UNIVERSIDADE DE ÉVORA

Departamento de Informática
Licenciatura em Engenharia Informática

- Gestão de processos: Modelo de cinco estados -

(Cadeira de Sistemas Operativos I)
2ºAno

Eduardo Simão Ramos Almeida Costa Martins, 29035
Daniel Gonçalo De Jesus Ramos, 29423
Marcus Vinicius Coelho Santos, 29764

Évora 2013

Introdução

Este trabalho tem como objectivo criar um simulador de um modelo de cinco estados (figura 1) para sistemas operativos. Um modelo de estados permite fazer o escalonamento de processos, em que o objectivo é por um processo a correr no processador de forma eficiente e de tempo em tempo ser interrompido enquanto aguarda pela realização de outras tarefas. Durante esse tempo o sistema operativo pode seleccionar outro processo a ser executado, evitando assim grandes períodos de tempo morto em que o processador não faz nada.

É o sistema operativo que é responsável pelo controlo da execução dos processos, logo este precisa de saber como seguir um processo e necessita de uma estrutura que contenha os dados relativos a um processo. É o PCB que irá conter toda a informação necessária acerca de um processo de forma a manter a consistência dos dados.

Uma descrição com maior detalhe acerca da implementação deste modelo será feita mais a frente neste relatório.

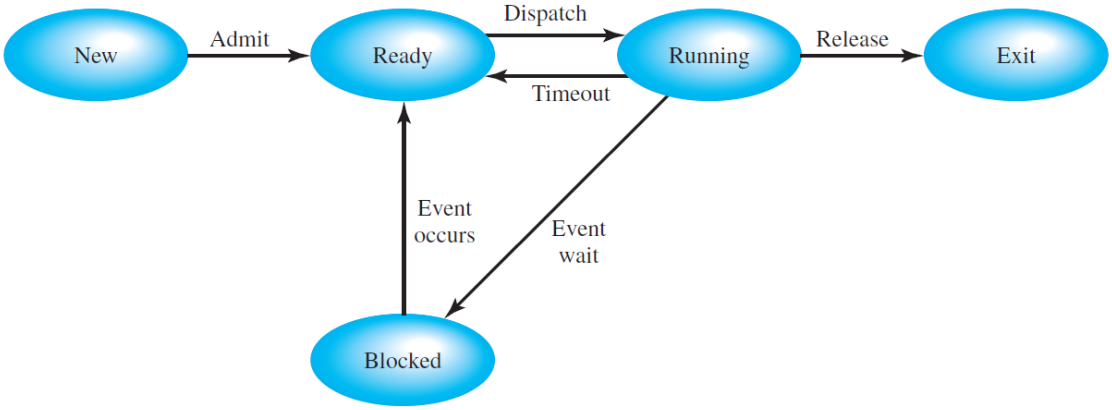


Figura 1: Modelo de cinco estados

Desenvolvimento

Para uma melhor compreensão do que cada estado do processo significa, segue uma descrição pormenorizada dos cinco estados que fazem parte deste modelo.

- **Estado New:** Significa que o processo acabou de ser criado mas que ainda não foi admitido para ser executado.
- **Estado Ready:** Significa que o processo está pronto a ser executado, quando houver oportunidade.
- **Estado Running:** Significa que o processo está a correr no processador.
- **Estado Exit:** Significa que um processo terminou a sua tarefa e está pronto a ser destruído.
- **Estado Blocked:** Significa que o processo aguarda por um evento, como por exemplo uma operação I/O.

Implementação dos vários estados e do simulador

Para implementar cada estado do modelo, cada estado representa uma classe única que conta com uma super classe abstracta chamada **PipelineStage** que é comum em todos os estados.

Para aplicar a lógica de cada estado, o método **update** é chamado em cada ciclo.

O incremento de ciclos e a admissão de novos processos são controlados pela classe Pipeline onde o método run verifica se há processos para serem admitidos e para passarem ao estado NEW. É ainda nesta classe que é ainda chamado o método **update** de cada estado para que os processos possam chegar ao processador, etc.

Lógica do estado new

No estado new, as seguintes verificações são efectuadas sobre o processo que está na frente da fila, caso este exista:

1. Verificar se existe processos na fila e verificar se existe menos de dois processos no estado ready.
2. Verificar se o processo está no ciclo correcto, isto é, o ciclo actual não é o mesmo que o ciclo do estado anterior.
3. Caso passe nas verificações, o processo é removido da fila, é actualizado o seu estado para o estado ready, é actualizado o valor do ciclo actual e é enviado para a fila do estado ready.

Lógica do estado ready

No estado ready, as seguintes verificações são efectuadas sobre o processo que está na frente da fila, caso este exista:

1. Actualizar o tempo de espera dos processos em fila.
2. Verificar se existe processos na fila
3. Verificar se o processo está no ciclo correcto, isto é, o ciclo actual não é o mesmo que o ciclo do estado anterior.
4. Verificar se não existe processos a correr no processador.
5. Caso passe nas verificações, o processo é removido da fila, é actualizado o seu estado para o estado running, é actualizado o valor do ciclo actual e é enviado para a fila do estado running, que neste caso só corre um processo de cada vez admitindo que o processador é single core.

Lógica do estado exit

1. Verificar se todos os processos estão no ciclo correcto, isto é, o ciclo actual não é o mesmo que o ciclo do estado anterior.
2. Por cada processo pronto a ser destruído, incrementar um contador.
3. Se o contador for igual ao total de processos que foram admitidos, então o simulador termina.

Lógica do estado blocked

1. Percorrer as filas associadas a cada dispositivo.
2. Verificar se existem processos na fila.
3. Verificar se o processo está no ciclo correcto, isto é, o ciclo actual não é o mesmo que o ciclo do estado anterior.
4. Chamar o método run do dispositivo para que este continue a processar o pedido.
5. Se o dispositivo acabou de processar o pedido para o processo na frente da fila, o processo é removido da fila, é actualizado o seu estado para o estado ready, é actualizado o valor do ciclo actual e é enviado para a fila do estado ready.

Lógica do estado running

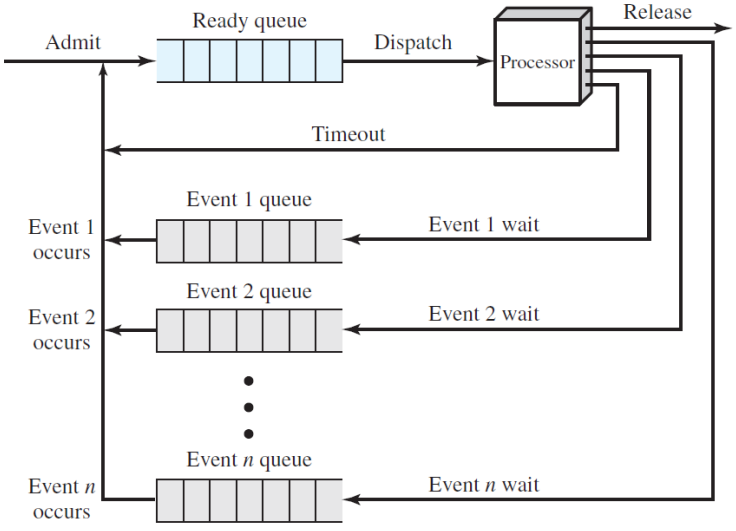
1. Verificar se existe algum processo na fila.
2. Verificar se o processo está no ciclo correcto, isto é, o ciclo actual não é o mesmo que o ciclo do estado anterior.
3. Caso o processo esteja pronto a ser executado, incrementar o número de ciclos de cpu para este processo, actualizar o valor do ciclo actual e decrementar o numero de ciclos restantes que precisam de ser processados.
4. Caso o numero de ciclos restantes que precisam de ser processados seja igual a zero, incrementa a posição dos dados seguintes no PCB.
5. Se o processo chegou ao fim, é actualizado o seu estado para o estado exit, é actualizado o valor do ciclo actual, é feito um reset aos ciclos do cpu e é enviado para a fila do estado exit.

6. Se o processo é interrompido por um evento, é actualizado o seu estado para o estado blocked, é actualizado o valor do ciclo actual, é feito um reset aos ciclos do cpu e é enviado para a fila do estado blocked
7. Caso ainda exista ciclos a serem processados, é actualizado o seu estado para o estado ready, é actualizado o valor do ciclo actual, é feito um reset aos ciclos do cpu e é enviado para a fila do estado ready

Implementação das filas

Cada estado do modelo tem uma fila de espera onde os processos aguardam. O estado blocked é a única excepção, onde este pode conter múltiplas filas, uma por dispositivo (figura 2).

As filas são implementadas usando listas duplamente ligadas (linked lists) na classe abstracta **PipelineStage** e tem um funcionamento do tipo FIFO (First In First Out), sendo que o processo que está a frente da fila tem prioridade sobre os outros processos que estão no fim da fila.



(b) Multiple blocked queues

Figura 2: Modelo para filas de espera

Algoritmo de escalonamento

O algoritmo de escalonamento implementado neste modelo é o algoritmo Round Robin (RR) com um quantum de quatro ciclos em que cada processo é executado durante 4 ciclos no processador e depois volta para a lista de espera a não ser que esteja a espera de um evento I/O, caso tal aconteça, fica na fila do estado blocked.

Estrutura do PCB

Os dados de cada PCB estão contidos numa classe chamada PCB para que o simulador consiga "executar" o processo. Uma representação da estructura do PCB está indicada na figura 3.

PCB
Pid
ArrayPosition
CurrentPipelineCycle
TempoDeEspera
State
Dados

Figura 3: Estrutura do PCB

- **Pid:** Contem o id do processo.
- **ArrayPosition:** Contem a posição actual dos dados que estão a ser processados.
- **CurrentPipeLineCycle:** Contem o valor do ultimo ciclo em que este processo teve alguma actividade.
- **State:** Contem o estado actual do processo
- **Dados:** Contem os dados do processo, como por exemplo o tempo de serviço e o acesso aos dispositivos.

Estrutura do programa

Classes implementadas

Class AbstractPCB

Classe abstracta para o pcb. Contem a geracao do pid.

Modifier and Type	Method and Description
`abstract int`	`getPID()` Obtem o id do processo.
`abstract int`	`getState()` Obtem o estado do processo.
`abstract void`	`setState(int value)` Atribui o estado do processo.

Class Cpu_scheduler

Modifier and Type	Method and Description
`static void`	`main(java.lang.String[] args)`

Class Device

Contem metodos que permitem manipular um dispositivo.

Modifier and Type	Method and Description
`boolean`	`isFinished()` Verifica se o pedido foi concluido.
`void`	`resetDeviceCycles()` Faz um reset aos ciclos do dispositivo para o valor inicial.
`void`	`run()` actualiza a quantidade de ciclos que este dispositivo ja processou para um processo que esta no stage blocked a espera.

Class FileInput

Le o conteudo de um ficheiro que contem os dados a serem processados pelo simulador.

Modifier and Type	Method and Description
void	carregarDados() Carrega os dados lidos para a pipeline.
int[]	getDadosOfProcess(int n) Retorna os dados associados a um processo n.
int	getNumberOfDevices() Retorna o numero de dispositivos.
int	getNumberOfProcess() Retorna o numero de processos.
int	getTimeOfDevice(int dev) Retorna o tempo de cada dispositivo dev varia entre 0 e o numero de dispositivos - 1, inclusive
int	getTimeProcess(int n) Retorna o tempo do processo quando chega ao stage NEW o valor de n varia entre 1 e o numero de dispositivos, inclusive caso contrario retorna 0

Class PCB

Contem os dados de um processo.

Modifier and Type	Method and Description
`void`	`decreaseRemainingCycles()` Reduz a quantidade de ciclos que precisam de ser processados pelo cpu.
`int`	`getCreationTime()` Obtem o tempo em que o processo e criado.
`int`	`getCurrentCycles()` Obtem os ciclos que precisam ainda de ser processados pelo cpu.
`int`	`getCurrentPipelineCycle()` Obtem o ultimo ciclo em que ouve actividade no processo.
`int[]`	`getDados()` Obtem os dados a serem processados.

int	getDevice() Obtem o id do dispositivo.
int	getPID() Obtem o id do processo.
int	getState() Obtem o estado do processo.
int	getTempoDeEspera() Obtem o tempo de espera no estado ready.
boolean	hasCPUData() Verifica se o pcb contem dados de cpu actualmente.
boolean	hasDeviceData() Verifica se o pcb contem dados de um dispositivo actualmente.
boolean	isFinished() Verifica se o processonão contem mais dados a serem processados.
boolean	isOdd(int i) Verifica se é par ou impar.
void	nextPCBData() Obtem os dados seguintes a serem processados.
void	setCreationTime(int creationTime) Atribui o tempo em que o processo e criado.
void	setCurrentPipelineCycle(int currentPipelineCycle) Atribui o ultimo ciclo em que ouve actividade no processo.
void	setDados(int[] dados) Atribui os dados a serem processados.
void	setState(int value) Atribui o estado do processo.
void	setTempoDeEspera(int tempoDeEspera) Atribui o tempo de espera no estado ready.
String	toString()

Class Pipeline

Classe responsavel por actualizar os varios stages e incrementar o numero de ciclos.

Modifier and Type	Method and Description
`void`	`queueProcessCreation(PCB pcb)` Adiciona um processo para ser admitido.
`void`	`run()` Permite executar a logica da pipeline.
`java.lang.String`	`toString()`

Class PipelineStage

Classe abstracta que contem metodos partilhados em todos os stages da pipeline.

Modifier and Type	Method and Description
`void`	`addProcessToQueue(PCB process)` Adiciona um processo a lista de espera.
`void`	`addProcessToQueue(PCB process, int device)` Adiciona um processo a lista de espera de um dispositivo.
`int`	`getDeviceQueueSize(int device)` Obtem o tamanho da fila para um dispositivo.
`int`	`getQueueSize()` Obtem o tamanho da lista de processos.
`java.lang.String`	`outputProcessStates()` Faz o output do estado dos processos.
`java.lang.String`	`outputProcessWaitTime()` Faz o output do tempo de espera no estado ready de cada processo.
`abstract void`	`update(int cycles)` Actualiza e gere os processos que estao em fila de espera no stage.

Class StageBlocked

Gere os processos do stage blocked.

Modifier and Type	Method and Description
`void`	`setupDevices()`
`void`	`update(int cycles)` Actualiza e gere os processos que estao em fila de espera no stage.

Class StageExit

Gere os processos do stage exit.

Modifier and Type	Method and Description
`void`	`update(int cycles)` Actualiza e gere os processos que estao em fila de espera no stage.

Class StageNew

Gere os processos do stage new.

Modifier and Type	Method and Description
`void`	`update(int cycles)` Actualiza e gere os processos que estao em fila de espera no stage.

Class StageReady

Gere os processos do stage ready.

Modifier and Type	Method and Description
`void`	`update(int cycles)` Actualiza e gere os processos que estao em fila de espera no stage.

Class StageRun

Gere os processos do stage run.

Modifier and Type	Method and Description
`int`	`inc(int i)`
`void`	`resetCPUCycles()` Faz um reset aos ciclos do cpu.
`void`	`scheduler(int cycles)` CPU scheduler.
`void`	`update(int cycles)` Actualiza e gere os processos que estao em fila de espera no stage.

Conclusão

Ouve alguma dificuldade em perceber se o simulador estava a funcionar como previsto ou não, de modo que provavelmente poderá haver erros no número de ciclos necessários para executar os processos. Também devido a falta de tempo não foi possível fazer mais testes.

Bibliografia

1. Operating Systems - Internals and Design Principles 7th ed - W. Stallings

Anexos

Anexo 1: Código fonte

AbstractPCB

```
package cpu_scheduler;


/**
 * Classe abstracta para o pcb.
 * Contem a geracao do pid.
 */
public abstract class AbstractPCB
{
        public static int pidNumber = 0;


    /**
     * Obtem o id do processo.
     * @return
     */
        public abstract int getPID();


    /**
     * Obtem o estado do processo.
     * @return
     */
        public abstract int getState();


    /**
     * Atribui o estado do processo.
     * @param value
```

```
     */
        public abstract void setState(int value);
}
```

Class Cpu_scheduler

```
package cpu_scheduler;

import java.util.logging.Level;
import java.util.logging.Logger;

public class Cpu_scheduler {
   public static boolean running = true;
   public static Pipeline pipeline;



   public static void main(String[] args)
   {
      try
      {
         pipeline = new Pipeline();
         FileInput file = new FileInput("dados.cpu");
         file.carregarDados();

         //executar a logica da pipeline
         while(running)
         {
            pipeline.run();
         }
      }
```

```
        catch (Exception ex)
        {
            Logger.getLogger(
                    Cpu_scheduler.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Class Device

```
package cpu_scheduler;


/**
 * Contem metodos que permitem manipular um dispositivo.
 *
 */
public class Device
{
    public static int numOfCycles;
    private int currentCycle;

    public Device(int time)
    {
        numOfCycles = time;
        currentCycle = 0;
    }
```

```
/**
 * actualiza a quantidade de ciclos que este dispositivo ja processou
 * para um processo que esta no stage blocked a espera.
 */
public void run()
{
   if(currentCycle <= numOfCycles)
   {
      currentCycle++;
   }
}


/**
 * Faz um reset aos ciclos do dispositivo para o valor inicial.
 */
public void resetDeviceCycles()
{
   currentCycle = 0;
}


/**
 * Verifica se o pedido foi concluido.
 * Neste caso o pedido e concluido quando o numero de ciclos processados
 * e igual ao numero de ciclos por pedido do dispositivo.
 * @return
 */
public boolean isFinished()
{
```

```
        return currentCycle == numOfCycles;
    }
}
```

Class FileInput

```
package cpu_scheduler;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

/**
 * Le o conteudo de um ficheiro que contem os dados
 * a serem processados pelo simulador.
 */
public class FileInput
{
    FileReader file;
    BufferedReader br;
    ArrayList<String> dados = new ArrayList<String>();

    public FileInput(String caminho) throws IOException
    {
        this.file = new FileReader(caminho);
        this.br = new BufferedReader(file);
        init();
    }
```

```
/**
 * Inicializa a leitura do ficheiro, lendo o conteudo inteiro
 * do ficheiro.
 * @throws IOException
 */
private void init() throws IOException
{
   String out = "";
   while((out = br.readLine()) != null)
   {
      dados.add(out);
   }
}

/**
 * Retorna o numero de processos.
 * @return
 */
public int getNumberOfProcess()
{
   String lineOne = dados.get(0);
   String[] inf = lineOne.split(" ");
   return Integer.valueOf(inf[0]);
}

/**
 * Retorna o numero de dispositivos.
```

```
 * @return
 */
public int getNumberOfDevices()
{
   String lineOne = dados.get(0);
   String[] inf = lineOne.split(" ");
   return Integer.valueOf(inf[1]);
}

/**
 * Retorna o tempo de cada dispositivo
 * dev varia entre 0 e o numero de dispositivos - 1, inclusive
 * @param dev
 * @return
 * @throws Exception
 */
public int getTimeOfDevice(int dev) throws Exception
{
   if(dev < 0 || dev >= getNumberOfDevices())
      throw new Exception();

   int startAtIndex = 2;
   int index = dev + startAtIndex;

   String lineOne = dados.get(0);
   String[] devicesTime = lineOne.split(" ");
   return Integer.valueOf(devicesTime[index]);
}
```

```
/**
 * Retorna o tempo do processo quando chega ao stage NEW
 * o valor de n varia entre 1 e o numero de dispositivos, inclusive
 * caso contrario retorna 0
 * @param n
 * @return
 */
public int getTimeProcess(int n)
{
   String aux = dados.get(n);
   String[] time = aux.split(" ");
   return Integer.valueOf(time[0]);
}

/**
 * Retorna os dados associados a um processo n.
 * @param n
 * @return
 */
public int[] getDadosOfProcess(int n)
{
   n++;
   String aux = this.dados.get(n);
   String[] dados = aux.split(" ");
   int[] result = new int[dados.length];
   for(int i = 0; i < result.length; i++){
         result[i] = Integer.valueOf(dados[i]);
```

```
    }
    return result;
}

/**
 * Carrega os dados lidos para a pipeline.
 * @throws Exception
 */
public void carregarDados() throws Exception
{
    int numP = getNumberOfProcess();
    Pipeline.numProcesses = numP;
    int numD = getNumberOfDevices();
    Pipeline.numDevices = numD;
    StageBlocked stage =
          (StageBlocked) Pipeline.pipelineStages[
          PipelineI.PIPELINE_STAGE_BLOCKED];

    stage.setupDevices();
    for(int i = 0; i < numP; i++)
    {
       int dados[] = this.getDadosOfProcess(i);
       PCB pcb = new PCB();
       pcb.setDados(dados);
       Cpu_scheduler.pipeline.queueProcessCreation(pcb);
    }

    for(int i = 0; i< numD; i++)
```

```
        {
            int time_device = getTimeOfDevice(i);
            System.out.println("device "+i+", time: "+time_device);
            Device device = new Device(time_device);
            System.out.println("device: "+device);
            StageBlocked.devices.add(device);
        }
    }
}
```

Class PCB

```
package cpu_scheduler;

/**
 * Contem os dados de um processo.
 *
 */
public class PCB extends AbstractPCB
{
    private int pid;
    private int arrayPosition;
    private int currentPipelineCycle;
    private int tempoDeEspera;
    private int state = PipelineI.PROCESS_STATE_NOT_CREATED;
    int[] dados;

    public PCB()
```

```
{
   this.pid = PCB.pidNumber++;
   this.arrayPosition = 1;
   this.currentPipelineCycle = 0;
   this.tempoDeEspera = 0;
}

public PCB(int state)
{
   this.pid = PCB.pidNumber++;
   this.state = state;
   this.arrayPosition = 1;
   this.currentPipelineCycle = 0;
}

/**
 * Obtem o ultimo ciclo em que ouve actividade no processo.
 * @return
 */
public int getCurrentPipelineCycle()
{
   return currentPipelineCycle;
}

/**
 * Atribui o ultimo ciclo em que ouve actividade no processo.
 * @param currentPipelineCycle
 */
```

```
public void setCurrentPipelineCycle(int currentPipelineCycle)
{
   this.currentPipelineCycle = currentPipelineCycle;
}

/**
 * Obtem o tempo de espera no estado ready.
 * @return
 */
public int getTempoDeEspera()
{
   return tempoDeEspera;
}

/**
 * Atribui o tempo de espera no estado ready.
 * @param tempoDeEspera
 */
public void setTempoDeEspera(int tempoDeEspera)
{
   this.tempoDeEspera = tempoDeEspera;
}

/**
 * Obtem o id do processo.
 * @return
 */
@Override
```

```
public int getPID()
{
   return pid;
}

/**
 * Verifica se o processonão contem mais dados a serem processados.
 * @return
 */
public boolean isFinished()
{
   return arrayPosition >= dados.length;
}

/**
 * Obtem o estado do processo.
 * @return
 */
@Override
public int getState()
{
   return this.state;
}

/**
 * Atribui o estado do processo.
 * @param value
 */
```

```
@Override
public void setState(int value)
{
   this.state = value;
}

/**
 * Obtem os dados a serem processados.
 * @return
 */
public int[] getDados()
{
   return this.dados;
}

/**
 * Atribui os dados a serem processados.
 * @param dados
 */
public void setDados(int[] dados)
{
   this.dados = dados;
}

/**
 * Obtem o tempo em que o processo e criado.
 * @return
 */
```

```
public int getCreationTime()
{
   return dados[0];
}

/**
 * Atribui o tempo em que o processo e criado.
 * @param creationTime
 */
public void setCreationTime(int creationTime)
{
   this.dados[0] = creationTime;
}

/**
 * Obtem os ciclos que precisam ainda de ser processados pelo cpu.
 * @return
 */
public int getCurrentCycles()
{
   return dados[arrayPosition];
}

/**
 * Obtem os dados seguintes a serem processados.
 */
public void nextPCBData()
{
```

```
    arrayPosition++;
  }

  /**
   * Reduz a quantidade de ciclos que precisam de ser processados pelo cpu.
   * @throws ProcessException
   */
  public void decreaseRemainingCycles() throws ProcessException
  {
    if(dados[arrayPosition] == 0) { throw new
ProcessException(ProcessException.PROCESS_EXCEPTION_RUNTIME_ERROR); }
    dados[arrayPosition]--;
  }

  /**
   * Verifica se o pcb contem dados de cpu actualmente.
   * @return
   */
  public boolean hasCPUData()
  {
    return isOdd(arrayPosition);
  }

  /**
   * Verifica se o pcb contem dados de um dispositivo actualmente.
   * @return
   */
  public boolean hasDeviceData()
  {
```

```
    return !(isOdd(arrayPosition));
}

/**
 * Obtem o id do dispositivo.
 * @return
 * @throws Exception
 */
public int getDevice() throws Exception
{
    if(!hasDeviceData()) { throw new Exception(); }

    return dados[arrayPosition];
}

/**
 * Verifica se é par ou impar.
 * @param i
 * @return
 */
public boolean isOdd(int i)
{
    return i % 2 != 0 ? true : false;
}

@Override
public String toString()
{
```

```
        String result = "PID:"+this.pid+"\narrayPosition:"+this.arrayPosition+
                        "\ncreationTime:"+dados[0]+"\nState:"+this.state+"\nDados: ";
        for(int i = 0; i< dados.length; i++){
              result+= dados[i]+", ";
        }
        return result+"\n";
    }
}
```

Class Pipeline

```
package cpu_scheduler;
import java.io.IOException;
import java.util.Iterator;
import java.util.LinkedList;

/**
 * Classe responsavel por actualizar os varios stages e
 * incrementar o numero de ciclos.
 *
 */
public class Pipeline implements PipelineI
{
    public static int READY_STAGE_PROCESS_LIMIT = 2;
    public static int numProcesses = 0;
    public static int numDevices = 0;
    private int numCycles = 0;
    private LinkedList<PCB> preProcessCreationQueue;
```

```
public static PipelineStage[] pipelineStages;



public Pipeline()
{
   preProcessCreationQueue = new LinkedList<PCB>();



    //inicializar os diferentes estados do pipeline.
    //array com todos os stages com listas para as queues.
   pipelineStages = new PipelineStage[5];
   pipelineStages[0] = new StageNew(PipelineI.PIPELINE_STAGE_NEW);
   pipelineStages[1] = new StageReady(PipelineI.PIPELINE_STAGE_READY) ;
   pipelineStages[2] = new StageRun(PipelineI.PIPELINE_STAGE_RUN) ;
   pipelineStages[3] = new StageBlocked(PipelineI.PIPELINE_STAGE_BLOCKED);
   pipelineStages[4] = new StageExit(PipelineI.PIPELINE_STAGE_EXIT) ;
}

/**
 * Adiciona um processo para ser admitido.
 * @param pcb
 */
public void queueProcessCreation(PCB pcb)
{
   preProcessCreationQueue.add(pcb);
}

/**
```

```
 * Obtem a queue de processos que precisam de ser admitidos.
 * @return
 */
LinkedList<PCB> getQueue()
{
   return preProcessCreationQueue;
}

/**
 * Permite executar a logica da pipeline.
 * @throws Exception
 */
public void run() throws Exception
{
   //admissao de processos para o stage new
   for(Iterator<PCB> it = preProcessCreationQueue.iterator(); it.hasNext();)
   {
      PCB pcb = it.next();
      if(pcb.getCreationTime() == numCycles)
      {
         pcb.setState(PipelineI.PROCESS_STATE_NEW);
         pcb.setCurrentPipelineCycle(numCycles);
         pipelineStages[Pipeline.PIPELINE_STAGE_NEW].addProcessToQueue(pcb);
         it.remove();
      }
   }

   //actualizar cada stage com prioridades baseado nos processos que entram
```

```
    //no stage ready
    //BLOCKED -> NEW -> RUN
    //dar prioridade aos processos que estao nos dispositivos que queiram
    //passar para o stage ready
    //os processos que vem do stage new tem prioridade sobre os processos
    //que estao no CPU
    //os processos que saiem do CPU passam para o stage ready quando der
    //timeout.

    pipelineStages[PipelineI.PIPELINE_STAGE_BLOCKED].update(numCycles);
    pipelineStages[PipelineI.PIPELINE_STAGE_NEW].update(numCycles);
    pipelineStages[PipelineI.PIPELINE_STAGE_READY].update(numCycles);
    pipelineStages[PipelineI.PIPELINE_STAGE_RUN].update(numCycles);
    pipelineStages[PipelineI.PIPELINE_STAGE_EXIT].update(numCycles);

    System.out.print(this);
    numCycles++;
    //output dos processos por ciclo.

}

@Override
public String toString()
{
    String output = "--------------------------------\n";
    output += "Ciclo: " + numCycles + "\n";
    //output dos estados de cada processo.
    for(int i = 0; i < Pipeline.pipelineStages.length; i++)
```

```
        {
            output += Pipeline.pipelineStages[i].outputProcessStates();
        }
        //output do tempo de espera
        //output dos estados de cada processo.
        for(int i = 0; i < Pipeline.pipelineStages.length; i++)
        {
            output += Pipeline.pipelineStages[i].outputProcessWaitTime();
        }
         output += "--------------------------------";
        return output;
    }
}
```

Interface PipelineI

```
package cpu_scheduler;

public interface PipelineI {
    public static final int PROCESS_STATE_NOT_CREATED = -1;
    public static final int PROCESS_STATE_NEW = 0;
    public static final int PROCESS_STATE_READY = 1;
    public static final int PROCESS_STATE_RUN = 2;
    public static final int PROCESS_STATE_BLOCKED = 3;
    public static final int PROCESS_STATE_EXIT = 4;

    public static final int PIPELINE_STAGE_NEW = 0;
    public static final int PIPELINE_STAGE_READY = 1;
```

```
    public static final int PIPELINE_STAGE_RUN = 2;
    public static final int PIPELINE_STAGE_BLOCKED = 3;
    public static final int PIPELINE_STAGE_EXIT = 4;

    public static String strStage[] = { "NEW", "WAIT/READY", "RUN", "BLOCKED", "EXIT" };

    public void run() throws Exception;
}
```

Class PipelineStage

```
package cpu_scheduler;
import java.util.LinkedList;

/**
 * Classe abstracta que contem metodos partilhados
 * em todos os stages da pipeline.
 */
public abstract class PipelineStage {

    protected LinkedList<PCB> queue[];
    protected int pipelineStageType;

    public PipelineStage(int stageType)
    {
        pipelineStageType = stageType;
    }

    /**
```

```
 * Adiciona um processo a lista de espera.
 * @param process
 */
public void addProcessToQueue(PCB process)
{
   queue[0].add(process);
}

 /**
 * Adiciona um processo a lista de espera de um dispositivo.
 * Apenas pode ser usado no stage blocked.
 * @param process
 */
public void addProcessToQueue(PCB process, int device)
{
   queue[device - 1].add(process);
}

/**
 * Obtem o tamanho da lista de processos.
 * @return
 */
public int getQueueSize()
{
   return queue[0].size();
}

/**
```

```
     * Obtem o tamanho da fila para um dispositivo.
     * @param device
     * @return
     */
    public int getDeviceQueueSize(int device)
    {
       return queue[device].size();
    }

    /**
    * Actualiza e gere os processos que estao em fila de espera no stage.
    * passando-os para o stage seguinte quando for oportuno.
    * @param cycles
    */
    public abstract void update(int cycles) throws Exception;

    /**
     * Faz o output do estado dos processos.
     * @return
     */
    public String outputProcessStates()
    {
       String output = "";
       for(int i = 0; i < queue.length; i++)
       {
         for(PCB pcb : queue[i])
         {
            output += "Processo id=" + pcb.getPID() + " estado=" +
PipelineI.strStage[pcb.getState()] + " ";
```

```
            output += "\n";


          }
        }
        return output;
    }


    /**
     * Faz o output do tempo de espera no estado ready de cada processo.
     * @return
     */
    public String outputProcessWaitTime()
    {
        String output = "";
        for(int i = 0; i < queue.length; i++)
        {
          for(PCB pcb : queue[i])
          {
            output += "Processo id=" + pcb.getPID() + " wait cycles=" +
pcb.getTempoDeEspera()+ " ";
            output += "\n";
          }
        }
        return output;
    }
}
```

Class ProcessException

```
package cpu_scheduler;


class ProcessException extends Exception

{

   public static final String PROCESS_EXCEPTION_RUNTIME_ERROR = "Ocurreu um erro ao
correr o processo, tempo invalido";


   public ProcessException()

   {

      super();

   }


   public ProcessException(String s)

   {

      super(s);

   }

}


Class StageBlocked

package cpu_scheduler;


import java.util.LinkedList;


/**

 * Gere os processos do stage blocked.

 *

 */

public class StageBlocked extends PipelineStage

{
```

```
public static LinkedList<Device> devices;
public StageBlocked(int stageType)
{
   super(stageType);
}

public void setupDevices()
{
   queue = new LinkedList[Pipeline.numDevices];
   for(int i = 0; i < queue.length; i++)
   {
      queue[i] = new LinkedList<PCB>();
   }
   //criar a lista de dispositivos
   devices = new LinkedList<Device>();
}

/**
* Actualiza e gere os processos que estao em fila de espera no stage.
* passando-os para o stage seguinte quando for oportuno.
* @param cycles
*/
@Override
public void update(int cycles)
{
   //percorrer as filas dos dispositivos e verificar os processos
   //que estao na frente da fila.
   for(int i = 0; i < queue.length; i++)
```

```
      {
         //existe processos neste device
         if(queue[i].size() > 0)
         {
            PCB pcb = queue[i].getFirst();
            //processo esta no ciclo correcto
            if(pcb.getCurrentPipelineCycle() < cycles)
            {
               Device dev = devices.get(i);
               dev.run();
               if(dev.isFinished())
               {
                  pcb.setState(PipelineI.PROCESS_STATE_READY);
                              pcb.setCurrentPipelineCycle(cycles);

          Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_READY].addProcessToQueue(
queue[i].removeFirst() );
                  dev.resetDeviceCycles();
               }
            }
         }
      }
   }
}
```

Class StageExit

```
package cpu_scheduler;

import java.util.LinkedList;
```

```
/**
 * Gere os processos do stage exit.
 *
 */
public class StageExit extends PipelineStage
{
   public StageExit(int stageType)
   {
      super(stageType);
      queue = new LinkedList[1];
      queue[0] = new LinkedList<PCB>();
   }

   /**
   * Actualiza e gere os processos que estao em fila de espera no stage.
   * passando-os para o stage seguinte quando for oportuno.
   * @param cycles
   */
   @Override
   public void update(int cycles)
   {
      //terminar o simulador se todos os processos estiverem no estado exit.
      //percorrer todos os processos para ver se estao no ciclo correcto
      int numProcessesOnCorrectCycle = 0;
      for(PCB pcb : queue[0])
      {
         //processo esta a espera a pelo menos 1 ciclo
```

```
            if(pcb.getCurrentPipelineCycle() < cycles)
            {
                pcb.setCurrentPipelineCycle(cycles);
                numProcessesOnCorrectCycle++;
            }
        }

        if(numProcessesOnCorrectCycle == Pipeline.numProcesses)
        {
            Cpu_scheduler.running = false;
        }
    }
}
```

Class StageNew

```
package cpu_scheduler;

import java.util.LinkedList;

/**
 * Gere os processos do stage new.
 *
 */
public class StageNew extends PipelineStage
{
    public StageNew(int stageType)
    {
```

```
        super(stageType);
        queue = new LinkedList[1];
        queue[0] = new LinkedList<PCB>();
    }

    /**
     * Actualiza e gere os processos que estao em fila de espera no stage.
     * passando-os para o stage seguinte quando for oportuno.
     * @param cycles
     */
    @Override
    public void update(int cycles)
    {
        //existe processos
        //adicionar processos do stage new ao stage ready
        //se tiver 2 ou menos processos na queue do stage ready.
        if(queue[0].size() > 0 &&
Cpu_scheduler.pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_READY].getQueueSize()
<= Pipeline.READY_STAGE_PROCESS_LIMIT)
        {
            //apenas avancar o processo se for o ciclo seguinte
            if(queue[0].getFirst().getCurrentPipelineCycle() < cycles)
            {
                PCB Process = queue[0].removeFirst();
                Process.setCurrentPipelineCycle(cycles);
                Process.setState(PipelineI.PROCESS_STATE_READY);

Cpu_scheduler.pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_READY].addProcessTo
Queue( Process );
                Process = null;
```

```
        }
      }
    }
}
```

Class StageReady

```
package cpu_scheduler;

import java.util.LinkedList;

/**
 * Gere os processos do stage ready.
 *
 */
public class StageReady extends PipelineStage
{
   public StageReady(int stageType)
   {
      super(stageType);
      queue = new LinkedList[1];
      queue[0] = new LinkedList<PCB>();
   }

   /**
    * Actualiza e gere os processos que estao em fila de espera no stage.
    * passando-os para o stage seguinte quando for oportuno.
    * @param cycles
```

```
   */
   @Override
   public void update(int cycles)
   {
      //actualizar o tempo de espera de cada processo.
      for(PCB pcb : queue[0])
      {
         if(pcb.getCurrentPipelineCycle() < cycles)
         {
            pcb.setTempoDeEspera( pcb.getTempoDeEspera() + 1);
         }
      }

      //existe processos no estado ready, passar um processo para o cpu
      if(queue[0].size() > 0 && queue[0].get(0).getCurrentPipelineCycle() < cycles &&
            Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_RUN].getQueueSize() == 0)
      {
         PCB process = queue[0].removeFirst();
         process.setState(PipelineI.PROCESS_STATE_RUN);
         process.setCurrentPipelineCycle(cycles);

Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_RUN].addProcessToQueue(process);
      }
   }
}
```

Class StageRun

```
package cpu_scheduler;
```

```
import java.util.LinkedList;


/**
 * Gere os processos do stage run.
 *
 */
public class StageRun extends PipelineStage
{
   public static int CPU_CYCLES_PER_PROCESS = 4;
   private int currentProcessCyles = 0;



   public StageRun(int stageType)
   {
      super(stageType);
      queue = new LinkedList[1];
      queue[0] = new LinkedList<PCB>();
   }


   /**
    * Actualiza e gere os processos que estao em fila de espera no stage.
    * passando-os para o stage seguinte quando for oportuno.
    * @param cycles
    */
   @Override
   public void update(int cycles) throws Exception
   {
```

```
  scheduler(cycles);
}

/**
 * CPU scheduler.
 * @param cycles
 * @throws Exception
 */
public void scheduler(int cycles) throws Exception
{
   //fazer trabalho no cpu para o processo que esta na frente da fila se este
   //esta a espera ha um ciclo atras.
   if(this.queue[0].size() > 0 && queue[0].getFirst().getCurrentPipelineCycle() < cycles)
   {
      PCB process = queue[0].getFirst();
      //incrementar o tempo de cpu para este processo.
      currentProcessCyles = inc(currentProcessCyles);
      process.setCurrentPipelineCycle(cycles);
      process.decreaseRemainingCycles();

      //o processo ja nao tem ciclos de cpu para processar
      //envia-lo para o stage exit.
      if( process.getCurrentCycles() == 0)
      {
         process.nextPCBData();
         if(process.isFinished())
         {
            process.setState(PipelineI.PROCESS_STATE_EXIT);
```

```
Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_EXIT].addProcessToQueue(queue[0].r
emoveFirst());

            resetCPUCycles();

          }

          else if(process.hasDeviceData())

          {

            process.setState(PipelineI.PROCESS_STATE_BLOCKED);


Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_BLOCKED].addProcessToQueue(queue
[0].removeFirst(), process.getDevice());

            resetCPUCycles();

          }

        }

        //numero de ciclos de cpu para este processo chegou ao limite

        //envia-lo para o stage ready.

        else if(currentProcessCyles == CPU_CYCLES_PER_PROCESS)

        {

          resetCPUCycles();

          process.setState(PipelineI.PROCESS_STATE_READY);


Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_READY].addProcessToQueue(queue[0
].removeFirst());

        }

        else

        {

          //nao fazer nada porque o processo ainda tem ciclos de cpu por processar.

        }

      }

    }
```

```
    public int inc(int i)
    {
        return i + 1 % CPU_CYCLES_PER_PROCESS;
    }

    /**
     * Faz um reset aos ciclos do cpu.
     */
    public void resetCPUCycles()
    {
        currentProcessCyles = 0;
    }
}
```