Simulador de um Escalonador de Processos

Desenvolvido em Java

Thiago A. Nakao França
UTFPR – Universidade Tecnológica Federal do Paraná
Campo Mourão,PR
nakaosensei@gmail.com

Lucas Rafael
UTFPR – Universidade Tecnológica Federal do Paraná
Campo Mourão,PR
rafa_lucasrafael@hotmail.com

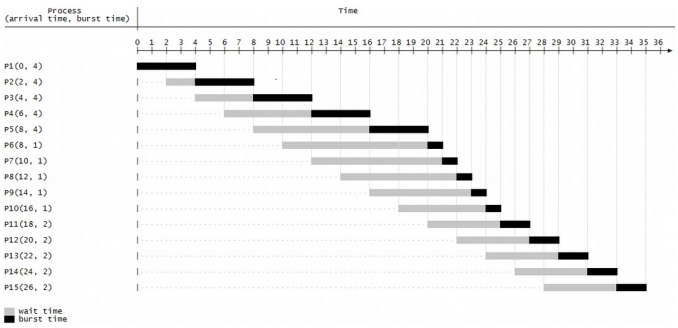
I. Introdução

Neste trabalho é apresentado um simulador de um escalonador de processos feito na linguagem de programação Java, foi desenvolvido apenas com o uso de bibliotecas nativas e sem reaproveitamento de códigos externos, o simulador pode escalonar os processos de entrada usando as políticas Round Robin, Shortest Job First, First Come First Served ou Múltiplas Filas. É produto de uma atividade prática supervisionada da UTFPR de Campo Mourão.

II. Os clássicos algoritmos de escalonamento

A. *First Come First Served(FCFS)*

First Come First Served(FIFO), se trata de um método não-preemptivo, e tem como principal vantagem sua extrema simplicidade, pois seu método de escalonar consiste em simplismente atender as tarefas em sequência, seguindo por ordem a lista de prontos.



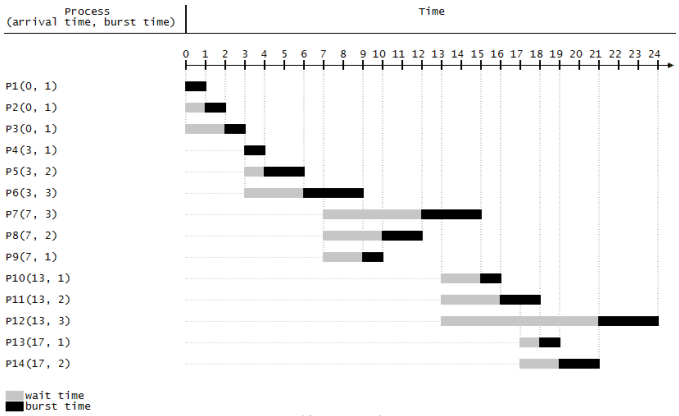
Fifura 2.1 - Fcfs

B. *Shortest Job First(SJF)*

Shortest Job First (SJF), assim como First Come First Served pode se tratar de um método não-preemptivo, o método SJF funciona com uma simples ideia, o processo com o menor tempo tera maior prioridade, sendo assim executado primeiro, diminuindo assim o tempo de espera para cada conjunto de processos a serem executados.

Shortest Job First possui uma variação chamada de Shortest-Remaining-Time-First (SRTF), essa que por sua vez se trata de um método preemptivo, se um processo chega na fila de prontos com um tempo menor do que o tempo que resta para o processo que esta executando terminar, o escalonamento é realizado.

Como o Shortest Job First sempre irá escolher um processo com o menor tempo para ser executado, pode ser que um processo mais demorado pode ser que demore muito, ou ainda, nunca chegue a ser executado, pois novos processos com tempos menores podem ser sempre priorizados, deixando assim o processo com um tempo maior sempre atraz, em prioridade, dos novos processos com um tempo menor.



Fifura 2.2 *Sjf*

C. *Random*

O algoritimo de escalonamento Random, simplesmente escolhe um processo, da lista de prontos, aleatório para ser execudado, no bloqueio do processo, o método simplesmente é enviado para a lista de bloqueados e ao ser desbloqueado é enviado para a lista de prontos.

O Random pode ser facilmente implementado e também é de facil entendimento, porém pode ser desvantajoso, ao escalonar um processo que precisa de muito tempo para terminar sua tarefa, deixando assim processos, que poderiam estar executados na fila. Além desta desvantajem, ainda existe a possibilidade de um processo nunca ser selecionado, se existir mais de um processo na fila de prontos.

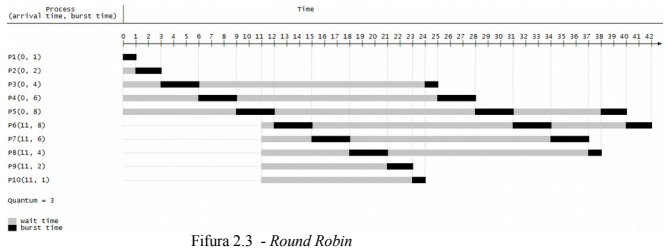
D. *Round-Robin*

A política de escalonamento Round-Robin, que é uma das politicas mais simples, preemptivo, eficiente e usada nos dias

de hoje, tem uma simples idéia de aplicar um tempo limite para o atual processo executando no momento, este tempo é chamado de quantum.

O algoritimo funciona da seguinte maneira: existe uma fila de prontos, que nela estão todos os processos que estão prontos para serem executados, sempre que um processo finaliza, ou o tempo do seu quantum termina, é verificado se a lista de prontos possui algum processo, se sim é escolhido o primeiro processo da fila de prontos, o tira da lista de prontos, estabelece seu quantum e o mesmo é posto para executar, se ouver bloqueio o processo é enviado para a lista de bloqueados e se o processo for desbloqueado ele é enviado para a lista de prontos novamente e espera até ser escalonado novamente.

Em geral a política de Round-Robin é a mais utilizade entre as outras, e também sua simplicidade o torna o mais utilizavel. Não possui o problema de processos que nunca serão executados, pois por mais demorado ou pegueno que seja o tempo do processo ele sera executado, e de forma justa.Uma de suas desvantagens está ligada ao fato de ajustar o tempo do quantum, que dependendodo tempo que foi lhe dado, pode tornar o escalonamento estremamente ineficiente.

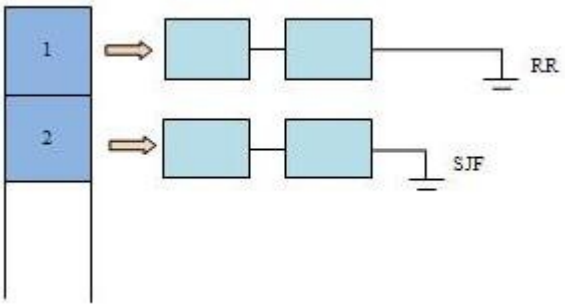


Fifura 2.3 - *Round Robin*

E. Múltiplas Filas

Esta politica possui varias classes de prioridades onde cada classes pussui quanta diferentes. Se trata de um algoritimo preemptivo, e é quase tão antigo quanto Round-Robin.

O algoritimo atende às filas com mais prioridade, e cada uma dessas filas pode utilizar um politica de escalonamento diferente, como o algoritimo é preemptivo, se um processo com uma prioridade maior que o que está executando atualmente possui ele é escalonado e posto para executar.



Fifura 2.4 – Fila de Prioridades

III. A DIVISÃO DE TAREFAS

A princípio, tivemos dificuldades para atuar em equipe num trabalho que requer bastante programação como este, pois apesar da facilidade de modularização(Existem diversas politicas no escalonador, poderiamos apenas delimitar quais politicas cada um deveria implementar), optamos por desenvolver o escalonador desde as raízes, sem reaproveitar o código passado pelo professor Mestre Juliano Foleiss, tal decisão nos levou a técnica *Pair Programming,* que funcionou muito bem para este desafio. Diversas vezes um programador ajudou o outro a ver que a implementação por certos caminhos poderia levar a um resultado estranho ou ruim, poupando muito tempo que ambos teriam gasto em modo solo, outra vantagem foi que os desenvolvedores se mantiveram totalmente alertas de tudo o que era criado sem "se perder", já que as trocas constantes de papéis de desenvolvedor e observador acabam forçando uma contextualização de ambas as partes.

Quanto ao relatorio, o acadêmico Lucas Rafael foi o responsavel por esta pelas seções I,II e III, enquanto da IV até o final deste documento foram escritas por Thiago Alexandre Nakao França.

IV. A IMPLEMENTAÇÃO

Como já dito anteriormente, o escalonador foi implementado sem o auxilio do código em C do professor, pois acreditavamos que com a nossa experiencia em Orientação a Objetos seriamos capazes de criar um simulador genérico do zero de forma bem feita, mesmo que isso nos tenha acarretado muito mais trabalho do que o requisitado para os demais grupos, onde o desafio estava em compreender o código e terminá-lo.

V. PACOTES

Primeiramente, irei explicar a estrutura de pacotes do projeto, existem quatro pacotes principais, sendo estes:

A. br.com.model

Aqui estão as estruturas que definem os objetos do sistema, o pacote possui quatro classes, chamadas Nivel, Evento, Experimento e Processo, todas estas classes estão aqui por apresentarem um papel mais estrutural na aplicação.

Dentro do pacote br.com.model, existe outro pacote, chamado politics, que possui outras seis classes, sendo estas Politica, Fcfs, FilaPrioridades, Random, RoundRobin e Sjf.

Neste pacote se encontram todas as possíveis politicas que o escalonador pode adotar, mais detalhes de sua implementação serão vistos mais a frente.

C. br.com.controller

Neste pacote encontra-se a classe main, que é a responsável pela execução do software , a classe Arquivo, que possui métodos para leitura e escrita de arquivos, a classe ProcessController, que é a responsável por carregar todos os processos a partir de uma entrada e executar alguns métodos, e por fim a classe Escalonador que é a responsável for

efetivamente escalonar os processos a partir usando uma Politica de escalonamento.

D. br.com.files

Neste pacote encontram-se os arquivos que foram usados para os testes realizados no simuador.

VI. Classes

A. main

```
public class main {
    public static void main(String[] args) {
        Arquivo arq = new Arquivo();
        String in = arq.lerArquivo("src/br/com/files/sjf.exp");
        String split[]=in.split("\n");
        String arquivoProcessos=arq.lerArquivo("src/br/com/files/"+split[1]);
        ProcessController pl = new ProcessController(arquivoProcessos.trim());
        Experimento exp = new Experimento(in,pl);
        String out = exp.getEscalonador().run();
        arq.escreverArquivo(exp.getCaminhoArqSaida(), out);
    }
}
```

O escalonador tem como ponto de partida a classe main, ela é quem obtem o caminho do arquivo de experimento e chama a classe responsável por carregar os dados necessários(ProcessController), além de também invocar a classe responsável por processar o escalonamento(Escalonador da classe Experimento), também é ela quem importa a classe Arquivo(Responsável por leitura e escrita de arquivos).

Figura A, a Classe main

B. ProcessController

A classe ProcessController é a primeira classe realmente importante para o simulador(A classe arquivo aparece antes, mas ela somente lê e escreve em arquivos), ela recebe como entrada uma String contendo todas as linhas do arquivo de processos e os carrega, para dizer o que realmente significa esse "carregar" precisamos saber que atributos a classe possui.

A classe contém quatro atributos, sendo estes o numProcessos, prioridadeMax, prioridadeMin, e uma ArrayList de Processo, que é outra classe, que será explicada na sequência.

```
public class ProcessController {
    private int numProcessos;
    private int prioridadeMinima;
    private int prioridadeMaxima;
    private List<Processo> processos;

    public ProcessController(String arq){
        carregarEstruturas(arq);
    }
```

Figura B, Atributos Classe ProcessController

B1. Processo

É responsável por carregar seus atributos e eventos, cada processo possui uma lista de Evento, que e outra classe mas bem pequena, basicamente contem os atributos nome(CRIACAO,BLOQUEIO,DESBLOQUEIO, TERMINO), tempo de ocorrencia e id do owner(processo dono deste Evento).

```
public class Evento{
    protected int tempoOcorrencia;
    protected int ownerId;
    protected String nome;


    public Evento(String nome,int tempoOcorrencia,int ownerId){
        this.tempoOcorrencia=tempoOcorrencia;
        this.ownerId=ownerId;
        this.nome=nome;

    }
```

Figura B1a, Atributos Classe Evento

Alem desta parte estrutural, a classe Processo mantém um atributo tempoOcupouCpu, que sera utilizado na execução do escalonador para setar quantos tempos o processo ja ocupou na CPU, outro atributo importante é o eventoSelecionado, que indica o proximo evento a ser executado pela CPU.

```
public class Processo {
    private int pid;
    private int prioridade;
    private int qtEventos;
    private Evento inicio;
    private Evento fim;
    private List<Evento> eventos;
    private int eventoSelecionado;
    public int tempoOcupouCpu;

    public Processo(String pString){
        carregarComTempoRelativoAoRelogio(pString);
    }
```

Figura B1b, Atributos Classe Processo

Com isso, a classe ProcessController foi explanada, a visão geral dela deve ser de uma classe que recebe como entrada os processos em forma de texto e carrega em estruturas de dados, além de possuir atributos de controle bastante importantes para a classe Escalonador.

C. Experimento

A classe de experimento contém atributos referentes ao experimento(nome, arquivoProcessos,arquivoSaida) e uma outra classe chamada Escalonador, a classe Experimento recebe uma String contendo as linhas do arquivo de experimento e carrega em si, além disto a classe pede como parametro uma classe ProcessController para poder repassar para a classe Escalonador.

Também é importante dizer que e nesta classe que se define qual a Politica de escalonamento que será repassada para a classe Escalonador, entender a classe Politica é muito importante para compreender como o escalonador funciona de forma generica, abaixo, a descrição da classe Politica.

C1. Politica

Politica é uma classe abstrata que contém dois atributos, nome e quantum, além de um método abstrato, abaixo a classe:

```
public abstract class Politica {
    protected String nome;
    protected int quantum;

    public Politica(String nome,int quantum){
        this.nome=nome;
        this.quantum=quantum;
    }

    public abstract Processo doSelectionPolitic(List<Processo> prontos);
```

Figura C1b, A Classe Politica

O metodo doSelectionPolitic(List<Processo> prontos) deve ser implementado por cada filho que herdar de Politica, é um

método usado pela politica do escalonador e varia a implementação de filho para filho, o metodo deve retornar o processo que deve sair da lista de prontos e ir para o estado executando, além disso o método deve também remover o processo da lista de prontos.

Ademais, a classe politica tem os getters e setters dos dois primeiros atributos.

C1a. Fcfs

A classe Fcfs herda de Politica, ou seja ela e uma Politica e deve escrever todos os metodos abstratos definidos, no caso, apenas o metodo doSelectionPolitic, no caso do Fcfs, remove-se o primeiro elemento da lista de prontos e o retorna.(O processo retornado será o executado do Escalonador).

```
@Override
public Processo doSelectionPolitic(List<Processo> prontos) {
    if(prontos.isEmpty()){
        return null;
    }
    Processo executando;
    executando=prontos.get(0);
    prontos.remove(0);
    return executando;
}
```

Figura C1a, doSelctionPolitic do Fcfs

C1b. Sjf

A classe Sjf também herda de Politica, a sua sobrescrita do método doSelectionPolitic encontra o processo da lista de prontos que tem o menor tempo de execução faltante, para descobrir o tempo de execução de cada processo foi muito simples, como todo processo tem uma variavel chamada tempoOcupouCpu(que indica quantos tempos na CPU esse processo ja ocupou em um determinado tempo do escalonador), bastou fazer TERMINO-tempoOcupouCpu para descobrir o tempo de execução que aquele dado processo ainda tinha, então comparando os tempos de execução de cada processo na lista de prontos bastava escolher o processo com o menor tempo de execução restante.

```
@Override
public Processo doSelectionPolitic(List<Processo> prontos) {
    if(prontos.isEmpty()){
        return null;
    }
    Processo menor=prontos.get(0);
    Processo executando;
    for(Processo p:prontos){
        if(!menor.isShortest(p)){
            menor = p;
        }
    }
    executando=menor;
    prontos.remove(menor);
    return executando;
}
```

Figura C1b, doSelctionPolitic do Sjf

C1c. Random

Mais um filho da classe Politica, sua sobrescrita de método retorna um Processo aleatório da lista de prontos para entrar em execução, assim como todos as demais politicas ele o remove da lista prontos também.

```
@Override
public Processo doSelectionPolitic(List<Processo> prontos) {
    if(prontos.isEmpty()){
        return null;
    }
    Processo executando;
    java.util.Random r = new java.util.Random();
    int next=r.nextInt(prontos.size());
    executando=prontos.remove(next);
    return executando;
}
```

Figura C1c, doSelctionPolitic do Random

C1d. RoundRobin

Por mais simples que pareça, a sobrescrita de metodo do doSelectionPolitic(List<Processo> prontos) do RoundRobin é a mesma do Fcfs, o que muda é que no RoundRobin o quantum normalmente é diferente de 0 e o escalonador usa o quantum da politica para gerar os eventos de QUANTUM_EX.

C1e. FilaPrioridades

Sem dúvidas a mais complexa de todas as politicas, antes de explicar sua sobrescrita do método abstrato doSelectionPolitic é preciso conhecer a estrutura de dados Nivel presente nesta politica, a classe Nivel possui dois atributos, sendo estes:

```
public class Nivel {
    private int prioridade;
    private Politica politica;
```

Figura C1e, Atributos Classe Nivel

Esta politica possui uma lista de niveis,estrutura essa que é carregada no momento da leitura do arquivo de experimento.

Partindo para a explanação da sobrescrita do metodo doSelectionPolitic(List<Processo> prontos) , primeiramente agrupam-se todos os Processos da lista de prontos com maior prioridade(a maior prioridade é sempre a numericamente menor) em uma nova lista, chamada newProntos, então para cada nivel da lista de niveis, procura-se o nivel com a mesma prioridade dos elementos da nova lista, tendo o encontrando, a classe FilaPrioridades recebe o quantum da Politica daquele nivel e invoca o metodo doSelectionPolitic(List<Processo> prontos) da Politica do nivel, passando como parametro a lista newProntos, mas como se trata de Java e a lista passada não tem a mesma referencia da lista prontos original, depois da execução do doSelectionPolitic da Politica do nivel é necessario remover o elemento selecionado da lista prontos na chamada de FilaPrioridades, abaixo o código:

```
@Override
public Processo doSelectionPolitic(List<Processo> prontos) {
    if(prontos.isEmpty()){
        return null;
    }
    List<Processo> newProntos = getProcessesWithHigherPriority(prontos);
    for(Nivel n:niveis){
        if(n.getPrioridade()==newProntos.get(0).getPrioridade()){
            this.quantum=n.getPolitica().getQuantum();
            Processo exec= n.getPolitica().doSelectionPolitic(newProntos);
            prontos.remove(exec);
            return exec;
        }
    }
    return null;
}
```

Figura C1e, doSelctionPolitic do FilaPrioridades

Por fim, e preciso dizer que a classe Experimento delimita qual sera a politica do Escalonador pelo nome obtido no arquivo de experimento, se for rr por exemplo, o atributo

Politica do Escalonador receberá um new RoundRobin(args), e assim por diante para as demais politicas.

C2. Escalonador

E finalmente vamos a classe que realmente executa a simulação do escalonamento de processos, o Escalonador tem um total de quatorze atributos, sendo estes:

```
public class Escalonador {
    private Politica politica;
    private ProcessController pc;
    private List<Processo> prontos;
    private Processo executando;
    private List<Processo> bloqueados;
    private int tempo;
    private int qtChaveamentos;
    private double tme;
    private double tmr;
    private double vazao;
    private List<Processo> sequenciaTermino;
    private List<Evento> diagramaEventos;
    private int quantum;
    private int unidadesMilenio;
```

Figura C2a, atributos do Escalonador

Abaixo, temos o metodo construtor da classe, que ja carrega a estrutura inicial do Escalonador, veja:

```
    public Escalonador(ProcessController processos,Politica politica){
        processController=processos;
        prontos=new ArrayList<>();
        bloqueados=new ArrayList<>();
        sequenciaTermino = new ArrayList<>();
        diagramaEventos=new ArrayList<>();
        tempo=0;
        this.politica=politica;
        this.quantum=this.politica.getQuantum();
        this.unidadesMilenio=1;
    }
```

Figura C2b, contrutor do Escalonador

Então, temos que a lista de bloquados/prontos inicia vazia, o relogio inicia em 0, a Politica e o quantum são setadas, e a variavel unidadesMilenio inicia em um, esta variavel será util para calcular a vazão e será melhor explicada posteriormente.

E finalmente, vamos ao método run, que é quem efetivamente realiza o escalonamento, abaixo, cada trecho do código será exibido e explicado, muita atenção pois agora vem o coração do sistema.

```
43  public String run(){
44      while(processController.getProcessos().size()>0||prontos.size()>0
45              ||bloqueados.size()>0||executando!=null){
46          List<Processo> plusToProntos=processController.getWhoArrivedAtt(this.tempo);
47          for(Processo p:plusToProntos){
48              processController.remove(p);
49              diagramaEventos.add(new Evento("CRIACAO", tempo, p.getPid()));
50              p.moveToNextEvent();
51              prontos.add(p);
52          }
53
```

Figura C2c, inicio do método run

Esta primeira parte do código é deveras importante, pois define a condição de saida do escalonamento e adiciona os processos que chegaram em um determinado tempo na lista de prontos. Começarei explicando o porque da condição do while.

Notamos que o escalonador efetivamente só deveria parar quando não houvessem mais processos para chegar no futuro, ninguém na lista de prontos/bloqueados e também ninguém executando, se a execução chegar neste estado com certeza não existe mais nada para ser feito.

A lista plusToProntos pega todos os processos do ProcessController cujo tempo de chegada coincida com o tempo atual do simulador, abaixo o metodo getWhoArrivedAtt(int tempo):

```
    public List<Processo> getWhoArrivedAtt(int time){
        List<Processo> arrived=new ArrayList<>();
        for(Processo p:this.processos){
            if(p.getInicio().getTempoOcorrencia()==time){
                arrived.add(p);
            }
        }
        return arrived;
    }
```

Figura C2d, metodo getWhoArrivedAtt

Para cada processo que plusToProntos retorna, ele é removido do ProcessController e adicionado a lista de prontos, um evento de CRIACAO é adicionado a lista de Evento diagramaEventos e o processo tem seu evento ativo movido para a próxima posição.

Prosseguindo, ainda dentro do laço while vemos outro bloco grande de código, uma condicional, que só executa se a lista de bloqueados conter algum processo.

```
53      if(!bloqueados.isEmpty()){
54          List<Processo> toRm = new ArrayList<>();
55          for(Processo p:this.bloqueados){
56              Evento e = p.getSelectedEvent();
57              while(!e.getNome().equals("DESBLOQUEIO")){
58                  p.moveToNextEvent();
59                  e = p.getSelectedEvent();
60              }
61              int lockTime=this.getLockTime(p);
62              if(tempo==lockTime+e.getTempoOcorrencia()){
63                  this.diagramaEventos.add(new Evento("DESBLOQUEIO",tempo,p.getPid()));
64                  toRm.add(p);
65              }
66          }
67          for(Processo r:toRm){
68              bloqueados.remove(r);
69              prontos.add(r);
70          }
71      }
```

Figura C2e, run() - condicional em caso de bloqueados

Para cada processo bloqueado, das linhas 56 a 60 tudo que se deseja é garantir que o processo da vez vá para o seu evento de DESBLOQUEIO(que indica quanto tempo o processo deve ficar bloquado) mais próximo, na linha 61 ocorre a chamada de um método para descobrir em qual tempo do relógio ocorreu o bloqueio que levou o processo para a lista de bloquados, abaixo o método:

```
private int getLockTime(Processo p){
    Evento tmp;
    for(int i = diagramaEventos.size()-1;i>=0;i--){
        tmp=diagramaEventos.get(i);
        if(tmp.getOwnerId()==p.getPid()&tmp.getNome().equals("BLOQUEIO")){
            return tmp.getTempoOcorrencia();
        }
    }
    return -1;
}
```

Figura C2f, , metodo getLockTime

Veja que a ideia do metodo e bem simples, ele percorre a List<Evento> diagramaEventos de baixo para cima procurando o ultimo bloqueio daquele processo e retorna o tempo que ocorreu, esse tempo é referente ao tempo do relógio.

Na linha 62 existe uma condicional bem simples, se o tempo atual for igual ao tempo do relogio que ocorreu o ultimo bloqueio do processo mais o tempo de evento DESBLOQUEIO, com certeza é hora de retirar o processo da lista de bloqueados, e isso é feito, o processo é adicionado a uma lista de objetos a serem removidos e o evento de desbloqueio é adicionado ao diagrama de eventos. Por fim, o ultimo for do trecho retira os processos desbloquados da lista de bloqueados e os põe na lista de prontos.

Tendo passado este bloco, agora iremos para o bloco de código que define o que fazer quando não existe ninguém sendo executado em um dado ponto, a seguir, o codigo:

```
72          if(executando==null){//Se não tem ninguem executando
73              this.quantum=this.politica.getQuantum();//No caso de multiplas filas, o qu
74              executando = politica.doSelectionPolitic(prontos);
75              if(executando!=null){
76                  this.qtChaveamentos++;//Sempre que um processo sai da lista de prontos
77                  int tempoDaUltimaOp=this.getLockOrCreationTime();
78                  if(tempo==tempoDaUltimaOp+1){//Se esta execucao ocorre exatamente depo
79                      tempo--;
80                  }
81                  Evento last=lastOperationIsCreation(executando);
82                  if(last!=null){
83                      tme+=tempo-last.getTempoOcorrencia();
84                  }
85                  diagramaEventos.add(new Evento("EXEC",tempo,executando.getPid()));
86              }
87          }else{
```

Figura C2g , metodo run() - condicional executando==null

Quando o Processo executando aponta para null, observe que nas linhas 73 e 74 o quantum do escalonador muda para o quantum da Politica do escalonador, além de que o metodo doSelectionPolitic retorna o processo que deve ser executado(Em caso de sucesso, já o remove da lista prontos também), em caso de sucesso do metodo doSelectionPolitic(prontos não é vazio) a condicional da linha 75 é verdadeira e entramos no bloco da 76.

É aqui que se incrementa a quantidade de chaveamentos, porquê? Observando o diagrama de execução de um processo foi percebido que a troca de contexto sempre ocorre quando um processo vai para o estado executando é que ocorre a contagem de um chaveamento completo, salvo o do primeiro processo que entra no Escalonador.

Figura 3.1 Transições de estado de processo.

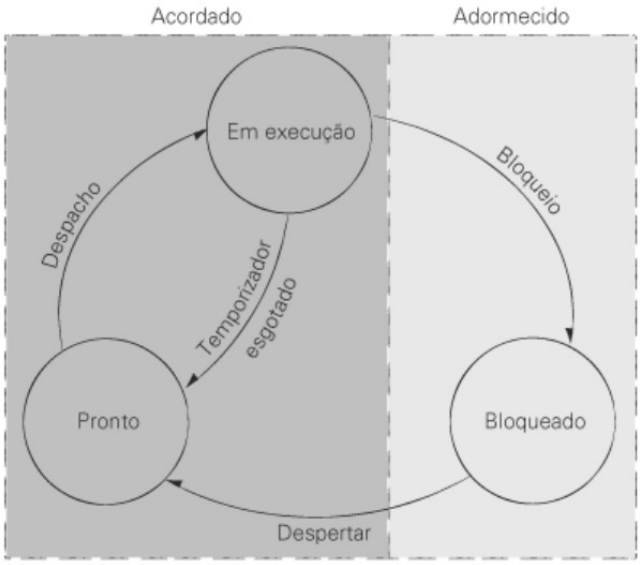


Figura C2h ,transições dos estados de um processo.

Das linhas 78 a 80, é um codigo para adequar o simulador ao resultado esperado pelo professor, a triste realidade é que como foi feito duas operações nunca ocorrem ao mesmo tempo neste simulador, no entanto o resultado esperado pelo professor é que após uma criação ou desbloqueio, se for possível no mesmo tempo o processo disponivel assuma o cargo de executando, para adequar o simulador a estas condições existem essas linhas.

Das linhas 81 a 84, estão ali para calcular o tempo medio de espera de cada processo, basicamente se a operação anterior desta execução for uma CRIACAO, o tempo que este processo teve que esperar para ser executado foi o tempo atual do relogio menos o tempo da ocorrencia da criação, e isto é feito.

```
private Evento lastOperationIsCreation (Processo p){
    Evento tmp;
    for(int i = diagramaEventos.size()-1;i>=0;i--){
        tmp=diagramaEventos.get(i);
        if(tmp.getOwnerId()==p.getPid()){
            if(tmp.getNome().equals("CRIACAO")){
                return tmp;
            }else{
                return null;
            }
        }
    }
    return null;
}
```

Figura C2i , metodo lastOperationIsCreation

Veja que o novamente o diagrama de eventos foi util, admito que a principio havia pensado nele somente como saida mas acabou sendo bastante util para outras coisas.

Na linha 85, finalmente o diagrama de eventos adiciona o evento de execução no dado tempo que ocorreu.

Agora, se o Processo executando não for nulo, vamos para o caso do else da linha 87, que é:

```
87          }else{
88              //Se alguem esta executando
89              int tempoExecucao=getExecutionTime(executando);
90              executando.tempoOcupouCpu+=1;
91              if(tempo==tempoExecucao+quantum&&quantum!=0){
92                  if(executando.tempoOcupouCpu<executando.getFim().getTempoOcorrencia()){
93                      prontos.add(executando);
94                      this.diagramaEventos.add(new Evento("QUANTUM_EX", tempo, executando.getPid()))
95                      executando=null;
96                  }
97              }else{
```

Figura C2j , metodo run – caso de Quantum_Ex

Observe que na linha 89 ocorre uma chamada de metodo geteExecutionTime(), este metodo retorna o tempo em que ocorreu a ultima EXEC deste processo no diagrama de eventos, o tempo de ocupacao da CPU deste processo é incrementado na linha 90.

```
private int getExecutionTime(Processo p){
    Evento tmp;
    for(int i = diagramaEventos.size()-1;i>=0;i--){
        tmp=diagramaEventos.get(i);
        if(tmp.getOwnerId()==p.getPid()&tmp.getNome().equals("EXEC")){
            return tmp.getTempoOcorrencia();
        }
    }
    return -1;
}
```

Figura C2j , metodo getExecutionTime

Na linha 91, existe uma condicional verificando se chegou o momento de uma exceção de quantum, ela é descoberta pela soma do tempo da ultima execução o quantum do Escalonador, e só por garantia na 92 existe uma verificação se o processo já não ocupou mais tempo na CPU do que deveria, nas linhas 93 a 95 o processo retorna para a lista de prontos e um evento QUANTUM_EX vai para o diagrama de eventos.

Só que se ainda não for o tempo de uma QUANTUM_EX, caimos na linha 97, que vai tratar dos bloqueios por chamada de sistema, veja:

```
97  ;e{
98  Evento e = executando.getSelectedEvent();
99  while(!e.getNome().equals("BLOQUEIO")&&!e.getNome().equals("TERMINO")){
100             executando.moveToNextEvent();
101             e = executando.getSelectedEvent();
102 }
103 int remainingTime=executando.getFim().getTempoOcorrencia()-executando.tempoOcupouCpu;
104 if(e.getNome().equals("BLOQUEIO")){
105     if(tempo==tempoExecucao+e.getTempoOcorrencia()
106        &&remainingTime>=e.getTempoOcorrencia()){
107         diagramaEventos.add(new Evento(e.getNome(),tempo, executando.getPid()));
108         bloqueados.add(executando);
109         executando=null;
110     }
111 }
112 if(remainingTime<=0){
113     if(tempo<=(1000*unidadesMilenio)){
```

Figura C2l , metodo run – executando!=null e não quantum_ex

Das linhas 98 a 102, tudo que se dejeja é mover o evento atual do Processo executando para o proximo ponto de parada, que é ou um TERMINO ou um BLOQUEIO.

Na linha 103, calcula-se quanto tempo o processo ainda precisa executar no simulador através da subtração do total de tempos que o processo deve ocupar na CPU menos o quanto já ocupou, em caso do evento atual ser um BLOQUEIO, é checada a condicional das linhas 105/106, que só sera valido se o tempo do relogio do simulador for igual ao tempo da ultima execução mais o tempo do bloqueio E se o tempo do bloqueio isoladamente(descontextualizado do relogio do Escalonador) não ultrapassou o limite de tempo que o processo ocupa na CPU, essa segunda condição pode dar false para casos em que o quantum é diferente de zero.

Se todas essas condições forem verdadeiras, o processo é posto na lista de bloquados, o diagrama de eventos adiciona o evento de bloqueio e executando aponta para nulo.

```
112             if(remainingTime<=0){
113                 if(tempo<=(1000*unidadesMilenio)){
114                     this.vazao++;
115                 }
116                 this.sequenciaTermino.add(executando);
117                 diagramaEventos.add(new Evento("TERMINO",tempo, executando.getPid()));
118                 executando=null;
119                 tempo--;
120             }
121         }
122     }
123     tempo++;
124     if(tempo>1000*unidadesMilenio){
125         unidadesMilenio++;
126     }
127 }
128 this.qtChaveamentos--;
```

Figura C2m , metodo run – semi final

Continuando, na linha 112 é checado se o tempo de execução do processo acabou, se sim, nas linhas 113 a 115 se incrementa o atributo vazao, agora finalmente sera explicado o porque do atributo unidadesMilenio, a ideia é a seguinte, veja que for a do caso else de “ninguém executando no momento” o tempo é incrementado, depois ocorre uma chicarem para ver se o tempo é maior que 1000*unidadesMilenio, se sim, unidadesMilenio é incrementado, ou seja, a cada vez que o escalonador rodar 1000 unidades, unidades Milenio sera incrementado, na linha 113 se o tempo for inferior a 1000*unidadesMilenio, a vazao é somada, no final a vazao será dividida por unidadesMilenio para se descobrir a vazao real.

Das linhas 116 a 119 se faz o procedimento de termino, adiciona o processo atual na sequencia de termino, adiciona o evento ao diagrama de eventos, executando passa a apontar para null. Na 120 aquele tempo-- existe por questões de ajustes para com a resposta final esperada pelo professor.

E agora, finalmente sera exibida a parte final do metodo run, que calcula o tempo médio de resposta e se encarrega de deixar a saida nos padrões esperados, segue:

```
126             }
127         }
128         this.qtChaveamentos--;
129         tme=tme/sequenciaTermino.size();
130         for(Processo p:sequenciaTermino){
131             tmr+=getLastExecutionTime(p)-getFirstExecutionTime(p);
132         }
133         tmr=tmr/sequenciaTermino.size();
134         String saida="";
135         saida+="CHAVEAMENTOS: "+this.qtChaveamentos+"\n"+"TME: "+tme+"\n"+"TMR: "
136         +tmr+"\nVAZAO: "+vazao/unidadesMilenio+"\nSEQUENCIA DE TERMINO: ";
137         for(Processo p:sequenciaTermino){
138             saida+=p.getPid()+" ";
139         }
140         saida+="\n"+getDiagramaEventosString();
141         return saida;
142     }
```

Figura C2m , metodo run – final

Veja que na tinha 131 se subtrai um chaveamento, referente a primeira vez que um processo foi em execução e um chaveamento foi contado, o tme é calculado pela divisão do mesmo pelo numero de processos que executaram, o tmr de cada processo é calculado pela diferença do tempo da execução da ultima instrução EXEC com a primeira, depois é dividido pela quantidade de processos que foram escalonados tambem, as linhas subssequentes apenas colocam a saida em uma String, o ultimo detalhe importante e que a vazao e dividida pela quantidade de unidades de milenio por fim, e pronto, processos escalonados.

VII. Considerações Finais

Realizar este trabalho nos foi muito gratificante pois nos ajudou a ficar as principais politicas de escalonamento de processos usadas pelo Sistema Operacional, além de que tivemos a chance de usar uma aplicação útil de orientação a objetos e rever o conteúdo de sala de forma prática.

De fato vimos que a politica de escalonamento de processos interfere e muito no tempo final de processamento. De forma que a escolha da mesma não pode ser arbitrária.

VIII. Referencias

CORMEN, Algoritmos: Teoria e Prática. Sexta edilçao. Rua Sete de Setembro, 116, decimo sexto andar, 20050-006, Rio de Janeiro(RJ) Brasil: , Campus, 2002. 950 páginas.