UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL – UFMS
FACULDADE DE COMPUTAÇÃO – FACOM

Diego Takaki Matsubara

RELATÓRIO 2

Campo Grande, 2017

Diego Takaki Matsubara

RELATÓRIO 2

Relatório técnico apresentado como requisito parcial para obtenção de aprovação na disciplina Sistemas Operacionais, no Curso de Sistemas de Informação, na Universidade Federal de Mato Grosso do Sul.

Prof. Dr. Irineu Sotoma.

Campo Grande, 2017

Introdução - Motivação e objetivos

O presente relatório tem como objetivo resolver o problema descrito no trabalho 2 de Sistemas Operacionais de simulação de escalonamento de Threads por uma CPU.

O trabalho tem como requisito principal a utilização de Threads e a sincronização delas por meio primitivo, na linguagem Java. A motivação utilizada é a de resolver o problema de forma mais simples possível, sem precisar criar meios que consomem muitos recursos, somente o que for requisitado. Além disso, é vista a necessidade de criação de algumas controladoras de Threads, tudo isso controlado pela classe Scheduler, a classe principal de gerenciamento de threads.

O objetivo foi alcançado com sucesso, aplicando todas as entradas na formatação requisitada pelos objetivos do trabalho. A medida de tempo utilizada é em milissegundos, que está descrito nas especificações do trabalho.

Desenvolvimento

Neste trabalho foram criadas 3 classes: SyncMain, Scheduler e Process. A classe Process foi estendida de Thread, para que sejam tratadas como Threads. Foi escolhido essa maneira devido ao fato de que é utilizado um ArrayList de processos, impossibilitando o uso de Runnable.

Classe Process

A classe Process possui os seguintes atributos: estado, resta, prio, pId, inter, quantum e scheduler. Diferentemente do trabalho 1, as variáveis mantiveram os nomes que estão na descrição do trabalho, com exceção da variável pId e scheduler. A primeira representa o ID do processo, que o usuário precisa especificar no começo da execução da simulação, e scheduler, que é uma instância única do escalonador.

O método *run()*, que obrigatoriamente precisa ser implementado quando se utiliza Thread, foi implementado de maneira simples; Primeiramente é utilizado *Print*,

que mostra o status de criação da Thread. Após isso, as linhas 59 e 60 implementam o Thread.sleep(this.pId*this.inter), em que o parâmetro de sleep é um tempo descrito em milissegundos. Então, como especificado na descrição do trabalho, o tempo que o processo leva para chegar a fila de prontos é seu ID, multiplicado pelo tempo padrão de demora(inter). Quando a Thread conclui seu tempo de inatividade, ela vai emitir um comando para ser adicionado à lista de prontos. Além desse método, a classe possui um construtor para passar as informações vindas da classe principal, e também setters e getters para que seu acesso seja controlado. Deve-se ressaltar um comando especial, *finishedProcessing(),* que serve para zerar o resta e o prio do processo, para que ele possa sair da fila de prontos.

Classe Scheduler

Essa classe representa o escalonador. O escalonamento é feito com base na prioridade dos processos, e se existirem dois processos com prioridade igual, é escolhido o primeiro processo que chegou à fila de prontos, respeitando a ordem FIFO. Os seus atributos são os seguintes:

- 3 ArrayList de Processos; Onde*process* representa todos os processos que vão ser escalonados, *readyProcesses* é a fila de prontos, e *finishedProcesses* é a lista dos processos já encerrados.
- Os ints*interval* e *quantum*. interval representa inter na especificação do projeto, e quantum representa a variável de mesmo nome do detalhamento do projeto.
- Os booleanos *processExecuting*e *finishedProcessing*servem para indicar se tem um processo executando no momento, e se todos os processos já esgotaram seu resta, respectivamente.
- Por último, o Processo *highestPrio* é o processo selecionado pelo escalonador para ser executado

O construtor é criado com base em inter e quantum somente, deixando a parte de popular o array de processos com a classe SyncMain.

O método sincronizado addToReadyList é chamado por cada Thread *Process* separadamente. Por essa razão que existe somente uma instância de Scheduler.
O método execute é o que comanda o escalonamento. Começando com um while infinito que checa a condição de que todos os processos foram executados, um lock de sincronização é utilizado para que somente um processo possa adquirir o processo na posição 0 da lista de prontos (caso a lista só tenha um processo). Após isso, um outro loop infinito é inicializado para que somente um processo comece a executar. Em seguida, caso tenha mais processos na lista, um comparador é criado para que todas as *prio* sejam verificadas e o processo mais adequado seja escolhido. Com o processo pronto, seu status é alterado para executando, e a Thread é executada pelo tempo *quantum*. Após isso, é verificado se todo o resta foi utilizado ou não. Caso tenha sido, o processo é retirado da fila de prontos, inserido na fila de terminados, seu estado é alterado e o loop continua com o próximo processo com prioridade alta. Caso contrário, a sua prioridade e resta são decrementados, e a lista continua circularmente, e o *processExecuting* é liberado para o próximo processo. Finalmente, o array de processos terminados é verificado pela quantidade de elementos, e quanto atinge o mesmo número que a quantidade de processos existentes, o laço é encerrado e a simulação também.

Classe SyncMain

A classe SyncMain é o ponto de entrada da simulação. Os atributos do projeto são pedidos conforme especificação do projeto. Um scan é feito pedindo os atributos na seguinte ordem: *num, inter, quantidade, quantum* e para cara ID(*pId*), a *prio* é determinada.

Em seguida, o número *num* de processos são criados, e executados.

O método público getTime() retorna o horário atual, para ser usado nos logs da execução do projeto.

Conclusão

A maior dificuldade apresentada foi executar o próximo processo após o primeiro processo executar. Antes da conclusão do trabalho, o programa entrava numa condição que não podia ser satisfeita. Essa condição tinha como causa a concorrência de threads chegando na fila de prontos. Quando uma Thread chegava a fila de prontos, ela não era imediatamente executada. Isso foi resolvido utilizando synchronized. Quando há muitas Threads pedindo o mesmo recurso, ao mesmo tempo, por mais que locks ou barreiras de entradas para a seção crítica sejam implementadas, muitas vezes elas não conseguem conter as várias Threads acessando a parte crítica do sistema. As timestamps mostraram isso nos testes. No mesmo instante de tempo várias Threads conseguiam acessar a seção crítica de uma classe, atrapalhando na organização da fila de processos.

Apesar do ponto de grande dificuldade, o resto da implementação foi considerado fácil. Todos os outros pontos relativos a criação da lógica, seção crítica e ordem de escalonamento foram concluídos sem problemas. Em relação ao primeiro trabalho apresentado, a dificuldade foi considerada menor. O problema proposto no trabalho 2 tinha um grande embasamento, que ia desde o conceito das Threads, como utilizá-las até a melhor maneira de fazer a seção crítica. Isso tudo contribuiu para que o trabalho 2 fosse executado com maior facilidade.