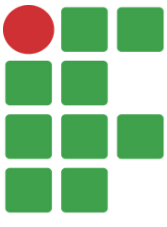

INSTITUTO FEDERAL
Triângulo Mineiro

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO TRIÂNGULO MINEIRO

ANALISE E DESENVOLVIMENTO DE SISTEMAS

**Trabalho de Sistemas Operacionais**

Patrocínio - MG

2023

**Integrantes do Grupo:**
Alejandro Gonzaga Leles
Ana Luiza dos Santos
Lucas Lemos Pavesi
Paulo Henrique Dirino

# Trabalho de Sistemas Operacionais

Patrocínio – MG
2023

## Introdução

O escalonamento de processos é fundamental na ciência da computação e é crucial para a eficiência dos sistemas operacionais. O algoritmo SRTF, escolhido para este trabalho, destaca-se pela sua capacidade de otimizar o tempo de resposta médio minimizando o tempo que cada processo espera na fila.

## Objetivos

O objetivo deste trabalho foi entender a lógica por trás do escalonamento SRTF, implementá-lo em C e testar sua eficácia em diferentes cenários.

## Metodologia

O algoritmo foi implementado em C, fazendo uso de estruturas para representar cada processo. Durante a implementação, houve um desafio em gerenciar entradas inválidas do usuário, que foi resolvido com a criação de funções específicas.

## Funcionamento do Algoritmo

O algoritmo SRTF seleciona o processo com o menor tempo restante de execução. Se dois processos têm o mesmo tempo restante, o primeiro na ordem de entrada é escolhido.

## Imagens do código (com comentários):

```c
// Inclusão das bibliotecas necessárias
#include <stdio.h>    // Usada para operações de entrada e saída (como printf e scanf)
#include <stdbool.h>  // Usada para usar tipos booleanos (true e false)
#include <limits.h>   // Usada para definir constantes, como INT_MAX

// Definição da estrutura de um processo
typedef struct {
    char nome[10];          // Nome do processo
    int tempoChegada;       // Tempo em que o processo chega à fila
    int tempoExecucao;      // Tempo total que o processo leva para executar
    int tempoRestante;      // Tempo restante para o processo finalizar
    bool finalizado;        // Flag indicando se o processo foi finalizado
} Processo;

// Função para limpar o buffer de entrada
// Isso é necessário porque, às vezes, o buffer contém caracteres indesejados que podem afetar a leitura subsequente
void limparBuffer() {
    int ch;
    while ((ch = getchar()) != '\n' && ch != EOF);
}

// Função que lê um inteiro do usuário e trata erros
// Continua pedindo uma entrada até que o usuário insira um número inteiro válido
int lerInteiro(const char* mensagem) {
    int valor;             // Valor que será retornado
    char linha[256];       // Buffer para armazenar a entrada do usuário

    // Loop até que uma entrada válida seja fornecida
    while (1) {
        printf("%s", mensagem);
        // fgets lê uma linha do stdin e sscanf tenta interpretar essa linha como um inteiro
        if (fgets(linha, sizeof(linha), stdin) && sscanf(linha, "%d", &valor) == 1) {
            return valor;  // Retorna o valor lido se for válido
        }
        // Mensagem de erro para o usuário
        printf("Entrada invalida! Tente novamente.\n");
```

```
    }
}

int main() {
    Processo processos[15];  // Array para armazenar até 15 processos
    int n;                   // Número de processos que o usuário deseja inserir
    int tempoAtual = 0;      // Mantém o controle do tempo atual durante a execução dos processos
    bool todosFinalizados;   // Flag para verificar se todos os processos foram finalizados

    // Ler o número de processos
    do {
        n = lerInteiro("Quantos processos voce deseja inserir (1-15)? ");
    } while (n < 1 || n > 15);

    // Ler detalhes de cada processo
    for(int i = 0; i < n; i++) {
        printf("Nome do processo (ex: A): ");
        scanf("%s", processos[i].nome);
        limparBuffer();

        // Ler o tempo de execução do processo e garantir que seja positivo
        do {
            processos[i].tempoExecucao = lerInteiro("Tempo de uso da CPU: ");
        } while (processos[i].tempoExecucao <= 0);

        // Ler o tempo de chegada do processo e garantir que não seja negativo
        do {
            processos[i].tempoChegada = lerInteiro("Tempo de chegada do processo: ");
        } while (processos[i].tempoChegada < 0);

        // Inicializar os valores restantes
        processos[i].tempoRestante = processos[i].tempoExecucao;
        processos[i].finalizado = false;
    }

   // Loop principal do escalonador SRTF (Shortest Remaining Time First)
   do {
        int idx = -1;                          // Índice do processo que será executado neste ciclo
        int menorTempoRestante = INT_MAX;      // Usado para encontrar o processo com o menor tempo restante
        todosFinalizados = true;               // Supõe que todos os processos estão finalizados e ajusta conforme necessário

        // Varre todos os processos para encontrar o que deve ser executado
        for(int i = 0; i < n; i++) {
            if(processos[i].tempoChegada <= tempoAtual && !processos[i].finalizado && processos[i].tempoRestante < menorTempoRestante) {
                menorTempoRestante = processos[i].tempoRestante;
                idx = i;
            }

            // Se algum processo não estiver finalizado, ajusta a flag
            if(!processos[i].finalizado) {
                todosFinalizados = false;
            }
        }

        // Se encontramos um processo para executar
        if(idx != -1) {
            processos[idx].tempoRestante -= 1;   // Reduz o tempo restante do processo
            printf("%d-%s ", tempoAtual, processos[idx].nome);  // Imprime o processo em execução
            tempoAtual++;                        // Avança o tempo

            // Se o tempo restante do processo for 0, ele foi concluído
            if(processos[idx].tempoRestante == 0) {
                processos[idx].finalizado = true;
            }
        } else {
            // Se nenhum processo pode ser executado neste ciclo, apenas avança o tempo
            tempoAtual++;
        }

    } while(!todosFinalizados);  // Continua o loop enquanto houver processos não finalizados

    return 0;  // Termina o programa
}
```

## Resultados

Durante os testes, observou-se que o algoritmo efetivamente minimizou o tempo de espera na fila, especialmente em situações com muitos processos de durações variadas.

## Conclusão

A implementação do algoritmo SRTF mostrou-se eficaz e ofereceu insights valiosos sobre a otimização do tempo de resposta em sistemas operacionais. Apesar dos desafios iniciais, o resultado final foi uma implementação robusta e eficiente.