

Primeiro Exercício-Programa

Norton Trevisan Roman

28 de agosto de 2017

1 Escalonador de Processos

Para este trabalho, vocês devem se organizar em grupos de até 4 (quatro) pessoas. Cada grupo deve então implementar um escalonador de tarefas para Time Sharing em uma máquina com um único processador, criando assim um sistema simples de multiprogramação. A linguagem usada na construção do escalonador deve ser Java.

A máquina foi criada para rodar pequenos programas, em que cada processo pode contar, no máximo, com 2 registradores de uso geral (além do Contador de Programa, como registrador de uso específico). Esses registradores são conhecidos internamente como X e Y . Além disso, o processador para o qual vocês irão construir o escalonador é extremamente simples, possuindo apenas 4 instruções:

1. Atribuição: na forma $X=<valor>$ ou $Y=<valor>$, onde $<valor>$ é um número inteiro e X e Y são os registradores de uso geral usados pelo processo (note a ausência de espaço antes e depois do '=').
2. Entrada e saída: representada pela instrução E/S (que faz as vezes de uma chamada ao sistema)
3. Comando: a tarefa executada pela máquina, representada pela instrução COM
4. Fim de programa: chamada com a única finalidade de remover o programa da memória, executando a limpeza final. Representada pela instrução SAIDA

Sabe-se que um processo pode estar em um dos seguintes estados: Executando, Pronto ou Bloqueado. Enquanto há apenas um processo executando, pode haver vários prontos para executar ou bloqueados, esperando alguma requisição de E/S se completar. Assim, sua implementação deve contemplar uma lista de processos prontos e outra de bloqueados.

Na ausência de um *clock* que comande a preempção, quem efetivamente rodará as instruções dos processos é o escalonador, que lê a instrução e a executa, funcionando como um interpretador. Isso deixa o processo mais lento, naturalmente, mas garante o compartilhamento de tempo. Dentro do escalonador, a fila de processos prontos deve ser ordenada conforme a prioridade do processo, enquanto que a fila de bloqueados é ordenada por ordem de chegada.

Seu sistema deve então possuir uma Tabela de Processos, representando todos os programas que estão rodando simultaneamente. Cada linha da tabela deve conter uma referência ao Bloco de Controle de Processo (BCP), sendo que este contém toda a informação necessária para que o processo, após interrompido temporariamente, possa voltar a rodar. Ou seja, o BCP deve conter, pelo menos, o Contador de Programa, o estado do processo (executando, pronto ou bloqueado), sua prioridade, o estado atual de seus registradores de uso geral, uma referência à região da memória em que está o código do programa executado (representado, por exemplo, por um arranjo de Strings, que já é uma referência natural à memória em Java) e o nome do programa.

Vale notar que há somente o segmento de texto na memória (representado, por exemplo, por um arranjo de 21 posições), em que é armazenado o código do programa. Por não conter nem variáveis nem desvios (sub-rotinas etc), não há sentido em ter um segmento de dados e da pilha. Além disso, lembre que, em java, qualquer instância a um objeto ou arranjo já é uma referência a memória externa ao objeto em que essa instância está declarada.

Os programas executados serão dados na forma de arquivos-texto (ver 1.1). O escalonador deve, então, carregar cada bloco de comandos (correspondente a um arquivo) na memória, posicionando seu BCP na Tabela de Processos e na fila de processos prontos, seguindo sempre sua ordem de prioridade. A prioridade de cada processo é, por sua vez, carregada a partir de um arquivo intitulado “prioridades.txt”, que apresenta, a cada linha, a prioridade de cada processo (quando estes são ordenados em ordem alfabética pelo nome de seus arquivos). Nesse caso, quanto maior o valor do número contido no arquivo, maior a prioridade do processo.

Como uma simplificação adicional, em vez de fatias de tempo, o escalonador irá permitir que cada processo no estado *executando* rode no máximo *n*.com comandos (ou seja, o quantum será de *n*.com comandos, em vez de uma quantia de milissegundos). Esse número de comandos é uma simulação do tempo de ocupação do processador relacionado ao time-sharing, e deve ser lido de um arquivo denominado “quantum.txt”. Esse arquivo conterá tão somente um inteiro.

Uma vez tendo carregado todos os processos, o escalonador começa a rodá-los, usando o seguinte algoritmo de prioridades (bastante semelhante ao usado no Linux):

1. Inicialmente, distribua um número de créditos, a cada processo, igual à sua prioridade;
2. Ordene, então, a fila de processos prontos conforme o número de créditos (do maior para o menor);
3. Cada processo deve executar um número fixo de instruções (seu quantum):
 - (a) Ao começar a rodar, o processo perde um crédito (seu contador de créditos é decrementado)
 - (b) Findado o quantum, o processo é reposicionado na fila de processos prontos
 - (c) O primeiro da fila é posto a rodar um quantum inteiro (note que, dependendo da prioridade, pode ser o mesmo processo de antes)
 - (d) Quanto todos os processos estiverem com **zero** crédito, então os créditos são redistribuídos, conforme sua prioridade, voltando assim ao passo 1;

4. Se, durante a execução de um quantum, o processo fizer uma entrada ou saída (instrução “E/S”):
 - (a) Ele será marcado como *bloqueado*, sendo então transferido para a lista de bloqueados;
 - (b) A ele é atribuído um tempo de espera (inteiro representando quantos quanta ele deve esperar para rodar novamente);
 - (c) A cada processo que passe pelo estado *executando* (ou seja, ao sair desse estado), esse tempo de espera é decrementado (note que todos na fila de bloqueados têm seu tempo decrementado);
 - (d) Cada processo fica *bloqueado* até que dois outros processos passem pelo estado *executando* (esse é o tempo de espera), não importando quantos comandos cada um executou (ou seja, se usou todo seu quantum ou não). Essa é uma simulação do tempo de espera por um dispositivo de E/S (note que, uma vez que o tempo de resposta de uma E/S é igual para todos, a lista de processos bloqueados acaba se comportando como uma fila comum);
 - (e) Quando o tempo de espera de algum processo bloqueado chegar a zero, este deve receber o status de *pronto*, sendo então removido da fila de bloqueados e inserido na fila de processos prontos, na posição correspondente ao seu número atual de créditos. Note que ele não necessariamente ocasionará a preempção do processo em execução, ou seja, o escalonador escolherá sempre o de maior número de créditos;
 - (f) Quando esse processo for rodar novamente, deve reiniciar a partir da instrução seguinte à E/S (uma vez que o PC é armazenado no BCP e este contém a instrução seguinte à E/S). Atente que a instrução de E/S foi contada nas estatísticas do sistema durante o momento anterior ao bloqueio (ver Seção 1.2).
5. Se não houver nenhum processo em condição de ser executado (ex: existirem apenas dois processos e ambos estiverem *bloqueados*), deve-se decrementar os tempos de espera de todos os processos na fila de bloqueados, até que um chegue a zero, podendo então ser rodado (como visto no item 4e).
6. Ao encontrar o comando SAIDA, o escalonador deve remover o processo em execução da fila apropriada e da tabela de processos.

Vale lembrar que apenas um máximo de n_{com} instruções (de qualquer um dos 4 tipos definidos mais adiante) podem ser executadas por vez pelo processador quando o processo estiver no estado *executando*. Quando isso ocorrer, o processo terminou seu quantum e deve ir para sua posição na fila de prontos (não necessariamente ao seu final). Um novo processo dessa fila deve então ir para o estado *executando*. Note que isso implica saber qual será o próximo comando a ser executado nesse processo, ou seja, saber o conteúdo de seu Contador de Programa.

1.1 Entrada

Serão dados como entrada 10 arquivos-texto, fornecidos dentro do diretório “processos” (no anexo “EP1.zip”), em que cada arquivo dentro de “processos” corresponde a um processo, construído da seguinte forma:

1. O nome do arquivo corresponde a um inteiro sequencial de dois dígitos (01.txt, 02.txt etc)
2. A primeira linha do arquivo contém o nome do programa
3. As linhas seguintes apresentam uma sequência qualquer de instruções aceitas pela máquina, terminando com SAIDA
4. Cada processo será composto por no máximo 21 comandos (incluindo SAIDA). Assim, cada arquivo conterá, no máximo, 22 linhas (uma linha por comando, além do nome), do tipo:
 - <registrador>=<valor>
 - COM
 - E/S
 - SAIDA

Um exemplo de arquivo seria:

```
TESTE-1
X=8
COM
COM
COM
E/S
Y=10
X=2
COM
E/S
SAIDA
```

Dentro do mesmo diretório, serão também fornecidos o arquivo “prioridades.txt”, que define as prioridades de cada processo (processo 01.txt na linha 1, processo 02.txt na linha 2 e assim por diante), além do arquivo “quantum.txt”, que contém um único inteiro, representando o tamanho do quantum a ser usado (ou seja, o número de instruções rodadas por surto de CPU).

1.2 Saída

Durante o processamento, o escalonador deve construir um logfile, denominado “logXX.txt”, em que XX é o valor do quantum escolhido (2 dígitos) . Nesse logfile, o escalonador deve gravar:

1. Os nomes dos processos carregados, na ordem em que estão na fila de prontos
2. O nome do processo que está sendo interrompido, juntamente com o número de instruções executadas até seu interrompimento (ex: “Interrompendo TESTE-1 após 3 instruções”). Essas instruções referem-se às executadas no último quantum, não o número total desde o início do processo.
3. O nome do processo que passará a ser executado (ex: “Executando TESTE-1”)
4. O nome do processo que inicia uma E/S (ex: “E/S iniciada em TESTE-1”)
5. O nome do processo que terminou (ou seja, teve todos seus comandos executados), juntamente com o valor final dos registradores X e Y (ex: “TESTE-1 terminado. X=2. Y=3”). Se o programa não usar um dos registradores, ele será zero.

Ao final do sistema, você deve incluir no logfile o número médio, por processo, de trocas de processo (ou seja, a média, dentre todos os processos, do número de vezes em que cada processo deixa o processador, incluindo-se o momento de seu término), o número médio de instruções executadas por quantum (corresponde à média, dentre todos os processos, do número de instruções executadas até o processo ser interrompido, seja por E/S, seja porque executou n.com instruções, seja porque terminou no meio de seu quantum – é uma média das instruções executadas em cada quantum, para todos os processos), além do quantum usado.

Um exemplo de logfile é (os valores são meramente ilustrativos, não correspondendo a nenhum exemplo real. Notas entre parênteses são comentários para vocês. Não devem constar do log.):

```
Carregando TESTE-1
Carregando TESTE-3
Carregando TESTE-2
Executando TESTE-1
Interrompendo TESTE-1 após 3 instruções
Executando TESTE-3
E/S iniciada em TESTE-3
Interrompendo TESTE-3 após 2 instruções (havia um comando antes da E/S)
Executando TESTE-2
E/S iniciada em TESTE-2
Interrompendo TESTE-2 após 1 instrução (havia apenas a E/S)
TESTE-2 terminado. X=0. Y=3
Executando TESTE-1
```

```
...
TESTE-1 terminado. X=3. Y=1
...
TESTE-3 terminado. X=4. Y=0
MEDIA DE TROCAS: 5
MEDIA DE INSTRUÇÕES: 2.5
QUANTUM: 3
```

1.3 Implementação

O logfile deve ser atualizado toda vez que o escalonador tiver que tomar uma decisão de escalonamento, conforme descrito em 1.2, ou seja:

- Ao se iniciar (carregar) um processo
- Ao se executar um processo (seja pela primeira vez ou volta de fila de prontos)
- Ao se terminar um processo
- Ao se interromper um processo (bloqueio de E/S, fim do quantum ou término natural do processo)

Também devem ser incluídas as vezes em que a instrução a ser executada for uma E/S, bem como as estatísticas gerais do sistema (médias etc.).

Seu escalonador deve se chamar Escalonador.java. Ao ser chamado da linha de comando, o sistema irá carregar os programas fornecidos (os 12 arquivos estarão em um subdiretório chamado “processos”, dentro do diretório do seu programa, descompactados), colocá-los na fila de prontos, ordená-los conforme suas prioridades (arquivo “prioridades.txt”), e rodá-los usando o valor contido em “quantum.txt” como tamanho do quantum.

Em seu código, devem-se deixar explícitas as seguintes estruturas (com comentários e modularização adequada):

- BCP, contendo:
 - Contador de Programa
 - Estado do processo
 - Prioridade
 - Registradores de uso geral
 - Referência ao segmento de texto do programa
 - Possivelmente os créditos, embora estes, e somente estes, possam ser deixados em estrutura à parte
- Tabela de processos;
- Lista de processos prontos;
- Lista de processos bloqueados.

1.4 Situações Adversas

Algumas situações que podem ocorrer:

- E se, após descontado o crédito, um processo que acabou de rodar (primeiro da fila) tiver prioridade igual ao segundo da fila?

R: O que é mais custoso? Fazer uma troca completa de contexto, ou manter aquele que estava rodando antes? Embora nesse EP o custo seja idêntico, quero que analisem o que seria melhor, caso fosse real.

- E se, durante a execução, todos da fila de prontos ficarem com zero crédito, ainda existindo processos com crédito na fila de bloqueados (ou seja, impedindo a redistribuição)?

R: Nesse caso, a fila de prontos torna-se uma fila comum, até que os processos bloqueados voltem da fila de bloqueados (caso em que acabarão tendo prioridade naturalmente, por possuírem mais créditos). A redistribuição só é feita quando todos estiverem com zero crédito.

- SAIDA conta como um comando executado?

R: Sim. Ela deve entrar nas estatísticas.

- Quando conto E/S nas estatísticas, quando começa ou quando se completa?

R: No caso de E/S, a instrução entra para as estatísticas (número de instruções por quantum) quando iniciar, ou seja, imediatamente antes de bloquear.

- E se dois processo tiverem prioridades iguais quando inicia-se o programa?

R: Eles devem ficar em ordem alfabética pelo nome do arquivo que os contém.

- E se, ao ter seu crédito reduzido e tiver que ser reposicionado na fila de prontos, houver outros processos nessa fila com a mesma prioridade do que foi reduzido?

R: Coloque-o à frente destes (ele era mais importante antes).

1.5 Testes

Como forma de teste e avaliação do sistema, vocês devem gerar o arquivo de log para diferentes valores de `n_com` (pelo menos 10 valores diferentes, distribuídos de maneira uniforme em um intervalo que você julgue útil – lembre que há um número máximo de instruções em um programa), informando, para cada quantum definido:

- Número médio de trocas de processo, por processo.
- Número médio de instruções executadas por grupo de `n_com` (quantum).

Com base nesses dados, construam um relatório dizendo qual o valor de `n_com` que vocês consideram mais adequado, levando em conta o número de trocas do processo, bem como a relação entre o tamanho de `n_com` e a média de instruções executadas por quantum. Atenção!

Não tirem coelhos de cartolas! Fundamentem suas conclusões com base no comportamento do sistema; apresentem gráficos e tabelas para convencer o leitor do relatório de que sua escolha está coerente.

O relatório deve ser entregue em pdf.

1.6 Material para Entrega

A entrega será feita única e exclusivamente via Tidia. Você deve criar um arquivo “No_USP.zip” (em que No_USP é seu número USP) contendo o seguinte material:

- Logfiles gerados, organizados conforme o quantum definido pelo usuário (para cada quantum haverá um logfile diferente)
- Código java do programa
- Relatório de avaliação do sistema, em pdf, conforme descrito acima

Deve ser submetido um único arquivo por grupo (ou seja, um único integrante irá fazer a submissão), sendo que o relatório deverá conter o nome e número usp de **todos** os participantes.

1.7 Data de Entrega

O prazo de entrega é 01 de Outubro de 2017.

1.8 Observações Teóricas

- Note que em nenhum momento fala-se do contexto do escalonador. Isso porque, nesse trabalho, os registradores e demais recursos usados pelo escalonador estão transparentes ao sistema. Naturalmente, isso não corresponde à realidade em um processador, estando mais alinhado ao que ocorre em uma máquina virtual, em que o escalonador da máquina não se preocupa com seu próprio contexto, tratando tão somente de gerenciar o contexto dos processos que nele rodam.