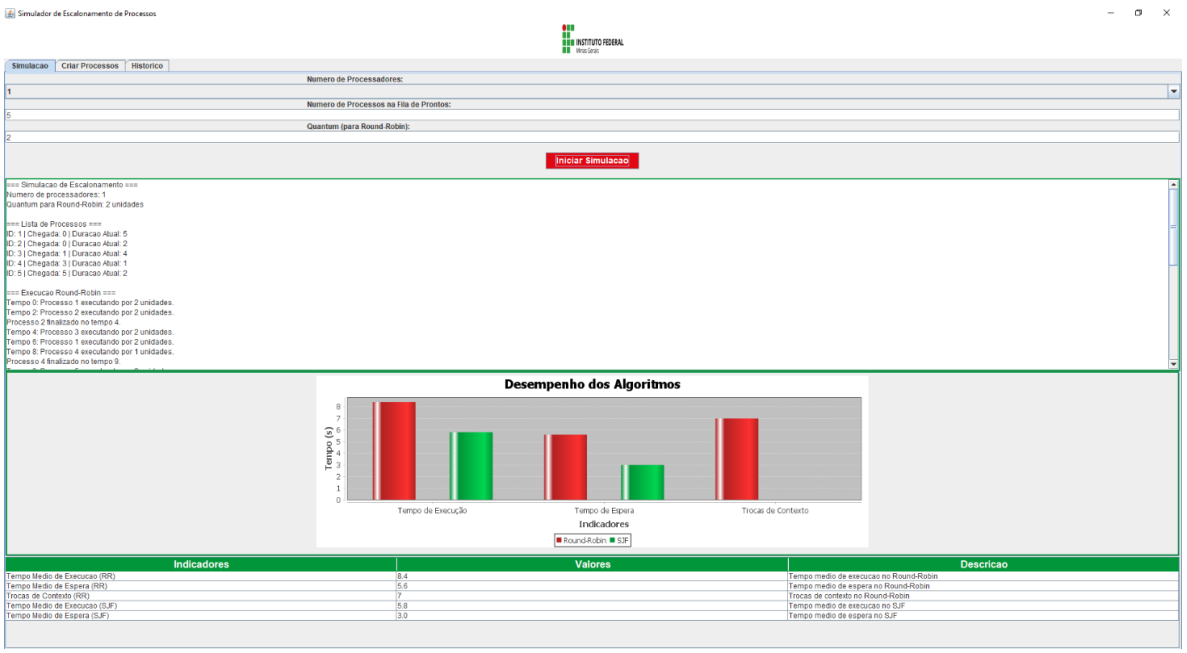
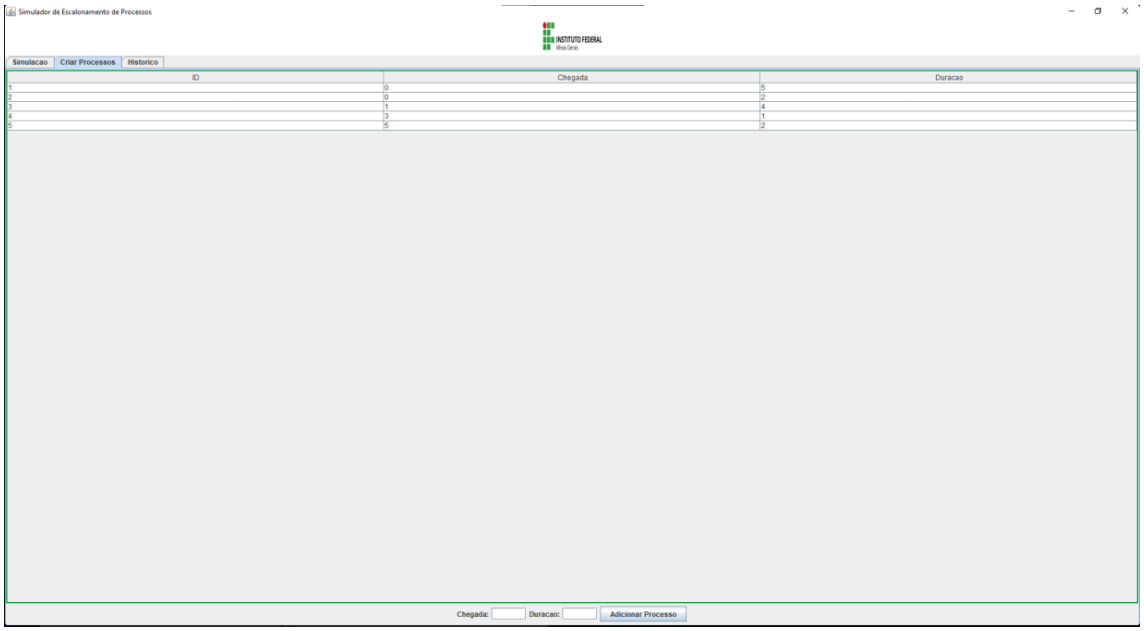


Imagem da página histórico do simulador, mostrando que e possível obter o histórico de simulações feitas.

Imagem mostrando a página de criação dos processos, na qual e possível criar processos e caso optar por não criar processos, a simulação ira utilizar uma logica para criar processos randômicos.



3. Análise dos Resultados:

3.1 Comparação entre os dois algoritmos implementados:

3.1.1 Pontos fortes e fracos de cada um:

Round-Robin (RR):

Pontos Fortes:

- Equidade: O RR garante que todos os processos recebam tempo de CPU de forma justa, evitando a fome de processos (starvation).
- Previsibilidade: Ao usar um quantum fixo, o algoritmo fornece uma estimativa previsível para o tempo máximo que um processo pode esperar até ser executado novamente.
- Adequado para ambientes multitarefa: Ideal para sistemas interativos onde todos os processos precisam de uma parte regular da CPU.

Pontos Fracos:

- Alto número de trocas de contexto: Um quantum muito pequeno resulta em frequentes trocas de contexto, o que impacta negativamente o desempenho geral do sistema.

- Ineficiente para processos curtos: Processos pequenos podem ser interrompidos desnecessariamente, aumentando seu tempo médio de execução e espera.

Shortest Job First (SJF):

Pontos Fortes:

- Redução do tempo médio: Priorizar os processos com menor duração reduz significativamente os tempos médios de espera e execução.
- Eficiência em cargas previsíveis: Funciona muito bem em sistemas onde os tempos de execução dos processos são conhecidos ou previsíveis.

Pontos Fracos:

- Não-preemptividade: Processos de longa duração que chegam antes podem bloquear a execução de processos mais curtos.
- Dependência de previsões: O SJF requer conhecimento prévio do tempo de execução dos processos, o que pode ser difícil de obter em sistemas dinâmicos.
- Possível fome de processos: Processos longos podem ser constantemente adiados se novos processos curtos continuarem a chegar.

3.1.2 Impacto dos parâmetros ajustados nos tempos médios e no número de trocas de contexto:

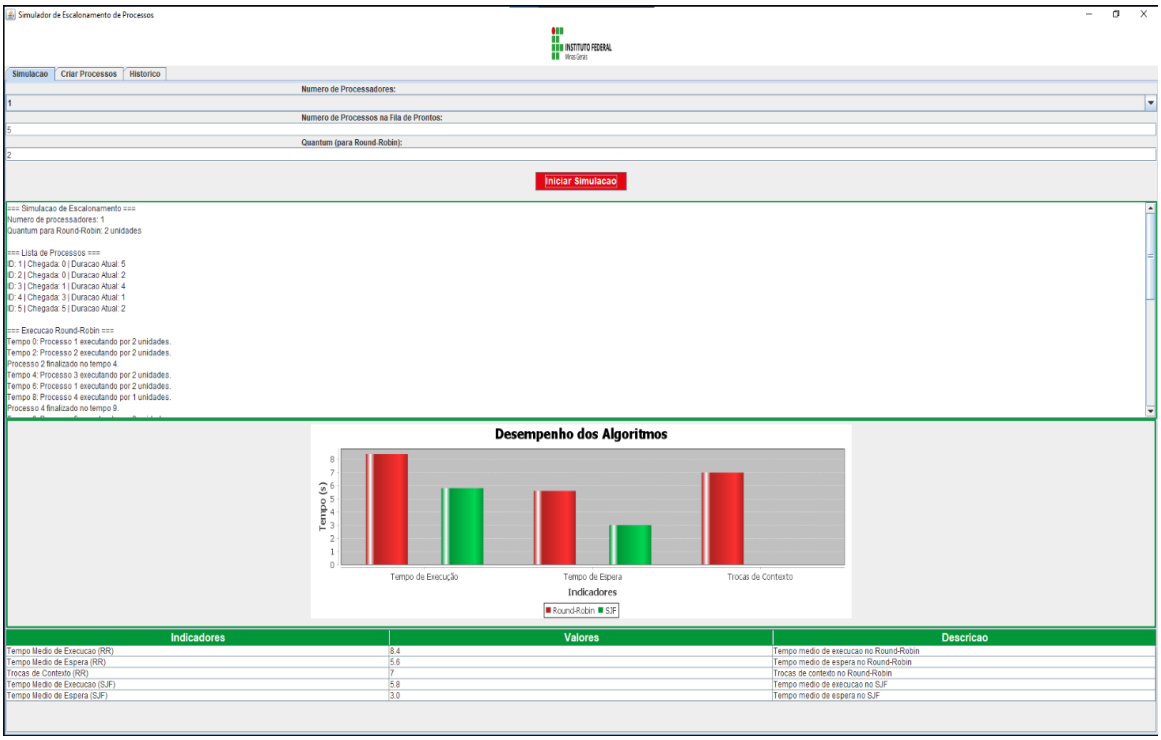
Round-Robin (RR):

1. Quantum:

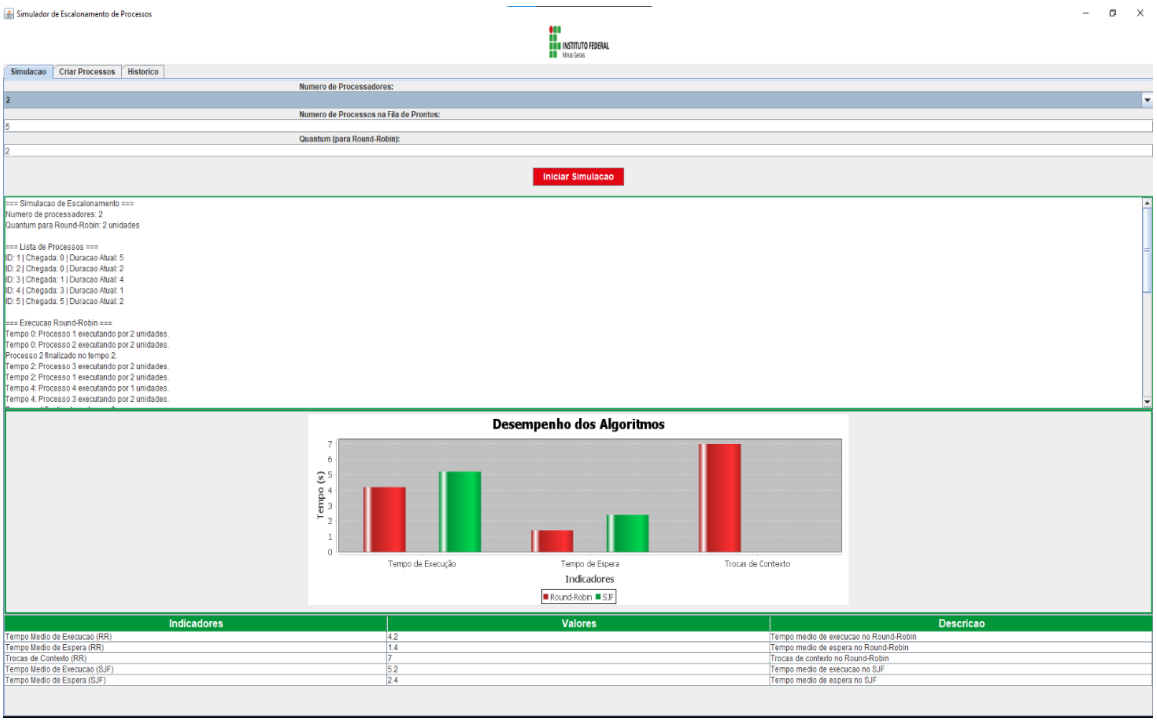
- Um quantum muito pequeno leva a um número elevado de trocas de contexto, aumentando o overhead do sistema e prejudicando o desempenho.
- Um quantum muito grande reduz as trocas de contexto, mas aproxima o algoritmo de um comportamento FCFS (First-Come, First-Served), prejudicando a equidade.

2. Número de processadores (threads):

- Com mais processadores disponíveis, o RR distribui melhor a carga de trabalho, reduzindo o tempo médio de execução. Porém, em sistemas com um único processador, o tempo de espera e execução tendem a ser maiores.



Acima uma simulação utilizando apenas 1 thread e logo abaixo, usando 2 threads, foram utilizados nas duas simulações, os mesmos valores.



Shortest Job First (SJF):

1. Tempo de chegada dos processos:

- Processos com tempos de chegada tardios podem ser penalizados, especialmente se processos longos já estiverem em execução.

2. Duração dos processos:

- A eficiência do SJF depende diretamente da precisão na estimativa dos tempos de execução. Erros na estimativa podem comprometer o tempo médio de execução e espera.

3. Número de processadores (threads):

- Em sistemas com múltiplos processadores, o SJF pode executar processos paralelamente, o que reduz os tempos médios. Porém, a eficiência depende de como os processos são distribuídos.

Ao analisar os algoritmos do Round-Robin (RR) e do Shortest Job First (SJF) revelou diferenças importantes que afetam seu desempenho e aplicabilidade. O RR é um algoritmo justo, ideal para sistemas que exigem equidade entre os processos, mas sofre com o impacto direto do valor do quantum. Um quantum muito pequeno aumenta significativamente o número de trocas de contexto, reduzindo a eficiência do sistema, enquanto um quantum maior pode aproximá-lo de um comportamento semelhante ao First-Come, First-Served (FCFS), prejudicando a equidade. Já o SJF mostrou-se mais eficiente em termos de tempos médios de execução e espera, priorizando processos mais curtos. No entanto, sua dependência de informações precisas sobre a duração dos processos e o risco de fome de CPU para tarefas mais longas limitam sua aplicabilidade em alguns cenários.

Ao ajustar os parâmetros de ambos os algoritmos têm impacto direto nos resultados. No RR, a escolha do quantum afeta tanto os tempos médios quanto o número de trocas de contexto. No SJF, a variação nos tempos de chegada e duração dos processos influencia

sua eficiência, destacando a necessidade de equilíbrio entre priorização e justiça. Em resumo, a escolha entre os algoritmos deve considerar as demandas específicas do sistema, sendo o RR mais apropriado para contextos em que a equidade é essencial, enquanto o SJF se destaca em sistemas com baixa variação nos tempos de duração dos processos.

4.Conclusão:

O aprendizado técnico adquirido durante este trabalho foi extenso e abrangeu diversas áreas fundamentais da computação. A implementação dos algoritmos de escalonamento exigiu um domínio sólido de conceitos teóricos, como tempo de execução, tempo de espera, trocas de contexto e a diferenciação entre algoritmos preemptivos e não preemptivos. No entanto, transformar esse conhecimento teórico em um sistema funcional revelou ser um processo desafiador e enriquecedor.

Do ponto de vista prático, o trabalho exigiu o uso de estruturas de dados eficientes para lidar com as filas de processos, o que nos levou a explorar o funcionamento detalhado de listas, filas prioritárias e coleções sincronizadas em Java. Também foi necessário implementar mecanismos de paralelismo utilizando “ExecutorService” para simular múltiplos processadores. Isso proporcionou uma visão mais profunda sobre concorrência, sincronização e como evitar problemas como condições de corrida e inconsistências nos dados.

Outro aspecto técnico que demandou atenção foi o tratamento adequado de métricas de desempenho. Implementar os cálculos de tempos médios de execução e espera, assim como contabilizar as trocas de contexto, exigiu uma abordagem detalhista para evitar erros que pudessem comprometer a precisão dos resultados. Foi fundamental entender como essas métricas são inter-relacionadas e como cada parâmetro ajustado, como o quantum no Round-Robin, impacta diretamente o desempenho do sistema.

A comparação entre os dois algoritmos, Round-Robin e Shortest Job First (SJF), foi particularmente esclarecedora. O Round-Robin mostrou-se eficaz em garantir uma divisão justa de tempo entre os processos, mas seu desempenho foi diretamente impactado pela escolha do quantum, que exigiu ajustes cuidadosos para equilibrar trocas de contexto e tempos médios. Já o SJF destacou-se pela eficiência em reduzir os tempos médios de espera e execução, porém revelou sua limitação em cenários onde o conhecimento antecipado da duração dos processos não está disponível.

A implementação prática reforçou habilidades em depuração e testes, especialmente em um ambiente onde pequenos erros podem levar a resultados significativamente diferentes dos esperados. A identificação de bugs, como a ordem incorreta de execução no SJF ou inconsistências no cálculo de tempos no Round-Robin, nos ensinou a importância de validar os resultados com exemplos teóricos e realizar testes exaustivos em diferentes cenários.

Além disso, foi necessário aprimorar a compreensão de interfaces gráficas e como apresentar os resultados de forma clara e acessível. A criação de gráficos e tabelas que sintetizassem os dados reforçou a importância de traduzir informações técnicas em visualizações compreensíveis para facilitar a análise e a interpretação.

Em resumo, este trabalho não apenas consolidou nosso conhecimento teórico, mas também expandiu significativamente nossa habilidade de aplicar conceitos de sistemas operacionais em um contexto prático, trabalhar com programação concorrente, depurar problemas complexos e apresentar resultados de forma clara e fundamentada. O aprendizado técnico foi um marco importante, preparando-nos para desafios futuros em projetos ainda mais complexos.