RELATÓRIO DE SISTEMAS OPERACIONAIS

Projeto – Algoritmo de Escalonamento de Processo

201940601013 – Luis Felipe Soares Coelho

201940601025 – Manoel Malon Costa de Moura

Marabá

2021

UNIVERSIDADE FEDERAL DO SUL E SUDESTE DO PARÁ
INSTITUTO DE GEOCIÊNCIAS E ENGENHARIAS
FACULDADE DE COMPUTAÇÃO E ENGENHARIA ELÉTRICA

LUIS FELIPE SOARES COELHO
MANOEL MALON COSTA MOURA

Projeto – Algoritmo de Escalonamento de Processo

Relatório referente ao projeto de Sistemas Operacionais denominado ALGORITMO DE ESCALONAMENTO DE PROCESSO como critério de avaliação da disciplina Sistemas Operacionais do professor João Victor Costa Carmona.

Marabá

2021

SUMÁRIO

1. INTRODUÇÃO

A priori o algoritmo tem por finalidade escalonar processos, sendo os 10 primeiros processos não-preemptivos nos primeiros ciclos de execução, e quando haver sinalização, o mesmo algoritmo deve passar a trabalhar de forma preemptiva. Sendo assim foi feito um programa na linguagem de programação C contendo dois algoritmos de escalonamento, após a seleção dos algoritmos o código divido em três partes para que haja uma facilidade no escalonamento e na programação, o próximo passo foi como deveria ser tudo apresentado ao usuário e qual seria a melhor forma de dar prioridade aos processos, após a resolução desses empecilhos de desenvolvimento foi iniciado o código com todo o planejamento necessário, a última grande dificuldade foi como tirar um nó da fila pelo seu ID.

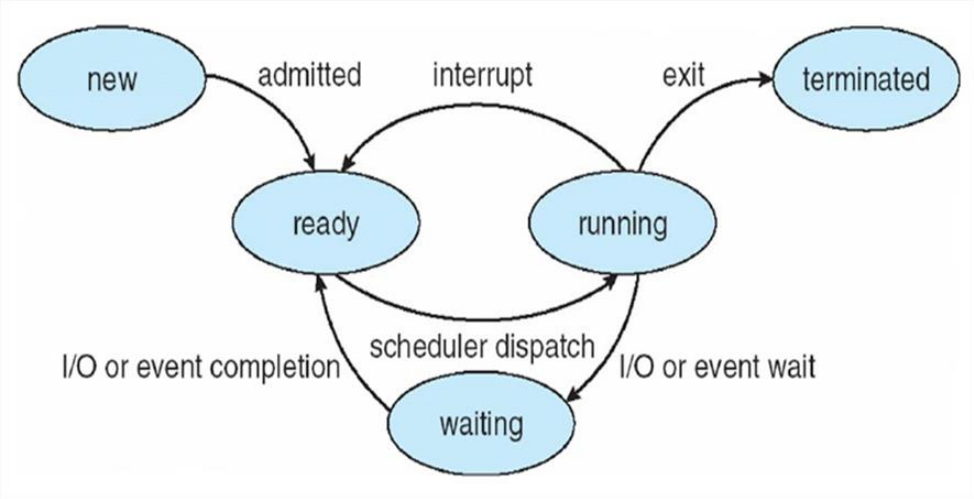

Figura 1 - Ciclo de Escalonamento.

Esse é o Ciclo de Escalonamento, basicamente é o que está sendo implemento no programa pelos escalonadores, onde o novo processo vai para o estado de **pronto** depois para executando. E se o processo for do tipo **CPU-Bound** ele ira para terminado, caso o processo seja do tipo **I/O-Bound** ele ira para **espera** depois novamente para o estado de **pronto** e será executado de novo e finalizado.

2. OBJETIVO

O principal objetivo foi criar o algoritmo para os 10 primeiros processos não serem preemptivos, sendo assim foi escolhido o FIFO (do inglês: first in, first out) sendo um algoritmo de fila simples, assim solicitando a CPU a receber primeiro. Depois o escalonamento deveria ser preemptivo e foi escolhido o Round-robin(RR) qual sua função foi atribuir frações de tempo para cada processo em partes iguais e de forma circular.

3. SEPARAÇÃO DE ATIVIDADES

O programa foi feito pelo Manuel Malon, sendo ele o programador principal e com a responsabilidade de escolher o algoritmo, com isso o aluno Luis Felipe, ficou com a responsabilidade de revisar o código e fazer o relatório, e o aluno Malon ficaria com a revisão do relatório, cada atividade estaria em equilíbrio e devidamente implementada em seu tempo.

4. METODOLOGIA E DESENVOLVIMENTO

No início o desafio foi escolher os algoritmos, sendo eles preemptivos e não-preemptivos. Sabendo disso foi selecionado os algoritmos Round-robin e FIFO. Round-robin é o algoritmo preemptivo e o FIFO o algoritmo não preemptivo. descrever brevemente os materiais usados. Mostrar uma foto do circuito. Colocar o código usado e explicá-lo. Colocar as possíveis modificações feitas no código e os resultados obtidos, se for o caso. Explicar se algum material interfere ou não no experimento, se for o caso. Explicar os resultados. Primeira definição foram os parâmetros apresentados ao usuário, então, demostrasse a listagem de processos contendo o Processo gerado por um número aleatório, Prioridade Gerada pelo... tempo de duração e os tipos de processo CPU-BOUND e I/O-BOUND e. Iniciando pelo escalonamento FIFO, foi delimitado para os primeiros 10 processos criados sejam não-preemptivos, a lógica dele está associada a prioridade, sendo sua prioridade o processo que chegar primeiro para execução, gerando assim uma facilidade por necessitar apenas de uma fila de processos. Sendo pré-definido o tempo de cada tipo de processo, deixando cada processo ser executado num tempo máximo e fazendo ele retorna a fila após ultrapassar esse tempo de processamento. Com o antecessor não-preemptivo, o próximo escalonador denominado Round-robin foi escolhido por levar em consideração o tempo de execução de cada processo e a sua prioridade gerada de 0 a 5. Sendo assim o processamento será levando em conta a prioridade gerada e caso ocorra de vários processamentos terem a mesma prioridade eles serão processados pela ordem de chegada.

Inicialmente o código foi divido em três partes, o programa principal, o TAD (onde fica os protótipos das funções) e as implementações das funções. É interessante utilizar o Tipo Abstrato de Dados porque divide o programa em partes, o que deixa

mais organizado e fácil de manipular. Além de não precisar das funções implementadas ao compilar, somente o programa principal e o TAD. Vamos começar a explicar pelo as implementações das funções, pois é onde o código realmente funciona.

4.1 O funcEscal.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <String.h>
5  #include <locale.h>
6  #include "tadFunc.h"
7
8  const int quantum = 1;
9  const int velocidade = 800;
10
11 enum tipo{
12     CPU,
13     ES
14 };
15
```

Figura 2 - Bibliotecas utilizadas, constantes e tipo.

As partes dos **includes** se referem as bibliotecas e arquivos utilizados dentro de todo o código, como respectivamente, a biblioteca para incluir as funções de saída, a para alocar dinamicamente as filas utilizadas, a **time** para determinar o tempo de duração real da execução do programa, o **String** para utilizar a função para copiar strings, o **locale** para utilizar caracteres da língua portuguesa e o **tadFunc.h** que é o arquivo onde fica os protótipos das funções. Definimos duas constantes, o quantum que se referem a quantidade de tempo de execução no algoritmo round-robin e a velocidade que determinar a rapidez da execução. Este **enum** tipo se refere ao que tipo é o processo, CPU-bound ou ES(I/O-bound).

```
15
16  struct filaprocessono{
17      int id;
18      int tmpDuracao;
19      int prioridade;
20      Tipo tipo;
21      int cont;
22      int quant;
23
24      FilaProcessoNo* prox;
25  };
26
27  struct filaprocesso{
28      FilaProcessoNo* ini;
29      FilaProcessoNo* fim;
30  };
```

Figura 3 - Estruturas do processo e da fila.

Há dois registros definidos, o que se refere ao processo (**filaprocessono**), que contém o número de identificação do processo, o tempo de duração, a prioridade, o tipo, uma variável de contagem de execução, outra para contagem da fila de espera e um ponteiro para ligar ao próximo processo da fila, respectivamente. O registro **filaprocesso** só contém dois ponteiros que determinar quais os processos iniciais e finais da fila.

```
32  FilaProcesso* fila_cria(){
33      setlocale(LC_ALL, "Portuguese");
34      FilaProcesso* novaFila = (FilaProcesso*) malloc(sizeof(FilaProcesso));
35
36      if (novaFila == NULL) {
37          printf("Erro Fatal: Falha Alocação de memória.\nFinalizar.\n");
38          exit(1);
39      };
40
41      novaFila->ini = novaFila->fim = NULL;
42      return novaFila;
43  }
```

Figura 4 - Função de criação da fila.

Essa função cria um espaço dinâmico para alocar a fila utilizando o **malloc** e depois faz uma verificação se houve espaço suficiente para tal. Depois seta os ponteiros da fila para NULL, pois não tem processos ainda inseridos, e retorna o endereço de alocação.

```c
void fila_insere_inicio(FilaProcesso* fila, int quant, int id, int tempoDuracao, int prioridade, int tipo){
    FilaProcessoNo* novo = (FilaProcessoNo*) malloc (sizeof(FilaProcessoNo));

    novo->id = id;
    novo->tmpDuracao = tempoDuracao;
    novo->prioridade = prioridade;
    novo->tipo = tipo;
    novo->prox = NULL;
    novo->cont = 0;
    novo->quant = quant;

    novo->prox = fila->ini;
    fila->ini = novo;
}
```

Figura 5 - Função para inserção no início da fila.

Essa função aloca espaço dinamicamente para o processo criado com seus argumentos, inserindo ao começo da fila e depois atualiza os ponteiros da fila e do seu próprio ponteiro para apontar para o próximo.

```c
void fila_insere_final(FilaProcesso* fila, int quant, int id, int tempoDuracao, int prioridade, int tipo){
    FilaProcessoNo* novo = (FilaProcessoNo*) malloc(sizeof(FilaProcessoNo));

    novo->id = id;
    novo->tmpDuracao = tempoDuracao;
    novo->prioridade = prioridade;
    novo->tipo = tipo;
    novo->prox = NULL;
    novo->cont = 0;
    novo->quant = quant;


    if(fila->fim != NULL){
        fila->fim->prox = novo;
    }else{
        fila->ini = novo;
    }

    fila->fim = novo;
}
```

Figura 6 - Função para inserção no fim da fila.

Função com o mesmo objetivo da anterior, com uma diferença que insere no final da fila. As condicionais são para verificação dos ponteiros.

```
81  FilaProcessoNo* fila_retira(FilaProcesso* fila){
82      FilaProcessoNo* aux = fila->ini;
83
84      fila->ini = aux->prox;
85      if(fila->ini == NULL){
86          fila->fim = NULL;
87      }
88
89      return aux;
90  }
91
```

Figura 7 - Função para retirar do início da fila.

Função responsável por retirar somente um processo do inicio da fila, para tal, atualiza os ponteiros do processo para NULL, pois não está apontando para nenhum outro processo e retorna o endereço do mesmo.

```
92  void fila_libera(FilaProcesso* fila){
93      FilaProcessoNo* aux = fila->ini;
94
95      while(aux != NULL){
96          FilaProcessoNo* t = aux->prox;
97          free(aux);
98          aux = t;
99      }
100
101     free(fila);
102 }
103
```

Figura 8 - Função para desalocar os processos e a fila.

Função que libera todo o espaço da fila, utilizando para isso um nó auxiliar que percorrer cada processo, liberando seu espaço e ao final o espaço da própria fila.

```
189
190 void fila_imprime(FilaProcesso* fila){
191     FilaProcessoNo *aux;
192
193     for(aux = fila->ini; aux != NULL; aux = aux->prox){
194         printf("%d\t", aux->id);
195     }
196 }
```

Figura 9 - Função para imprimir os identificadores.

Como o nome diz, imprime a fila recebida, usando um nó que percorre cada processo – porém, essa função imprime somente a identificação do processo.

```
198 FilaProcessoNo* retiraPorId(FilaProcesso* fila, int id){
199     FilaProcessoNo* ant = NULL;
200     FilaProcessoNo* p = fila->ini;
201 
202     while(p != NULL && p->id != id){
203         ant = p;
204         p = p->prox;
205     }
206 
207     if(p != NULL){
208         if(ant == NULL){
209             fila->ini = p->prox;
210         } else {
211             ant->prox = p->prox;
212         }
213     }
214     p->prox = NULL;
215     return p;
216 }
```

Figura 10 - Função que retira um processo da fila por id.

Faz o mesmo da função **retira**, mas diferentemente, esta função percorre (por isso o **while**) uma fila recebida com seu id e identifica qual processo tem o **id** procurado, se encontrado (a condicional **if**), retira esse processo e atualiza seus ponteiros, assim como o da fila. Ao final, retorna o endereço do processo.

```
218 void listaProcessos(FilaProcesso* fila){
219     FilaProcessoNo* aux;
220     char tipo[20];
221 
222     printf("----------------- Listagem de Processos -----------------\n\n");
223     for(aux = fila->ini; aux != NULL; aux = aux->prox){
224         if(aux->tipo == 0){
225             strcpy(tipo, "CPU-Bound");
226         } else{
227             strcpy(tipo, "I/O-Bound");
228         }
229         printf("Processo: %d\t Prioridade: %d\t Tempo de duracao: %ds\t Tipo: %s\n", aux->id, aux->prioridade, a
230     }
231     printf("\n\n");
232 }
```

Figura 11 - Função para listar os processos na fila.

Utilizada para fazer uma listagem inicial dos processos inseridos, o fundamento é igual a função **imprime**, mudando somente a parte de verificação do tipo e inserindo a um vetor de string para a impressão, pois o tipo é determinado por 0 (CPU-bound) e 1 (I/O-bound) e queremos apresentar como uma string para o usuário.

```
234 FilaProcessoNo* maiorPrioridade(FilaProcesso* fila){
235     FilaProcessoNo* aux;
236     FilaProcessoNo* noPrio;
237     int prioridade = -1;
238 
239     for(aux = fila->ini; aux != NULL; aux = aux->prox){
240         if(aux->prioridade > prioridade){
241             prioridade = aux->prioridade;
242             noPrio = aux;
243         }
244     }
245 
246     return noPrio;
247 }
```

Figura 12 - Função que procura o processo com a maior prioridade.

Esta função percorre toda a fila recebida, procurando qual o processo de maior prioridade. Para tal, ela atualiza sempre a variável de prioridade para a maior encontrada até o momento, por isso o uso do **if**. Quando terminar de percorrer toda a fila, ela retorna o endereço do processo que contém a maior prioridade (**noPrio**).

```
249 void adicionaNoDe(FilaProcesso* fila, FilaProcessoNo* no){
250     FilaProcessoNo* novo = (FilaProcessoNo*) malloc(sizeof(FilaProcessoNo));
251     novo = no;
252 
253     if(fila->fim != NULL){
254         fila->fim->prox = novo;
255     }else{
256         fila->ini = novo;
257     }
258     fila->fim = novo;
259 }
```

Figura 13 - Função que adiciona um processo criado a uma fila.

Essa função faz o mesmo das funções de inserção para o inicio e o final da fila, porém ela necessita dos argumentos do processo, pois somente utiliza o endereço, depois, atualiza os ponteiros da fila e do processo afim de inserir no final da fila. Esta função foi utilizada para não tornar dificultoso a inserção para a fila ao passar todos os valores do processo.

```c
int verificaExitenciaDoProcesso(FilaProcesso* fila, int id){
    FilaProcessoNo* aux;

    for(aux = fila->ini; aux != NULL; aux = aux->prox){
        if(id == aux->id){
            return 1;
        }
    }
    return 0;
}
```

Figura 14 - Função que verifica se existe um processo na fila.

Como o nome sugere esta função percorre a fila recebida, verificando se a identificação do processo é a mesma que recebida do argumento; caso seja a mesma retorna 1, se não for, retorna 0.

```c
void imprimeResultadoFinal(FilaProcesso* fila){
    setlocale(LC_ALL, "Portuguese");
    FilaProcessoNo* aux;

    printf("\n\n------------------- Resultado dos Processos -------------------");
    for(aux = fila->ini; aux != NULL; aux = aux->prox){
        printf("\nProcesso: %d\t Prioridade: %d\t Quantidade de Vezes da CPU: %d\t Duração do Processo: %ds", au
    }

    printf("\n\n-----Produzido por Felipe Coelho e Manoel Malon\n");
}
```

Figura 15 - Função que imprime os resultados dos processos.

Apresenta na tela do usuário uma impressão dos resultados de cada processo, tal como identificação, duração, quantidade de vezes na CPU, ao percorrer a fila e imprime os nomes dos autores. Com todas as funções auxiliares apresentadas, seguiremos para a explicação do código principal de escalonamento de processos, por questões de espaço não será possível inserir toda a função numa imagem só, portanto será dividida por partes:

```
104 void escalonaProcesso(FilaProcesso* pronto){
105     int ciclos = 0;
106     float  tmpTotal = 0;
107 
108     FilaProcessoNo* aux;
109     FilaProcessoNo *atual;
110 
111     clock_t t;
112 
113     FilaProcesso* espera = fila_cria();
114     FilaProcesso* resultado = fila_cria();
115 
116     atual = maiorPrioridade(pronto);
117 
118     listaProcessos(pronto);
119     system("Pause");
120     system("cls");
121 
```

Figura 16 - Função de Escalonamento Parte I.

Como dito anteriormente é a função responsável pela implementação dos algoritmos **FIFO** e **ROUND-ROBIN** – é importante ressaltar que estes algoritmos estão entrelaçados na função. Primeiramente é recebida a fila onde está os processos inseridos, ou seja, a fila de pronto. São inicializadas duas variáveis, uma para contagem de ciclos da CPU e outra para determinar o tempo total de execução do algoritmo. Dois nós auxiliares são utilizados para receber os processos retirados da fila de pronto, por isso o **aux** e o **atual**. **Clock_t t** é uma variável para determinar o tempo de execução do programa. Depois duas outras filas são criadas, a de espera (onde o processo ficará aguardando o tratador de interrupção) e outra onde os processos finalizados serão armazenados para serem apresentados depois da execução do algoritmo. Para inicializar o algoritmo, é necessário que primeiramente seja retirado o processo de maior prioridade da fila, caso não tenha um processo de maior prioridade, será retirado o primeira da fila. Para apresentar ao usuário, os processos criados, é utilizada a função **listaProcessos**. As funções denominadas **system(“Pause”)** e **system(“cls”)** são para pausar o prompt de comando e apagar os textos do mesmo.

```
122 while(pronto->ini != NULL){
123     t = clock();
124 
125     aux = retiraPorId(pronto, atual->id);
126     tmpTotal = tmpTotal + aux->tmpDuracao;
127 
128     aux->quant++;
129     printf("\nA fila de pronto: \n"); fila_imprime(pronto);
130 
131     Sleep(400);
132     printf("\n\nO processo em execução: %d", aux->id);
133     Sleep(400);
134     printf("\b\b\b %s", "       ");
135 
136     if(ciclos > 10){
137         if(aux->tmpDuracao > quantum){
138             aux->tmpDuracao = aux->tmpDuracao - quantum;
139             if(verificaExitenciaDoProcesso(pronto, aux->id) != 1){
140                 fila_insere_inicio(pronto, aux->quant, aux->id, aux->tmpDuracao, aux->prioridade, aux->tipo);
141             }else{
142                 retiraPorId(pronto, aux->id);
143                 fila_insere_inicio(pronto, aux->quant, aux->id, aux->tmpDuracao, aux->prioridade, aux->tipo);
144             }
145         }
```

Figura 17 - Função de Escalonamento Parte II.

É utilizado um laço de repetição que tem como condição de parada o ponteiro da fila, ou seja, se este ponteiro for nulo, significa que não existe mais processos na fila de pronto e o algoritmo pode ser encerrado. **T = clock( )** é a inicialização do tempo de execução do algoritmo. **Aux** vai receber o valor do processo que foi inicializado como valor de **atual**, depois é somado os tempos de duração de cada processo para exibir ao final a duração da CPU. Com isso, o termo de quantidade do processo é incrementado para alegar que foi já retirado para a execução. Posteriormente é imprimida a fila de pronto já com o processo retirado. A função **Sleep** denota o tempo de execução das linhas de comando, sem ela, tudo seria executado rapidamente, portanto, precisando desta pausa para analisar os processos mudando de estado. Depois é imprimido o processo em execução usando o **printf** e o próximo **printf** contém caracteres de formatação, neste caso, este caractere apaga a quantidade necessário dos caracteres imprimidos anteriormente e substitui pelo o espaço vazio – isto é feito para demonstrar a retirada do processo de execução para pronto. A condicional principal a seguir é para delimitar a execução de ciclos de execução como a proposta do trabalho requer, ou seja, caso entre mais de 10 processos em execução, o algoritmo trabalhará de forma preemptiva. Contudo, ela verifica se o tempo de execução do processo é maior que o quantum, pois assim, o processo atual executará novamente, depois é retirado uma fatia do tempo de duração do processo. Agora entra a função de verificação do processo, pois se o processo não estiver na fila de pronto, ele será inserido; caso contrário, ele será retirado da fila e posteriormente inserido novamente com seus atributos atualizados.

```
148     atual = maiorPrioridade(pronto);
149
150     if(aux->tipo == 1 && aux->cont == 0){
151         aux->cont++;
152         aux->tipo = 0;
153
154         adicionaNoDe(espera, aux);
155
156         fila_insere_inicio(pronto, aux->quant, aux->id, aux->tmpDuracao, aux->prioridade, 0);
157     }
158
159     printf("\n\nEm Espera: \n"); fila_imprime(espera);
160
161     if(espera->ini != NULL){
162         fila_retira(espera);
163     }
164
165     if(verificaExitenciaDoProcesso(resultado, aux->id) != 1){
166         fila_insere_inicio(resultado, aux->quant, aux->id, aux->tmpDuracao, aux->prioridade, aux->tipo);
167     }else{
168         retiraPorId(resultado, aux->id);
169         fila_insere_inicio(resultado, aux->quant, aux->id, aux->tmpDuracao, aux->prioridade, aux->tipo);
170     }
171
```

Figura 18 - Função de Escalonamento Parte III.

Depois da execução, um novo processo será retirado de acordo com a prioridade. O **if** a seguir verifica qual o tipo de processo estive em execução, pois se for 1 (I/O-bound) ele será inserido na fila de pronto usando a função **adicionaNoDe** e depois inserido novamente na fila de pronto com seus atributos atualizados, por isso é importante retirar o novo processo antes da inserção novamente do processo que estava em espera. Tem uma impressão da fila de espera e caso a fila não esteja vazia, é retirado o processo da fila. A próxima condicional é para inserir os processos terminados na fila de resultados, para tal, verifica se o processo já está na fila, se não estiver, é inserido na fila de resultado, caso contrário, é retirado pelo o seu **id** e inserido novamente com seus atributos atualizados.

```
172         ciclos++;
173         Sleep(velocidade);
174         system("cls");
175     }
176     t = clock() - t;
177     printf("Tempo Real de execucao: %lfms", ((double)t)/((CLOCKS_PER_SEC/1000)));
178     printf("\nCiclos da CPU: %d", ciclos);
179     printf("\nTempo total da CPU: %.2fs", tmpTotal);
180     imprimeResultadoFinal(resultado);
181     fila_libera(resultado);
182     fila_libera(espera);
183     fila_libera(pronto);
184     printf("\n");
185     system("Pause");
186 }
187
```

Figura 19 - Função de Escalonamento Parte IV.

A variável de ciclos é incrementada, tem a função para dar uma pausa no prompt e apagar os textos apresentados no prompt. Depois da execução do laço de repetição, ou seja, depois que não tiver mais processos na fila de pronto, o tempo será calculado

pela subtração do início da execução pelo o final da execução. As variáveis da função são imprimidas como o tempo real de execução em milissegundos, a quantidade de ciclos de CPU e o tempo total de utilização da CPU. A função de impressão dos resultados é chamada passando como parâmetro a fila que está contido os processos finalizados. Finalmente, as filas utilizadas são desalocadas para economizar a memória.

4.2 O tadFunc.h

```
1
2  typedef enum tipo Tipo;
3
4  typedef struct filaprocessono FilaProcessoNo;
5
6  typedef struct filaprocesso FilaProcesso;
7
8  FilaProcesso* fila_cria();
9
10 FilaProcessoNo* fila_retira(FilaProcesso* fila);
11
12 void fila_libera(FilaProcesso* fila);
13
14 void fila_imprime(FilaProcesso* fila);
15
16 void escalonaProcesso(FilaProcesso* pronto);
17
18 void listaProcessos(FilaProcesso* fila);
19
20 FilaProcessoNo* retiraPorId(FilaProcesso* fila, int id);
21
22 void fila_insere_inicio(FilaProcesso* f, int quant, int id, int tempoDuracao, int prioridade, int tipo);
23
24 void fila_insere_final(FilaProcesso* fila, int quant, int id, int tempoDuracao, int prioridade, int tipo);
```

Figura 20 - TAD parte I.

Primeiramente é criado o tipo do processos e as estruturas tanto do processo, quanto da fila. Depois, é simplesmente colocados os protótipos das funções já descritas.

```
24 void fila_insere_final(FilaProcesso* fila, int quant, int id, int tempoDuracao, int prioridade, int tipo);
25
26 FilaProcessoNo* maiorPrioridade(FilaProcesso* fila);
27
28 void adicionaNoDe(FilaProcesso* fila, FilaProcessoNo* no);
29
30 int verificaExitenciaDoProcesso(FilaProcesso* fila, int id);
31
32 void imprimeResultadoFinal(FilaProcesso* fila);
```

Figura 21 - TAD parte II.

É importante salientar que não importa a ordem das assinaturas das funções, pois é identificado automaticamente as funções.

4.3 O main.c

Para facilitar a leitura o código será divido em duas partes:

```c
#include <stdio.h>
#include <stdlib.h>
#include "tadFunc.h"
#include <time.h>
#include <locale.h>
#define QUANT 15

int main(){
    setlocale(LC_ALL, "Portuguese");
    srand(time(NULL));

    FilaProcesso* filaPronto1 = fila_cria();
    FilaProcesso* filaPronto2 = fila_cria();
    int contador;

    printf("PROCESSOS COM PRIORIDADE 0: \n\n");
    for(contador = 0; contador < 7; contador++){
        fila_insere_final(filaPronto1, 0, rand() % 1000, 1+rand() % 5, 0, rand() % 2);
    }
    escalonaProcesso(filaPronto1);
    system("cls");

```

Figura 22 - Main parte I.

As bibliotecas usadas são a mesmas do **funcEscal.c** pelas mesmas razões já explicitadas. O **tadFunc.h** é inserido para realizar a ligação com as funções implementas e além é definida uma constante **QUANT** com valor 15 para limitar o laço de repetição da segunda parte do algoritmo. Os valores dos processos são gerados aleatoriamente, por isso o uso da função **srand** inicializada. Depois é criada duas filas de armazenado de processsos para execução, a primeira para executar sem a prioridade e sem o uso de ciclos de execução. Fizemos duas execução do algoritmo, pois como os valores são gerados aleatoriamente, com bastante certeza, não teriamos uma sequência total de processos com prioridade iguais, o que faria com que o código não executasse da forma não preemptiva, a solução portanto é fazer duas execuções com prioridades iguais e outra com variás. Sendo assim, o primeiro laço insere processos na fila de pronto com tempo de duração, **id** e prioridade aleatoriamente e prioridade e contagem iguais a 0 – depois é chamada a função de escalonamento de processos, o que executará de forma não preemptiva, seguindo o modelo **FIFO**.

```c
    printf("\nPROCESSOS COM VÁRIAS PRIORIDADES: \n\n");
    for(contador = 0; contador < QUANT; contador++){
        fila_insere_final(filaPronto2, 0, rand() % 1000, 1+rand() % 5, rand() % 5, rand() % 2);
    }
    escalonaProcesso(filaPronto2);

    return 0;
}
```

Figura 23 - Main parte II.

Neste laço é inserido novamente na fila de pronto, como no laço anterior, porém com a diferença que a prioridade é gerada aleatoriamente, posteriormente é chamada a função de escalonamento, porém executará de forma preemptiva usando o **round-robin** devido as prioridades.

4.4 Execução do Programa

Vejamos a execução do programa para determinar a funcionamento do mesmo:

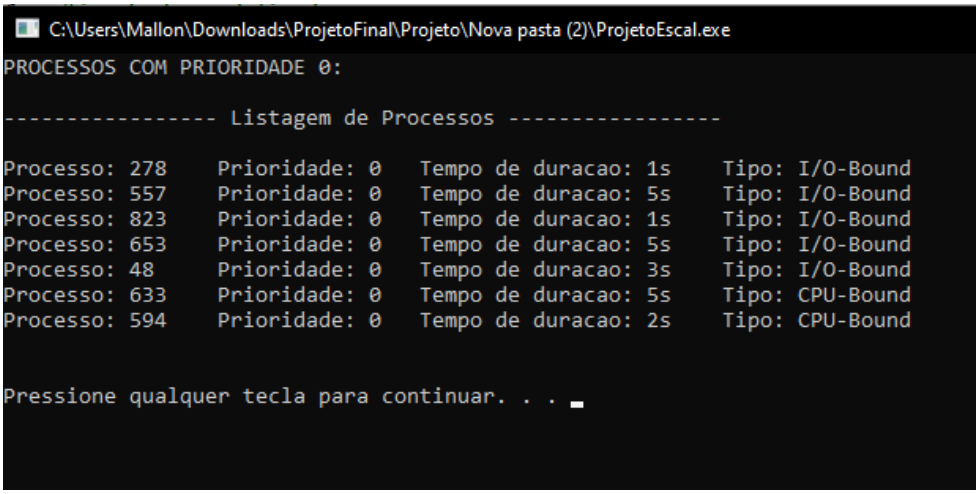

Figura 24 - Listagem dos processos da 1ª execução.

Primeiramente é apresentado, a listagem dos processos pela fila de pronto com seus atributos. Como explicado no código do **main** é chamada a função de listagem para apresentar essa tela e as prioridades nessa execução é 0 para deixar o algoritmo executar de forma não preemptiva. Sendo assim ao clicar qualquer tecla o programa é continuado na sua execução.

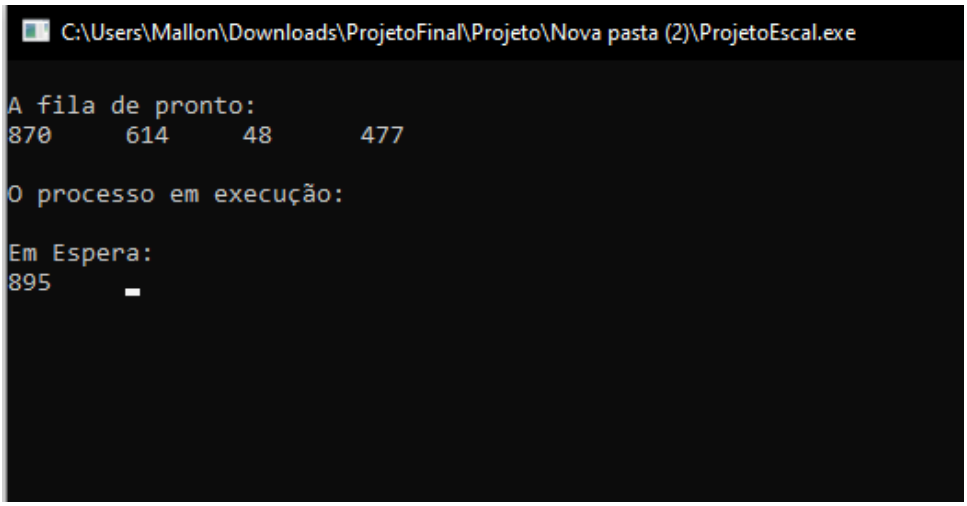

Figura 25 - Estados dos processos do não-preemptivo.

São apresentados 3 estados dos processos. Primeiramente a **fila de pronto**, **o processo em execução**, e **em espera**. O primeiro processo da fila é retirado e vai para execução, caso seja do tipo CPU-Bound ele executa, é finalizado e vai para a fila dos resultados, mas se for do tipo I/O-Bound, ele vai para a fila de espera e volta depois para a fila de **pronto** no final da fila, posteriormente é executado e finalizado seguindo o caminho de um processo do tipo CPU-Bound. Contudo, esse processo vai se repetindo até finalizar todos os processos.

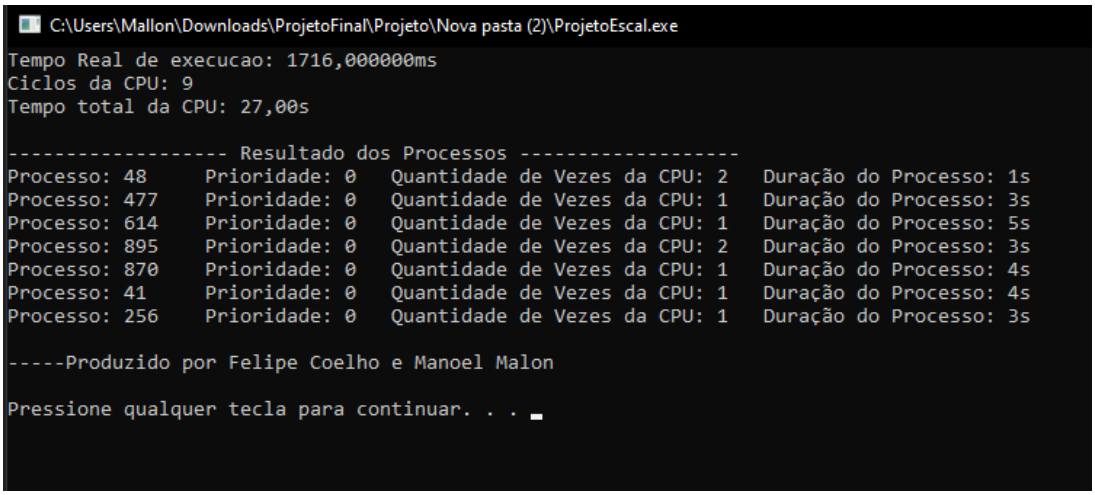

Figura 26 - Resultados dos processos do não-preemptivo.

Ao terminar a execução é apresentado as variáveis da execução e a listagem dos processos como **id**, **prioridade**, **quantidade de vezes** e **duração do processo**. Pressionamos qualquer tecla para continuar.

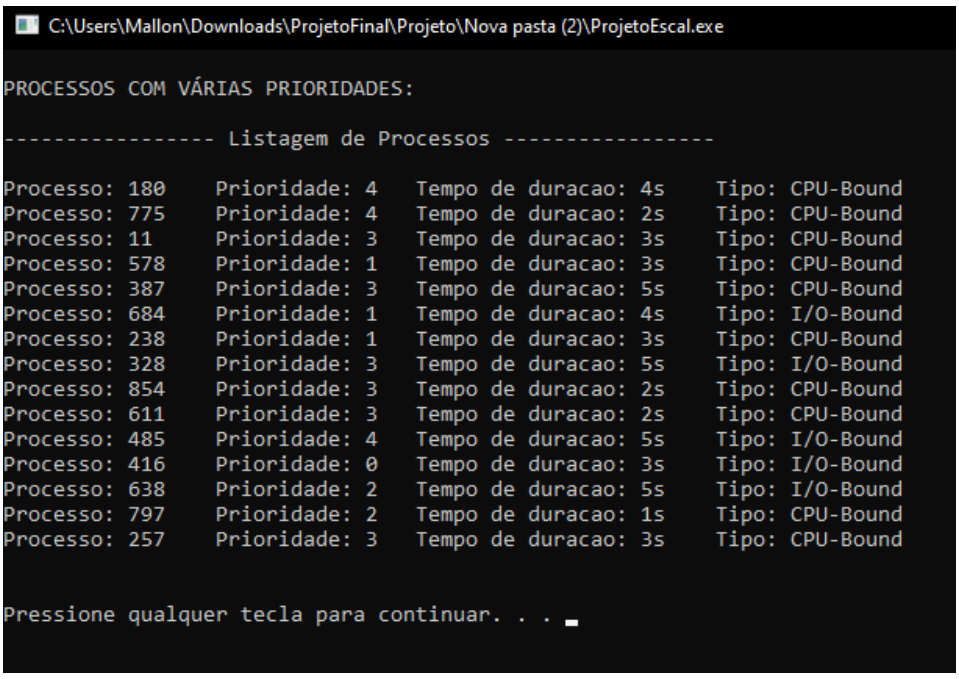

Figura 27 - Listagem dos processos da 2ª execução.

O segundo tipo de execução é dado com processos com várias prioridades, para trabalhar de forma preemptiva. Entretanto é apresentado a listagem com todos os processos gerados aleatoriamente. Ao clicar qualquer tecla, é executado o algoritmo.

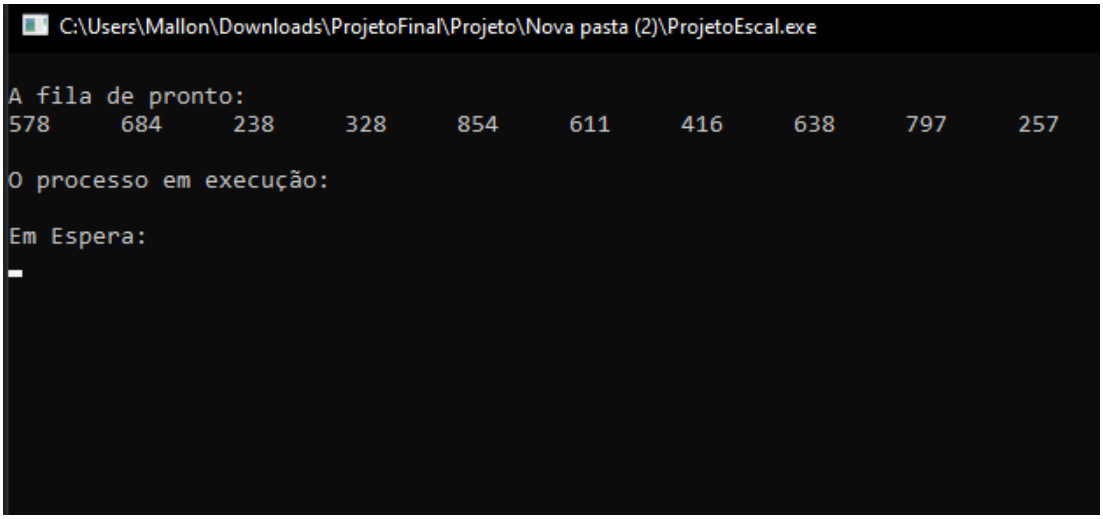

Figura 28 - Estados dos processos do preemptivo.

É igual a tela anterior, porém neste algoritmo não terá uma sequência como o **FIFO**, mas sim serão executados de forma preemptiva através do algoritmo **round-robin**. Um processo vai ser executado, caso seja do tipo CPU-Bound, ele retornará a fila de pronto se o tempo de duração dele for maior que o **quantum**, caso seja do tipo I/O-Bound, ele vai para a fila de **espera** uma vez e retornará para a fila de **pronto** depois se comportando como se fosse do tipo CPU-Bound até terminar seu **tempo de duração**.

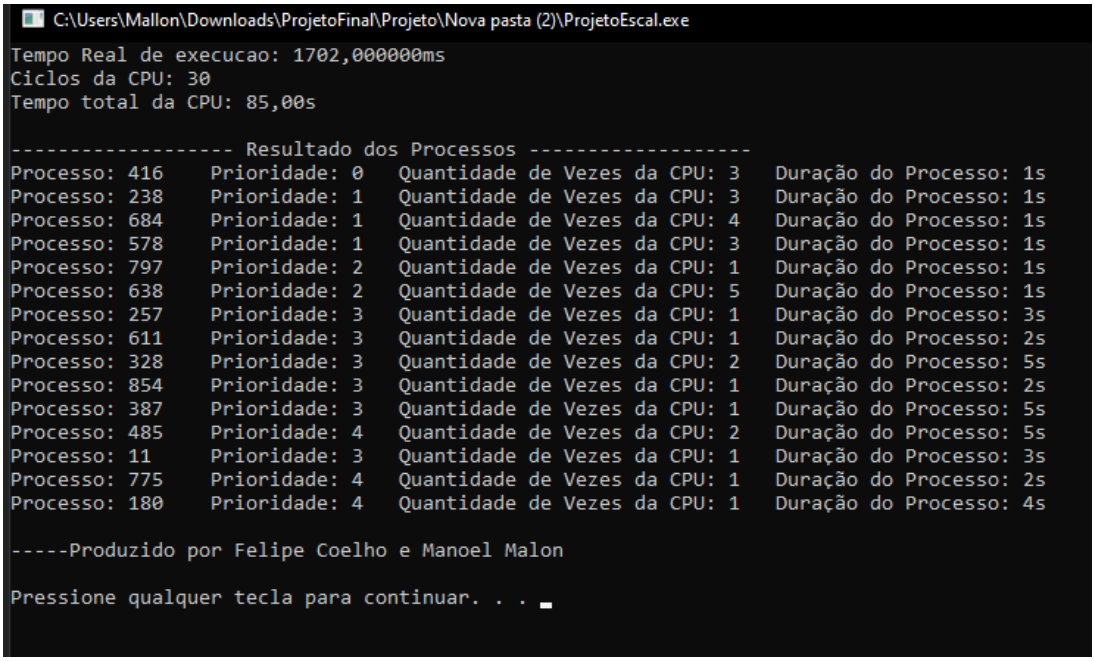

Figura 29 - Resultados dos processos do não-preemptivo.

Quando terminar todas as execuções, é apresentado os resultados como na primeira execução. Uma diferença importante entre as duas execuções é que na primeira, a quantidade de vezes máxima na CPU é 2 devido aos processos de I/O, enquanto que nesta segunda execução, a quantidade de vezes depende do tipo do processo e do tempo de duração do processo devido ao rodizio dos processos.

5. CONCLUSÃO

Em suma, pode-se notar um escalonador rotando e tudo que é preciso para que ele funcione corretamente, o desafio inicial foi a escolha dos escalonadores que se complementem, sendo eles o FIFO e o Round-robin para torna todo o programa mais eficiente, depois disso foi pensando em como o programa deveria executar cada função e qual seria a melhor forma do escalonador preemptivo escolher o programa por prioridade caso haja conflitos de prioridade. Outro ponto sério o ciclo do escalonamento sendo implementado, onde o processo passa duas na fila sé ele é do tipo CPU-Bound para sua execução e como séria demonstrando no programa. E por fim a implementação do programa foi de grande aprendizagem para saber como funciona na pratica um escalonador e todas suas dificuldade e praticidade em processos.