Simulador de Escalonamento dum Sistema Operativo

Autor: Vasco Barnabé
Number: nº42819

9 de Abril de 2020

1 Introdução

Foi proposto a realização de um simulador de escalonamento de um sistema operativo, tendo por base uma arquitetura do modelo de 3 estados que consome programas constituídos por um conjunto de instruções.

No fundo, este trabalho exige não a implementação de um simulador de um escalonamento, mas sim de dois escalonamentos distintos (veremos mais à frente como podemos adaptar este simulador de modo a funcionar para ambos os escalonamentos), denominados por **FCFS** e **Round Robin**, e para isso, foi implementada uma arquitetura do modelo de 3 estados, sendo os mesmos os estados **READY**, **RUN** e **BLOCKED**. Veremos como funcionam estes tipos de escalonamento nesta arquitetura.

2 Escalonamento FCFS

Este tipo de escalonamento é do tipo não preemptivo, ou seja, não há interrupção dos processos quandos estes entram no CPU (estado **RUN**), pelo que os mesmos só irão sair do CPU quando o seu tempo lá terminar. Este tipo de escalonamento apresenta um problema óbvio: ao ser um escalonamento do tipo não preemptivo, quando um processo entra em ciclo infinito nunca vai sair do CPU (estado **RUN**). Para além disso, este tipo de escalonamento apresenta outro problema, que será referido mais tarde. No entanto, é importante termos primeiro a perceção do funcionamento deste sistema, de modo a que se torne mais evidente o problema em causa.

Como já foi referido, ao utilizar uma arquitetura do modelo de 3 estados, o objetivo é, consoante o tempo de entrada de cada projeto no programa, estes sejam inseridos no estado **READY**, que será um estado de espera onde os processos vão esperar pela sua vez para irem para o CPU (**RUN**); esta libertação de processos do estado **READY** para o estado **RUN** será feita por ordem de chegada, ou seja, será libertado o processo que esteja à espera à mais tempo no estado **READY**.

Uma vez chegado ao estado **RUN**, o processo permanecerá lá o tempo que for necessário, sem interrupções, e quando esse tempo terminar, será enviado até ao estado **BLOCKED** (à exceção de quando o processo termina a sua estadia neste sistema, o que acontece sempre no

estado **RUN**, e aí o processo simplesmente desaparece e deixa de fazer parte do sistema) onde irá ter acesso a I/O com espera de x instantes.

Após estes instantes, o processo retornará ao estado **READY** para repetir todo o ciclo, até ao seu fim. De sublinhar que se num mesmo instante, existir um processo vindo do estado **BLOCKED** para o estado **READY**, e um processo novo a querer entrar também neste estado, o processo vindo do estado **BLOCKED** tem prioridade.

Assim sendo, uma vez que já foi exposta a forma como este tipo de escalonamento funciona numa arquitetura de 3 estados, é claro o problema anteriormente referido mas não revelado: os processos de baixo custo de execução podem ter de esperar muito tempo para serem executados.

3 Escalonamento Round Robin

Observando agora o escalonamento Round Robin, pode-se adiantar que este não é muito diferente do FCFS, apresentando apenas uma diferença. Enquanto que no FCFS os processos não sofrem interrupções no CPU (**RUN**), e por isso é um escalonamento do tipo não preemptivo, o Round Robin permite essas interrupções, sendo assim, logicamente, preemptivo.

Há um tempo limite para que cada processo possa "correr"no CPU sem que seja interrompido, denominado de QUANTUM. Não perdendo tempo com pormenores, apresento um exemplo que deixará, de forma rápida, clara e eficaz, a forma como este QUANTUM vai atuar. Imagine-se um processo que necessita de estar no CPU durante 5 ciclos de relógio, mas o QUANTUM tem valor de 3; o processo irá estar 3 ciclos de relógio no CPU e será enviado para o estado **READY**, ficando na fila de espera para poder retornar ao estado **RUN** (CPU); uma vez de volta ao CPU, o processo permanecerá lá 2 ciclos de relógio e de seguida ou sai do sistema (caso o processo tenha terminado) ou vai para o estado **BLOCKED**, à semelhança do que acontece no escalonamento FCFS.

Tal como acontece no FCFS e em todos os outros tipos de escalonamento, o Round Robin apresenta problemas. Neste caso, é percetível que processos que necessitem de muitos ciclos de relógio no estado **RUN**, estarão a ser constantemente interrompidos, levando a um aumento do tempo que o programa demorará a executar.

4 Implementação

Uma vez que em 2 dos 3 estados do sistema (**READY** e **BLOCKED**) poderiam estar vários processos e que estes estariam constantemente a entrar e a sair, tendo sempre prioridade na saída o processo que estivesse há mais tempo no estado em questão, foram implementadas filas de dados, mais conhecidas por queues, para resolver a situação. Pode-se dizer assim que os estados **READY** e **BLOCKED** são filas distintas.

Este trabalho começou com a implementação das queues, reunindo um total de 7 funções e claro, de uma struct, chamada de struct queue, onde seriam armazenados os dados, mais especificamente, onde seriam guardados os valores de PID de cada processo que estivesse naquele instante na queue (veremos mais à frente o que significa PID). Assim, as funções desenvolvidas foram:

- Queue* create queue(int size) - retorna uma queue vazia com o tamanho pretendido, recebido como argumento;
- void enqueue(Queue* fila, int value) - recebe como argumentos uma determinnada queue e um determinado valor, e insere na queue esse valor;
- bool empty(Queue* fila) - verifica se a queue recebida como argumento está vazia ou não, sendo por isso uma função booleana;

- int dequeue(Queue* fila) - função que retira de uma determinada queue o elemento que lá esta há mais tempo (um valor inteiro neste caso);
- void free queue(Queue* fila) - função que liberta toda a informação que esteja na queue, ficando esta vazia;
- int top(Queue* fila) - retorna o valor que está no topo da queue, sem o retirar desta, ou seja, apenas mostra qual o próximo valor a sair quando tal for pretendido;
- void print queue(Queue* fila) - função utilizada para mostrar todos os elementos presentes na queue num determinado instante.

Observemos agora um outro tópico importante deste trabalho: o input. O programa recebe como input vários processos e informações específicas e exclusivas para cada processo. Para que esta informação estivesse guardada de forma organizada, por forma a ser utilizada sem quaisquer problemas pelo sistema, foi criada uma nova struct, chamada de struct process, desta vez para guardar dados referentes a cada processo.

Assim, esta struct guarda valores fornecidos pelo input, tais como:

- o PID (número identificador de cada processo);
- o tempo de início de cada processo;
- uma sequência de valores, valores esses que correspondem ao número de ciclos de relógio que o processo em questão terá que estar no estado **RUN**;
- outra sequência de valores, correspondentes ao número de ciclos de relógio que o processo especificado terá que estar no estado **BLOCKED**;
- entre outros valores criados e modificados durante o desenvolvimento do programa, específicos de cada processo.

Para ler o input, organizar as informações relativas a cada processo e, num estado mais avançado do programa, trabalhar com estes processos e as suas informações, foram criadas as seguintes funções:

- Process* new process(int size) - função que retorna um novo processo;
- Process* organizar processos(int array[ ], int inicio, int fim) - função que atribui às variáveis os valores corretos, separando corretamente os valores de PID, tempo de início, etc...de cada processo;
- int find PID(int PID, Process* array processos[ ], int k) - todos os processos serão guardados num array de processos, onde em cada posição desse array será guardado um único processo. Assim, esta função recebe como argumentos o PID de um processo, o array de processos e o tamanho desse array, retornando a posição do array em que se encontra o processo com aquele PID em questão, e assim, será possível aceder ao resto da informação daquele processo.

Até ao momento, apenas foram apresentadas as estruturas de dados utilizadas e funções implementadas para simplificar o desenvolvimento do programa. Debrucemo-nos agora sobre o funcionamento do programa em si.

Na função "main"começa-se por construir um output simples e claro, de modo a permitir ao utilizador do programa escolher que tipo de escalonamento pretende simular e com qual ficheiro. De seguida, todo o input é lido para um array de inteiros, o qual vai ser repartido, utilizando a

função "organizar processos", que vai devolver um novo processo com todas as suas informações, cada vez que for chamada. Cada vez que for criado um novo processo, o mesmo será inserido no array de processos. Após este ciclo, serão atribuídos valores a algumas variáveis pertencentes à struct process que serão úteis durante o funcionamento do programa.

Posteriormente, será verificado qual escalonamento foi pretendido pelo utilizador, FCFS ou ROUND ROBIN, e será feita uma chamada a uma nova função:

- solve(array processos, k, QUANTUM FCFS) ou solve(array processos, k, QUANTUM RR) - estamos a falar da mesma função, mas serão enviados parâmetros diferentes consoante a escolha do utilizador sobre qual escalonamento testar; assim, se o utilizador escolher testar o FCFS, será chamada a função "solve"com o parâmetro QUANTUM FCFS, e se escolher testar o ROUND ROBIN, será chamada a mesma função, mas com o parâmetro QUANTUM RR.

Uma vez chamada a função "solve", é nela que ocorre o programa em si, onde se simula o funcionamento deste modelo de 3 estados.

No início desta função são criadas 2 queues, sendo estas representadas, como já foi referido, pelos estados **READY** e **BLOCKED**. O estado **RUN**, surge nesta implementação apenas como uma variável do tipo *int*.

Esta não é uma função recursiva, funcionando assim o sistema em ciclo, uma vez que a cada ciclo de relógio, todo o tipo de processos e verificações se repetem, começando o ciclo colocando no estado **READY** o processo com tempo de entrada inferior a todos os outros, e terminando no estado **RUN**, com o processo que mais tempo demore a percorrer o seu "percurso"no sistema.

Existem inúmeras passagens de processos entre os 3 estados, e por isso, é necessário que haja uma lista de prioridades. Assim, quando no mesmo instante, um processo novo ou vindo do **BLOCKED**, e /ou do **RUN** pretendem entrar na fila **READY**, o vindo do **BLOCKED** tem prioridade, seguido do do **RUN**, e por fim o processo novo. Por isso, a cada ciclo de relógio o primeiro passo é verificar se a fila do **BLOCKED** está vazia ou não, e se não estiver, verificar se há algum processo que deva ir para a fila **READY** nesse instante. De seguida, há que verificar se o processo presente no estado **RUN** tem de ir para a fila **READY** nesse instante ou não, e há uma série de prioridades no que toca à razão pela qual o processo deve continuar ou não no **RUN**. Assim, estas prioridades ordenam-se por:

1. verificar se o processo terminou todos os seus instantes de espera para acesso a I/O (estar no estado **BLOCKED**), e se é o seu último instante no CPU. Se for esse o caso, então o processo é excluído do array de processos e deixa de fazer parte do funcionamento do sistema daí para a frente e todas as seguintes condições não são sequer verificadas;

2. verificar se o processo terminou o seu tempo no CPU e ainda tem tempos de espera de acesso a I/O. Caso isso se verifique, o processo é enviado para o estado **BLOCKED** e nenhuma das condições seguintes é verificada;

3. verificar se o número de instantes seguidos do processo no CPU atingiu o QUANTUM, e caso isso se verifique, o processo é enviado para o estado **READY** e quando retornar ao **RUN** começa do ponto onde tinha ficado quando saiu (é importante sublinhar que quando se estiver a simular o escalonamento Round Robin, esta ação poderá ocorrer muitas vezes, enquanto que se a simulação for relativa ao escalonamento FCFS, não há limitação de instantes seguidos no CPU para qualquer processo, e por isso, nunca haverá transição de processos do estado **RUN** para o **READY**).

Só, e apenas posteriormente a isto, se verificará se existe algum processo novo para entrar no sistema. De seguida, se o estado **RUN** estiver vazio e a fila **READY** não estiver, então o processo mais adiantado nesta fila será enviado para o **RUN**.

Para terminar, no fim de todos os ciclos de relógio, é exposto em forma de output os 3 estados, mostrando os processos que contêm, sendo também verificado se todos eles estão vazios, e se tal acontecer, o programa termina.

5 Output

Como exemplo, é aqui apresentado o output resultante da simulação de ambos os escalonamentos tendo como input o ficheiro "input1.txt":

Escalonamento FCFS:

0 —READY: 101	—RUN: 100	—BLOCKED: Empty
1 —READY: 200 300	—RUN: 101	—BLOCKED: 100
2 —READY: 200 300	—RUN: 101	—BLOCKED: 100
3 —READY: 200 300	—RUN: 101	—BLOCKED: 100
4 —READY: 200 300 100	—RUN: 101	—BLOCKED: Empty
5 —READY: 300 100	—RUN: 200	—BLOCKED: 101
6 —READY: 300 100	—RUN: 200	—BLOCKED: 101
7 —READY: 100	—RUN: 300	—BLOCKED: 101 200
8 —READY: 100	—RUN: 300	—BLOCKED: 101 200
9 —READY: 100 101	—RUN: 300	—BLOCKED: 200
10 —READY: 100 101	—RUN: 300	—BLOCKED: 200
11 —READY: 100 101	—RUN: 300	—BLOCKED: 200
12 —READY: 100 101 200	—RUN: 300	—BLOCKED: Empty
13 —READY: 100 101 200	—RUN: 300	—BLOCKED: Empty
14 —READY: 101 200	—RUN: 100	—BLOCKED: 300
15 —READY: 101 200	—RUN: 100	—BLOCKED: 300
16 —READY: 101 200	—RUN: 100	—BLOCKED: 300
17 —READY: 101 200	—RUN: 100	—BLOCKED: 300
18 —READY: 101 200	—RUN: 100	—BLOCKED: 300
19 —READY: 101 200	—RUN: 100	—BLOCKED: 300
20 —READY: 101 200 300	—RUN: 100	—BLOCKED: Empty
21 —READY: 101 200 300	—RUN: 100	—BLOCKED: Empty
22 —READY: 101 200 300	—RUN: 100	—BLOCKED: Empty
23 —READY: 101 200 300	—RUN: 100	—BLOCKED: Empty
24 —READY: 200 300	—RUN: 101	—BLOCKED: 100
25 —READY: 200 300	—RUN: 101	—BLOCKED: 100
26 —READY: 300	—RUN: 200	—BLOCKED: 100
27 —READY: 100	—RUN: 300	—BLOCKED: 200
28 —READY: Empty	—RUN: 100	—BLOCKED: 200
29 —READY: 200	—RUN: 100	—BLOCKED: Empty
30 —READY: 200	—RUN: 100	—BLOCKED: Empty
31 —READY: 200	—RUN: 100	—BLOCKED: Empty
32 —READY: 200	—RUN: 100	—BLOCKED: Empty
33 —READY: 200	—RUN: 100	—BLOCKED: Empty
34 —READY: Empty	—RUN: 200	—BLOCKED: Empty
35 —READY: Empty	—RUN: 200	—BLOCKED: Empty
36 —READY: Empty	—RUN: 200	—BLOCKED: Empty

Escalonamento Round Robin:

0 —READY: 101	—RUN: 100	—BLOCKED: Empty
1 —READY: 200 300	—RUN: 101	—BLOCKED: 100
2 —READY: 200 300	—RUN: 101	—BLOCKED: 100
3 —READY: 200 300	—RUN: 101	—BLOCKED: 100
4 —READY: 300 100 101	—RUN: 200	—BLOCKED: Empty
5 —READY: 300 100 101	—RUN: 200	—BLOCKED: Empty
6 —READY: 100 101	—RUN: 300	—BLOCKED: 200
7 —READY: 100 101	—RUN: 300	—BLOCKED: 200
8 —READY: 100 101	—RUN: 300	—BLOCKED: 200
9 —READY: 101 300	—RUN: 100	—BLOCKED: 200
10 —READY: 101 300	—RUN: 100	—BLOCKED: 200
11 —READY: 101 300 200	—RUN: 100	—BLOCKED: Empty
12 —READY: 300 200 100	—RUN: 101	—BLOCKED: Empty
13 —READY: 200 100	—RUN: 300	—BLOCKED: 101
14 —READY: 200 100	—RUN: 300	—BLOCKED: 101
15 —READY: 200 100	—RUN: 300	—BLOCKED: 101
16 —READY: 100 300	—RUN: 200	—BLOCKED: 101
17 —READY: 300 101	—RUN: 100	—BLOCKED: 200
18 —READY: 300 101	—RUN: 100	—BLOCKED: 200
19 —READY: 300 101 200	—RUN: 100	—BLOCKED: Empty
20 —READY: 101 200 100	—RUN: 300	—BLOCKED: Empty
21 —READY: 200 100	—RUN: 101	—BLOCKED: 300
22 —READY: 200 100	—RUN: 101	—BLOCKED: 300
23 —READY: 100	—RUN: 200	—BLOCKED: 300
24 —READY: 100	—RUN: 200	—BLOCKED: 300
25 —READY: 100	—RUN: 200	—BLOCKED: 300
26 —READY: Empty	—RUN: 100	—BLOCKED: 300
27 —READY: 300	—RUN: 100	—BLOCKED: Empty
28 —READY: 300	—RUN: 100	—BLOCKED: Empty
29 —READY: 100	—RUN: 300	—BLOCKED: Empty
30 —READY: Empty	—RUN: 100	—BLOCKED: Empty
31 —READY: Empty	—RUN: Empty	—BLOCKED: 100
32 —READY: Empty	—RUN: Empty	—BLOCKED: 100
33 —READY: Empty	—RUN: Empty	—BLOCKED: 100
34 —READY: Empty	—RUN: 100	—BLOCKED: Empty
35 —READY: Empty	—RUN: 100	—BLOCKED: Empty
36 —READY: Empty	—RUN: 100	—BLOCKED: Empty
37 —READY: Empty	—RUN: 100	—BLOCKED: Empty
38 —READY: Empty	—RUN: 100	—BLOCKED: Empty
39 —READY: Empty	—RUN: 100	—BLOCKED: Empty

6 Conclusão

Com este trabalho foi possível aprofundar conhecimentos sobre estruturas de dados, nomeadamente structs e queues, e superar dificuldades.

Surgiram dificuldades, a primeira das quais, e a maior também, foi como ler bem o input e guardar corretamente essa informação, de modo a que pudesse ser usada sem problemas durante o desenvolvimento do programa; foi aqui um dos casos onde se deu uso às structs, e foi criada uma struct process para guardar a informação do input. Outra dificuldade que surgiu foi como saber se um processo já tinha terminado todo o seu percurso no sistema ou não, e para isso, foram criadas variáveis, específicas para cada processo, para que pudessem funcionar como contador e serem sempre consultadas para verificar se o dito processo terminou ou não a sua execução.

As queues foram, como já referi, outra estrutura de dados aqui aplicada, umas vez que os estados **READY** e **BLOCKED** necessitavam de ser filas para conterem mais que um processo no mesmo instante, e houvesse prioridade na ordem de saída.

Deste modo, para concluir, o desenvolvimento deste trabalho foi um desafio constante, contribuindo para treino na área da programação, para a recolha de conhecimento sobre como pode funcionar um modelo de 3 estados, e para ter total perceção do modo de funcionamento de dois tipos de escalonamento em particular: o FCFS e o Round Robin, ficando assim a perceber mais sobre sistemas operativos, aplicando sempre que possível o conhecimento obtido nas aulas de Sistemas Operativos I.