Algoritmo Round-Robin – Relatório de desenvolvimento

Gabriel Brunichaki e Paulo Aranha

02 de maio de 2019

Resumo

Este relatório tem por objetivo descrever uma solução para o problema proposto para o primeiro trabalho da disciplina de Sistemas Operacionais do quarto nível do curso de Engenharia de Software da Pontifícia Universidade Católica do Rio Grande do Sul, que trata de desenvolver um algoritmo de escalonamento *Round-Robin* que utilize leitura de arquivos de texto e imprima um gráfico textual exibindo como os processos foram executados.

Na introdução do relatório será descrito melhor o problema e adiante a solução desenvolvida pela dupla e a justificativa de suas escolhas, além da apresentação do resultado a partir do arquivo base.

Introdução

Dentro do escopo da disciplina de Sistemas Operacionais, o problema proposto no primeiro trabalho trata-se de um algoritmo de escalonamento *Round-Robin* sem prioridade e pode ser resumido da seguinte forma: a partir da entrada de um arquivo de texto que contém o número de processos, o tamanho da fatia de tempo e as informações de cada processo (chegada, tempo de execução e pedidos de E/S), o primeiro número da linha de processo representa o tempo de chegada dele no processador, o segundo o tempo de execução e os seguintes, caso existam, representam os tempos em que acessam operações de E/S (entrada e saída). A figura abaixo representa melhor essa descrição.

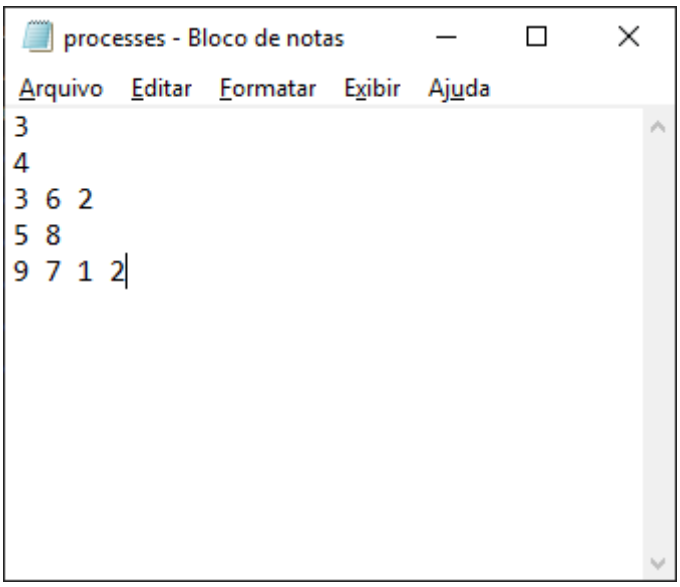

Figura 1: Exemplo de um arquivo que é lido pelo programa, representando o número de processos, fatia de tempo e os processos.

Pegando de exemplo os dados da imagem acima, temos três processos e uma fatia de tempo de quatro unidades. O primeiro processo tem sua chegada no tempo 3, tempo de duração de 6 unidades e uma solicitação de E/S ao executar duas unidades de tempo. Já no segundo processo não há operações de entrada e saída, enquanto no terceiro processo há duas solicitações, uma quando o processo executar uma unidade de tempo e outra quando o processo retornar ao processador e executar por mais duas unidades de tempo.

Neste problema também é considerado a troca de contexto, representado no grafo que virá a seguir pelo caractere *C*. A troca de contexto acontece quando um processo irá entrar no processador ou sair dele. Já quando o processador estiver livre de processos, representaremos no gráfico com um traço (–). Além disso, cada número no gráfico representa a identificação de um processo, que é definida de acordo com a sua ordem de chegada.

Vale destacar uma política deste problema que quando retornar à *CPU*, o processo deve terminar sua fatia de tempo, ou seja, não pode começar uma nova fatia a não ser que tenha finalizado a anterior.

Por fim, ao parar para realizar uma solicitação de E/S, o processo deverá permanecer fora da *CPU* por três unidades de tempo.

A solução foi desenvolvida em Java com o IDE Eclipse e nas próximas seções apresentaremos o resultado obtido, além dos testes realizados.

1 Classes

A solução proposta foi desenvolvida utilizando-se dos conceitos da Programação Orientada a Objetos e consiste em quatro classes: *Process*, *Scheduler*, *CPU* e *App*.

1.1 Process

A classe *Process* simula os processos que serão escalonados e processados pelo algoritmo. Seus atributos são: *id* (número maior que zero que identifica o processo), *arrivalTime* (instante que o processo chega na fila), *executionTime* (unidades de tempo que o processo necessita para ser finalizado), *alreadyExecuted* (unidades de tempo que o processo já utilizou do processador), *currentSlice* (unidades de tempo que o processo utilizou de sua fatia de tempo atual), *countToIo* (unidades de tempo restantes até a solicitação de E/S) e *doingIo* (unidades de tempo que o processo está parado fazendo E/S). Além de todos estes atributos numéricos, cada processo também possui uma lista numérica (*ioTimes*) do tipo *ArrayList* que armazena os tempos de acesso à operações de E/S.

Seu método principal é o *execute()*, que nada mais faz além de incrementar em uma unidade os atributos *currentSlice*, *alreadyExecuted* e *countToIo*.

Vale destacar que a classe implementa a interface *Comparable* para que as listas de processos possam ser ordenadas por ordem de chegada dos processos (*arrivalTime*).

1.2 Scheduler

A classe *Scheduler* realiza o escalonamento dos processos. Seus atributos são: *numberOfProcesses* (número de processos indicado no arquivo de texto), *timeSlice* (fatia de tempo indicada no arquivo de texto) e *countProcesses* (número de processos que já foram adicionados ao escalonador). Além destes atributos numéricos, o escalonador também possui uma lista de processos (*processesList*) do tipo *ArrayList* que armazena os processos recebidos através do arquivo de texto e uma fila de processos (*processesQueue*) que é responsável por realizar o controle do escalonamento dos processos, indicando o próximo processo a entrar no processador.

Os métodos principais da classe são: *addProcess* (que recebe um processo ainda durante a leitura do arquivo e o adiciona na fila de processos do escalonador), *getProcessesList* (que retorna uma lista

de processos do tipo *ArrayList* ordenada por ordem de chegada) e *calcSumTime* (que retorna a soma dos tempos de execução de todos os processos).

1.3 CPU

A classe *CPU* simula um processador e, junto com um escalonador, administra a execução dos processos. Seus atributos são: *totalTime* (provido pelo escalonador, armazena a soma dos tempos de execução de todos os processos), *usedTime* (unidades de tempo utilizadas até o momento para execução dos processos), *currentTime* (unidade de tempo atual) e *contextSwitch* (que indica se vai haver uma troca de contexto). Além destes três atributos numéricos e um atributo "booleano", a CPU também possui um escalonador, um processo em execução, uma lista de processos (*arrivalsList*) que ainda não chegaram e uma lista de processos que estão bloqueados aguardando E/S.

Seu único método é o *run*, que será explicado detalhadamente na seção 4, por se tratar do algoritmo principal da solução.

1.4 App

A classe *App* armazena o método *main*, que inicializa a aplicação. Além disso, é responsável por instanciar os objetos "Escalonador"e "CPU"e fazer a leitura do arquivo de texto, adicionando os processos na lista do escalonador. Por fim, chama o metodo *run* da classe *CPU*.

2 Método run()

O método *run* da classe *CPU* é o responsável por realizar toda a lógica da solução apresentada. O algoritmo todo está contido dentro de uma estrutura de repetição *while* e se repete enquanto a condição *usedTime* < *totalTime* for verdadeira, ou seja, enquanto a unidades de tempo utilizadas para a execução dos processos for menor que a soma dos tempos de execução dos processos.

Primeiramente, verificamos com mais uma estrutura de repetição se existem processos que estejam chegando no tempo atual. Se sim, adicionamos o(s) processo(s) na fila de espera do escalonador. Caso não exista um processo sendo executado neste tempo, a troca de contexto torna-se verdadeira, indicando que um processo entrará no processador.

Em seguida, verificamos se a lista de processos bloqueados não está vazia. Se verdadeiro e se o tempo necessário para fazer uma operação de entrada e saída for (3) for atendido, o processo é removido da fila de bloqueados e adicionado na fila de prontos para executar. Como na verificação anterior, a troca de contexto torna-se verdadeira se não há um processo sendo executado.

Posteriormente, criamos uma estrutura condicional composta por três condições:

2.1 Se há troca de contexto

Se a variável *contextSwitch* é verdadeira, inserimos no gráfico o caractere *C*. Em seguida, levamos para o processador o primeiro processo da fila de espera, se houver, e definimos a variável de troca de contexto como falsa.

2.2 Se há um processo sendo executado

Se existe um processo sendo executado pelo processador, executamos o método *execute* do processador, incrementamos a variável *usedTIme* da CPU e inserimos no gráfico o número identificador do processo. Em seguida, verificamos se o processo solicitou E/S neste tempo, ou se a fatia de tempo do processo acabou ou se o processador finalizou o processo.

Havendo solicitação de E/S, removemos o tempo de acesso da lista *ioTimes* do processador, zeramos a variável responsável por armazenar as unidades de tempo restantes até a solicitação de E/S (*countToIo*) e adiciona o processo na lista de processos bloqueados.

Se o processo não finalizou e se não houve solicitação de E/S, o processo retorna para o final da fila de espera para execução.

Por fim, o processo é removido do processador e a variável de troca de contexto passa a ser verdadeira.

2.3 Se não há troca de contexto e não há processo sendo executado

Se as duas condições anteriores não forem atendidas, inserimos no gráfico o caractere de traço (–), indicando que não há nenhum processo pronto para executar nesta unidade de tempo.

3 Casos de teste

- **Exemplo:** - - C 1 1 C 2 2 2 2 C 1 1 C 3 C 2 2 2 2 C 1 1 C 3 3 C - C 3 C 3 3 3
- **Teste1:** C 1 1 1 1 1 C 2 2 2 2 2 2 2 C 3 3 3 3 3 C 4 4 4 4 4 4 4 4 C 5 5 5 5 5 5 5 C 1 1 C 2 2 2 2 2 C 3 3 C 4 4 4 4 4 4 4 4 C 5 C 1 1 1 C 3 3 3 3 3 3 3 C 4 C 3
- **Teste2:** C 1 1 1 1 1 C 2 2 2 2 2 2 2 C 1 1 C 2 2 2 2 2 C 1 1 1 C - - - - - - C 3 3 3 3 C 4 4 4 4 4 4 4 4 C 5 5 5 5 5 5 5 C 3 3 3 C 4 4 4 4 4 4 4 4 C 5 C 3 3 3 3 3 C 4 C 3 3 C 3
- **Teste3:** C 1 1 1 1 1 C 2 2 2 2 2 C 1 1 1 1 1 C 3 3 3 3 3 C 2 2 2 2 2 C 3 3 3 3 3 C 4 4 4 4 4 C 2 2 C 5 5 5 5 5 C 3 3 3 3 3 C 4 4 4 4 4 C 5 5 5 C 4 4 4 4 4
- **Teste4:** C 1 1 C 2 C 3 3 C 4 C 5 5 C 6 C 1 C 7 7 C 8 C 9 9 C 2 C 10 C 11 11 C 12 C 3 C 13 13 C 14 C 15 15 C 4 C 16 C 17 17 C 18 C 5 C 19 19 C 20 C 21 21 C 6 C 22 C 23 23 C 24 C 7 C 25 25 C 8 C 2 C 9 C 10 C 11 C 12 C 13 C 14 C 4 C 15 C 16 C 17 C 18 C 19 C 20 C 6 C 21 C 22 C 23 C 24 C 8 C 25 C 10 C 12 C 14 C 16 C 18 C 20 C 22 C 24