**package** **caoss**.**simulator**.**os**.**scheduling**;

**import** java.util.Queue;

**import** java.util.concurrent.LinkedBlockingQueue;

**import** caoss.simulator.Program;

**import** caoss.simulator.configuration.Hardware;

**import** caoss.simulator.hardware.DeviceId;

**import** caoss.simulator.hardware.Timer;

**import** caoss.simulator.os.Dispatcher;

**import** caoss.simulator.os.Logger;

**import** caoss.simulator.os.ProcessControlBlock;

**import** caoss.simulator.os.Scheduler;

**public** **class** **FSOScheduler** **implements** **Scheduler**<SchedulingState> {

**private** **static** **final** **int** ***QUANTUM*** = 10;

**private** **static** **final** **int** ***QUOTA*** = 20;

**private** **final** Queue<**ProcessControlBlock**<SchedulingState>>[] readyQueues;

**private** **final** Queue<**ProcessControlBlock**<SchedulingState>> blockedQueue;

**private** **ProcessControlBlock**<SchedulingState> running; // just one CPU

**private** **final** **Timer** timer = (**Timer**) Hardware.devices.get(DeviceId.TIMER);

***@SuppressWarnings***("unchecked")

**public** **FSOScheduler**() {

**this**.readyQueues = (Queue<ProcessControlBlock<SchedulingState>>[]) **new** LinkedBlockingQueue[2];

**this**.blockedQueue = **new** LinkedBlockingQueue<ProcessControlBlock<SchedulingState>>();

**for** (**int** **i** = 0; i < 2; i++)

**this**.readyQueues[i] = **new** LinkedBlockingQueue<ProcessControlBlock<SchedulingState>>();

}

/\*\*

\* Metodo que cria um processo para executar o programa

\*

\* **@param** program

\*/

***@Override***

**public** **synchronized** **void** **newProcess**(**Program** program) {

**ProcessControlBlock**<SchedulingState> **pcb** = **new** ProcessControlBlock<SchedulingState>(program,

**new** **SchedulingState**(0, ***QUOTA***));

Logger.info("Create process " + pcb.pid + " to run program " + program.getFileName());

**if** (running == **null**) {

**this**.dispatch(pcb, ***QUANTUM***);

} **else** {

**this**.readyQueues[0].add(pcb);

}

logQueues();

}

/\*\*

\* E invocado quando o processo que esta sendo executado na CPU solicita uma

\* operacao de entrada / saida.

\*

\* **@param** pcb

\*/

***@Override***

**public** **synchronized** **void** **ioRequest**(**ProcessControlBlock**<SchedulingState> pcb) {

Logger.info("Process " + pcb.pid + ": IO request");

quota -= quantum;

// decrementar a quota ao quantum

// bloquear o pcb (processo)

**this**.readyQueue.poll(); // =????????? porque e aqui nao pomos dispatch

// adicionar a lista dos blocked

**this**.blockedQueue.add(pcb);

chooseNext();

logQueues();

}

/\*\*

\* Este e chamado quando a operacao de entrada / saida solicitada pelo processo

\* termina. nivel zero e a prioridade maxima e 1 a prioridade minima

\*

\* **@param** pcb

\*/

***@Override***

**public** **synchronized** **void** **ioConcluded**(**ProcessControlBlock**<SchedulingState> pcb) {

Logger.info("Process " + pcb.pid + ": IO concluded");

**int** **level** = pcb.getLevel();

// desbloquear o pcb (remover dos blocked)

**this**.blockedQueue.remove(pcb);

// verificar o nivel e se for 1 mudar para zero (high prioridade)

**if** (level == 1) {

level = 0;

// adicionar a lista readyQueue[0].add(pcb)

**this**.readyQueues[0].add(pcb);

} **else** **if** ((QUANTUM - QUOTA) <= 0) {

// ((((ou seja,caso sejam apenas fornecidos ao sistema p.e 7 clock ticks,

// enquanto qlq processo tem 20 clock ticks (QUOTA), e o time slice (QUANTUM)

// tem 10 clock ticks, a execução do processo tera de acabar mais cedo, yah

**this**.quantumExpired(pcb);

}

// apenas quando nenhum processo esta a correr pode acontecer este chooseNext()

**if** (running == **null**) {

chooseNext();

}

logQueues();

}

/\*\*

\* Esse metodo e invocado quando o processo em execucao na CPU esgota seu

\* quantum (intervalo de tempo).

\*

\* **@param** pcb

\*/

***@Override***

**public** **synchronized** **void** **quantumExpired**(**ProcessControlBlock**<SchedulingState> pcb) {

Logger.info("Process " + pcb.pid + ": quantum expired");

// verificar o resultado da subtracao que obtens quando retiras a quota ao

// quantum (time-slice)

**int** **level** = pcb.getSchudulingState().getState();

**int** **quota** = ***QUANTUM*** - ***QUOTA***;

**int** **comm** = quota;

**if** (quota <= 0)

comm = 1;

**else** // troquei os numeros

comm = 0;

// verificar com if (switch) o nivel do process, 0 ou 1, dependendo de ser high

// ou low priority

**switch** (comm) {

**case** 0: {

// no entanto #4, obviamente que, caso a quota nao tenha sido esgotada, teras de

// estabelecer o resultado da subtracao como a nova quota.

// no entanto #2, caso nao tenha esgotado a quota adicionar a high queue, ou

// seja, manter.

**if** (level == 1) {

level = 0;

**this**.readyQueues[0].add(pcb); // nao temos que remover da lista 1??????

}

QUOTA = quota;

// high

**break**;

}

**case** 1:

Logger.info("Process " + pcb.pid + ": quota expired");

// no entanto #3, e se o processo cujo quantum expirou pertencia antes de tal

// fenomeno ter acontecido a low priority queue? adicionas a low priority queue

// Mas caso tenha

// acontecido quota <= 0, definir a quota como QUOTA(20)

**if** (level == 0) {

level = 1;

**this**.readyQueues[1].add(pcb);

pcb.getQuota() = QUOTA(20);

}

// no entanto, tambem e necessario alterar o level do processo caso ele tenha

// estado na high queue e tenhas posto na low queue ou seja, 0 ---> 1,

// Vais verificar se o resultado e 1, ou seja, se a quota dada ao processo

// esgotou e caso tal tenha acontecido, adicionar a low priority queue

// low

**default**:

**break**;

}

chooseNext();

logQueues();

}

/\*\*

\*

\* E chamado quando o processo concluiu sua execucao

\*

\* **@param** pcb

\*/

***@Override***

**public** **synchronized** **void** **processConcluded**(**ProcessControlBlock**<SchedulingState> pcb) {

Logger.info("Process " + pcb.pid + ": execution concluded");

**long** **turnarround** = Hardware.clock.getTime() - pcb.arrivalTime;

Logger.info("Process " + pcb.pid + ": turnarround time: " + turnarround);

// verificar se o chocks ticks for menor a 10

**if** (turnarround < 10) {

time.set(10);

}

chooseNext();

logQueues();

}

**private** **void** **chooseNext**() {

Queue<**ProcessControlBlock**<SchedulingState>> **queue** = **this**.readyQueues[0];

**while** (queue.size > 0) {

**this**.dispatch(pcb, quantum);

**return**;

}

**this**.dispatch(**null**, 0);

// NAO FAZER BOOLEAN NOME = A UMA BOOLEAN PQ? PQ O PEDRO FEZ ASSIM YEY

// se a high priority queue estiver cheia, dispatch da high queue, caso nao

// seja o caso e a low priority esteja nao vazia, dispatch da low. No fim do

// casos da dispatch da null ------> dispatch(null, 0)

}

/\*\*

\* Proximo processo a ser escolhido

\*

\* **@param** pcb

\* **@param** quantum

\*/

**private** **void** **dispatch**(**ProcessControlBlock**<SchedulingState> pcb, **int** quantum) {

Dispatcher.dispatch(pcb);

running = pcb;

**if** (pcb != **null**) {

**SchedulingState** **state** = pcb.getSchedulingState();

state.setSchedulingTime(Hardware.clock.getTime());

timer.set(quantum);

Logger.info("Run process " + pcb.pid + " (quantum=" + quantum + ", quota=" + state.getQuota() + ")");

} **else**

timer.set(0);

}

/\*\*

\* Todos os processos que aguardam operaçoes de entrada / saida devem ser

\* colocados na fila bloqueada, para que seu status seja impresso pelo metodo

\* logQueues

\*/

**private** **void** **logQueues**() {

**int** **i** = 0;

**for** (Queue<**ProcessControlBlock**<SchedulingState>> **queue** : **this**.readyQueues) {

Logger.info("Queue " + i + ": " + queue);

i++;

}

Logger.info("Blocked " + blockedQueue);

}

}