UNIVERSIDADE DE ÉVORA

Departamento de Informática
Licenciatura em Engenharia Informática

- Gestão de processos e memória -

(Cadeira de Sistemas Operativos I)
2ºAno

Daniel Gonçalo De Jesus Ramos, 29423
Marcus Vinicius Coelho Santos, 29764

Évora 2013

ÍNDICE

ÍNDICE DE FIGURAS

ÌNDICE DE QUADROS

INTRODUÇÃO

Este trabalho tem como objectivo criar um simulador de um modelo de cinco estados (figura 1) para sistemas operativos com gestão de memória incorporada em que as instruções de execução de um processo são colocadas num espaço reservado em memória, em vez de estarem dentro do PCB.

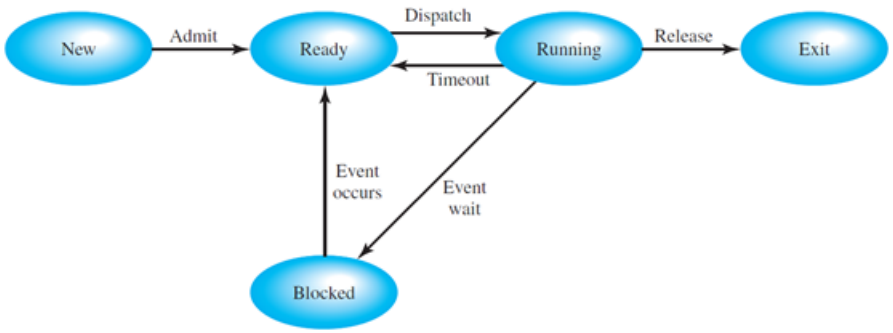

Figura 1: Modelo de cinco estados

A gestão de memória é feita com base nos algoritmos Best Fit e Worst Fit. O algoritmo Best Fit permite alocar memoria para um processo no espaço mais pequeno disponível, enquanto que o algoritmo Worst Fit permite alocar memoria para um processo no espaço maior disponível. A implementação da gestão do espaço livre e ocupado será explicada mais a frente em detalhe.

Baseado no trabalho anterior, a parte em que o cpu executa um processo, sofreu algumas modificações. O processador é agora responsável por executar um conjunto de instruções predefinidas, sendo que o estado running foi completamente redesenhado para suportar as mesmas. Um quadro com as instruções suportadas pode ser visualizado de seguida.

Quadro 1: Instruções suportadas

Códigos	Instruções	Significado
0X	ZERO X	X = 0
1X	ADD X	X ++
2X	SUB X	X --
3X	IF X	If X ==0, goto next line (PC++); else goto next line +1 (PC += 2)
4X	BACK X	Goto PC -= X
5X	FORW X	Goto PC += X
6X (facultativa)	FORK X	X = Fork() (facultativa)
7X	DISK SAVE X	Wait...
8X	COPY X	X0 = X
9X	EXIT	Exit

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.

DESENVOLVIMENTO

Implementação da gestão de memória

Blocos de memória

Para gerir a memória disponível e saber que zonas de memória estão ocupadas ou livres, a quantidade de memória disponível total é representada por um bloco de memória que contem a estrutura representada na figura 2.

MemBlock
startAddress
endAddress
memPoolSize
Allocated

Figura 2: Estrutura do MemBlock

Quando um bloco de memória é alocado, a variável Allocated muda para TRUE, significando que aquele bloco está ocupado. As variáveis startAddress e endAddress simbolizam o início e fim dos endereços de memória para esse bloco. A variável memPoolSize contem o tamanho de memória total ocupado por esse bloco.

Os blocos de memória estão inseridos numa lista ligada, em que cada bloco está ligado ao bloco seguinte e ao bloco anterior, o que significa que os blocos estão sequencialmente ligados e ordenados baseado no seu espaço de endereçamento. Um bloco que começa no endereço 100 e acaba no endereço 110 nunca pode estar ligado a um bloco seguinte que começa no endereço 120 e acaba no endereço 130, ou haveria um buraco entre eles do endereço 111 a 119 que nunca poderia ser preenchido.

A figura 3 contém uma representação de como os blocos estão distribuídos na lista ligada.

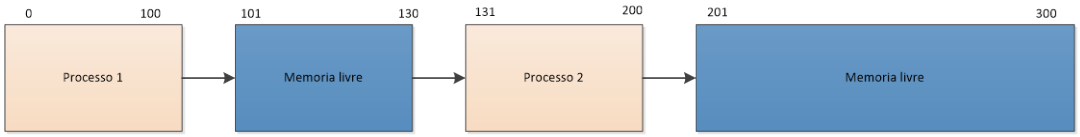

Figura 3: Lista com todos os blocos de memória ocupados/livres

Gestor de memória MemFactory

O gestor de memória MemFactory é responsável pela gestão dos pedidos de alocação e desalocação de memória.

Sempre que um processo é criado, é feito um pedido de alocação de memória através do método **memAlloc.** Este método permite alocar memória dependendo do algoritmo usado em que é retornado o endereço de memória do espaço alocado.

No geral, independentemente do algoritmo aplicado, quando um processo faz um pedido de alocação de memória, é verificado a lista de blocos de memória para ver se é possível atribuir memória suficiente ao processo. Isto é feito ao verificar os blocos que correspondem à memória livre.

Se um bloco de memória livre tem espaço suficiente para o processo, o bloco em questão é dividido ou não em dois, dependendo se o espaço do bloco é ou não igual a quantidade de memória necessária para o processo. Sendo que o bloco livre dividido ficará com um bloco com espaço ocupado e outro bloco com o restante espaço livre (figura 4).

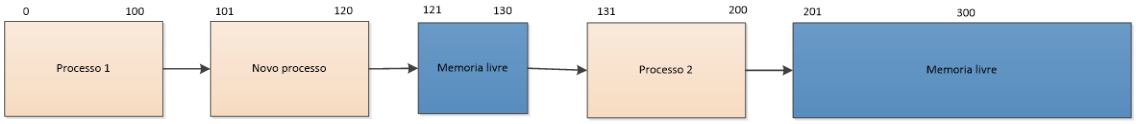

Figura 4: Lista com todos os blocos de memória ocupados e livres depois de uma alocação.

Sempre que um processo chega ao estado **Exit**, o método **memfree** é chamado para que a memória ocupada previamente pelo processo seja libertada. O bloco de memória associado a este processo passa ao estado de desalocado (memoria livre). Se houver algum bloco anterior ou a seguir a este bloco que esteja também livre, é feita uma fusão dos dois blocos, passando a estar apenas um bloco livre contendo o espaço de endereçamento dos dois blocos fundidos.

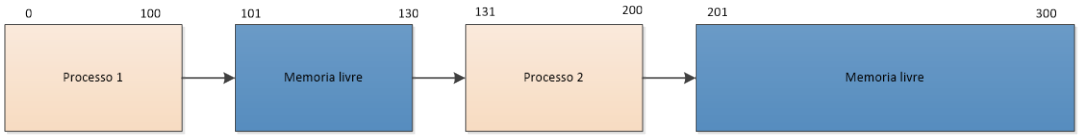

Figura 5: Lista com todos os blocos de memória ocupados e livres depois de um free.

Para finalizar é de notar que a simulação da memória é feita usando uma array estática de espaço finito e pré-definido que pode ser alterado para outros valores. Sendo assim o espaço de endereçamento corresponde às posições na array.

Implementação dos vários estados e do simulador

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.

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 dois ou menos 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

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

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

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

1. Verificar se existem processos 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. Chamar o método run do dispositivo para que este continue a processar o pedido.
4. 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

É neste estado que o processador irá executar a instrução que está no endereço de memória guardado no program counter. As seguintes verificações são efectuadas:

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.
4. Ler a instrução que está na memória através do program counter.
5. Descodificar a instrução, obtendo o opcode da operação e o valor associado a essa operação.
6. Executar a operação seleccionada e incrementar o program counter.
7. 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.
8. Se o processo é interrompido por um evento, é incrementado o program counter, 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.
9. Caso ainda exista instruções a serem processadas, é incrementado o program counter, 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. Ao contrário do trabalho anterior, a file dos dispositivos é única em vez de ter várias filas, uma por dispositivo (figura 6).

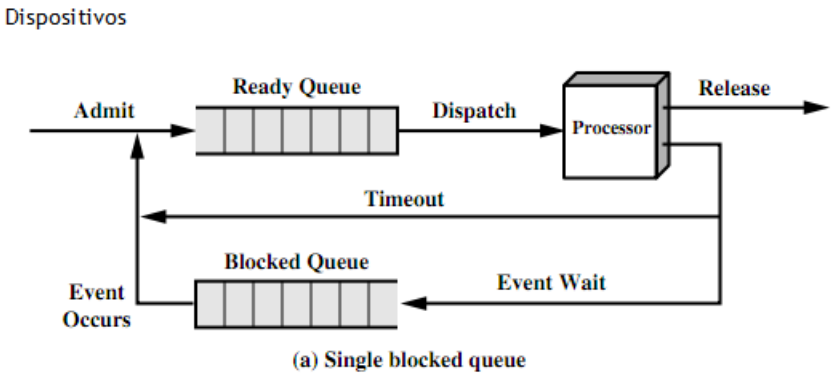

Figura 6: Lista única para o dispositivo do estado blocked.

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.

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 estrutura do PCB está indicada na figura 7.

PCB
Pid
Dataid
PC
creationTime
baseAddress
baseVarAddress
allocatedMemSize
CurrentPipelineCycle
TempoDeEspera
State

Figura 7: Estrutura do PCB

- **Pid:** Contem o id do processo.
- **DataId:** Contem o id da posição na array de dados que foram carregados a partir do ficheiro mas que ainda não estão em memória.
- **CreationTime:** Contem o ciclo em que o processo é criado e entre no estado NEW.
- **BaseAddress:** Contem o endereço de memória onde começa a primeira instrução.
- **BaseVarAddress:** Contem o endereço de memória onde começa a primeira variável.
- **AllocatedMemSize:** Contem o tamanho do espaço alocado em memória.
- **CurrentPipeLineCycle:** Contem o valor do último ciclo em que este processo teve alguma actividade.
- **State:** Contem o estado actual do processo

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() Obtém o id do processo.
abstract int	getState() Obtém o estado do processo.
abstract void	setState(int value) Atribui o estado do processo.

Class Cpu_scheduler2

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 concluído.
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	getNumberOfProcess() Retorna o numero de processos.
int	getCreationTimeOfProcess(int n) Retorna o tempo do processo quando chega ao stage NEW.

Class MemBlock

Contem a estrutura de um bloco de memoria.

Modifier and Type	Method and Description
MemBlock	split(int size) Parte o bloco em dois.

Class MemFactory

Gestor de memória.
Contem metodos para obter, criar e libertar memoria de um process.
Class singleton, apenas uma instancia da classe esta disponivel, usar getInstance()

Modifier and Type	Method and Description
MemBlock	split(int size) Parte o bloco em dois.
MemFactory	getInstance() Retorna a instancia da classe.
int	memAlloc(int size) Permite alocar memoria.
MemBlock	getBestFit(int size) Retorna um menor bloco livre que tenha tamanho maior ou igual ao size.
MemBlock	getWorstFit(int size)

	Retorna o maior bloco livre que tenha tamanho maior ou igual ao size.
void	Memfree(int size, PCB pcb) Permite libertar memoria que foi alocada por um processo. Funde blocos de memoria caso estes estejam ligados e libertados, criando um bloco de memoria livre maior.
void	memCopy(int address, int data[]) Permite copiar o conteúdo de uma array para um endereço especifico na memoria.
void	copyToAddress(int originAdress, int destinyAddress, int amountToCopy) Copia o conteúdo a partir de uma posição de memória para outra posição de memória.
MemBlock	getMemBlockByStartAddress(int address) Obtém o bloco de memoria que coincide com o endereço inicial.
MemBlock	getMemBlockByEndAddress(int address) Obtém o bloco de memoria que coincide com o endereço final.
LinkedList<MemBlock>	getMemBlock() Retorna a lista de blocos de memoria livres e alocados.
void	setMemBlock(LinkedList<MemBlock> memBlock) Atribui uma nova lista de blocos de memoria.
int	getMemAlgo() Retorna o algoritmo de gestão de memória a ser aplicado na alocação de memória.
void	setMemAlgo(int memAlgo) Define qual o algoritmo de gestão de memória que ira ser usado para a alocação de memória.

Class PCB

Contem os dados de um processo.

Modifier and Type	Method and Description
PCB	Fork() O novo processo criado pela função fork e chamado de processo filho. Todas as variáveis do processo são duplicadas dentro do sistema operativo.
int	getCreationTime() Obtém o tempo em que o processo e criado.
int	getAllocatedMemSize() Retorna o tamanho de memoria alocada.
int	getCurrentPipelineCycle() Obtém o último ciclo em que ouve actividade no processo.
void	setAllocatedMemSize(int allocatedMemSize) Atribui o espaço ocupado em memoria para este processo.
int	getDataid() Obtém o id da posição da array que contem os dados com as instruções deste processo para processamento.
int	getPID() Obtem o id do processo.
int	getState() Obtem o estado do processo.
int	getTempoDeEspera() Obtem o tempo de espera no estado ready.
void	setDataid(int dataid) Atribui o id da posição da array que contem os dados com as instruções deste processo para processamento.
int	getBaseVarAddress() Contem o endereço de memória onde começa a primeira variável.
void	setBaseVarAddress(int baseVarAddress) Atribui o endereço de memória onde começa a primeira variável.
int	getPID() Obtem o id do processo.
int	getPC() Retorna o valor do program counter.
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	void setPC(int PC) Atribui um endereco de memoria ao Program Counter.
void	setState(int value) Atribui o estado do processo.
void	setTempoDeEspera(int tempoDeEspera) Atribui o tempo de espera no estado ready.
int	getBaseAddress() Obtém o endereço de memoria base do programa.
void	setBaseAddress(int baseAddress) Atribui o endereço de memoria base do programa.
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.
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

Para esta segunda parte do trabalho, foi fácil implementar a parte da execução das instruções usando apenas um modulus e divisão, evitando problemas com as instruções que começavam por zero.

A parte mais desafiante foi pensar numa estrutura para saber que espaços na memória estavam livres ou ocupados, chegando-se a conclusão que o uso de listas ligadas seria a melhor opção para representar os blocos da memória.

Com este trabalho ficamos também com uma mínima ideia do que é necessário para emular um processador simples.

BIBLIOGRAFIA

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

ANEXOS

Anexo 1: Código fonte

Class AbstractPCB

```
/**
 * 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_Scheduler2
public class Cpu_scheduler2
{

   public static boolean running = true;
   public static Pipeline pipeline;

   public static int PRINT_MODE_NORMAL = 0;
   public static int PRINT_MODE_DEBUG = 1;
   public static int PRINT_MODE_SPECIAL_DEBUG = 2;

   public static int printMode = PRINT_MODE_NORMAL;
   /**
    * @param args the command line arguments
    */
   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_scheduler2.class.getName()).log(Level.SEVERE,
               null, ex);
      }
   }
}
```

Class Device

```
/**
 * Contem metodos que permitem manipular um dispositivo.
 *
 */
public class Device
{
   public 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
/**
 * Le o conteudo de um ficheiro que contem os dados
 * a serem processados pelo simulador.
 */
public class FileInput
{
   FileReader file;
   BufferedReader br;
   public static ArrayList<String> dados = new ArrayList<String>();
   int numProcesses = 0;

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

   /**
    * Retorna o tempo do processo quando chega ao stage NEW.
    * @param n
    * @return
    */
   public int getCreationTimeOfProcess(int n)
   {
      String aux = dados.get(n);
```

```
    String[] dados = aux.split(" ");
    return Integer.valueOf(dados[0]);

}

/**
 * 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);
        numProcesses++;
    }
}

/**
 * Retorna o numero de processos.
 * @return
 */
public int getNumberOfProcess()
{
   return numProcesses;
}
```

```
/**
 * Retorna os dados associados a um processo n.
 * @param n
 * @return
 */
public static int[] getDadosOfProcess(int n)
{
   String aux = dados.get(n);
   String[] dados = aux.split(" ");
   int[] result = new int[dados.length - 1];
   for(int i = 1; i < dados.length; i++){
         result[i - 1] = 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;

   for(int i = 0; i < numP; i++)
   {
```

```
            int dados[] = this.getDadosOfProcess(i);
            PCB pcb = new PCB();
            pcb.setDataid(i);
            pcb.setCreationTime( getCreationTimeOfProcess(i));

            Cpu_scheduler2.pipeline.queueProcessCreation(pcb);
        }
    }
}
Class MemBlock
/**
 * Contem a estructura de um bloco de memoria.
 *
 */
public class MemBlock
{
    public int startAddress;
    public int endAddress;
    public int memPoolSize;
    public boolean allocated;

    public MemBlock()
    {
        startAddress = 0;
        endAddress = 0;
        memPoolSize = 0;
        allocated = false;
    }
```

```
/**
 * Parte o bloco em dois.
 * @param size
 * @return novo bloco
 * @throws ProcessException
 */
public MemBlock split(int size) throws ProcessException{

        boolean sameSize = memPoolSize == size;

        if(!allocated && !sameSize)
   {
      MemBlock newBlock = new MemBlock();
      newBlock.memPoolSize = size;
      newBlock.startAddress = this.startAddress;
      newBlock.endAddress = newBlock.startAddress + size - 1;
      this.startAddress = newBlock.endAddress + 1;
      this.memPoolSize = memPoolSize - size;

      return newBlock;
        }
   else
      throw new ProcessException(
            ProcessException.PROCESS_EXCEPTION_RUNTIME_ERROR_OUT_OF_MEMORY);
}

@Override
```

```
    public String toString()
    {
        String output = "";
        output += "[Alocado: " + allocated + " Enderecamento: " + startAddress
        + " to " + endAddress + " Tamanho: " + memPoolSize + "]";
        return output;
    }
}
```

Class MemFactory

```
/**
 * Gestor de memoria.
 * Contem metodos para obter, criar e libertar memoria de um process.
 * Class singleton, apenas uma instancia da classe esta disponivel, usar
 * getInstance()
 */
public final class MemFactory
{
    //constants
    public static final int MEM_ALGO_BEST_FIT  = 0; //procura o espaco mais a medida( o
bloco mais pequeno de todos em que ele cabe).
    public static final int MEM_ALGO_WORST_FIT = 1; //procura o espaco maior disponivel.

    //singleton instance access
    private static MemFactory instance = new MemFactory();

    //public static variables.
    public static int MEM_POOL_SIZE = 300;
    public static int MEM_VARIABLE_SIZE = 10;
    public static int MEM[] = new int[MEM_POOL_SIZE];
```

```
private LinkedList<MemBlock> memBlock;
private int memAlgo;

/**
 * Constructor privado que impede a criacao de multiplas copias desta
 * classe.
 */
private MemFactory()
{
   memAlgo = MEM_ALGO_WORST_FIT;
   memBlock = new LinkedList<MemBlock>();
}

/**
 * Retorna a instancia da classe.
 * @return
 */
public static MemFactory getInstance()
{
   return instance;
}

/**
 * Permite alocar memoria.
 * Lanca uma excepcao se nao houver espaco disponivel.
 * @param size
 * @return
```

```
 * @throws Exception
 */
public int memAlloc(int size) throws ProcessException{
        MemBlock newBlock = null;
        MemBlock block = null;

        switch (memAlgo)
  {
     case MEM_ALGO_BEST_FIT:
        block = getBestFit(size);
        break;
     case MEM_ALGO_WORST_FIT:
        block = getWorstFit(size);
        break;
     default:
        break;
  }

       if(block == null)
     throw new ProcessException(
       ProcessException.PROCESS_EXCEPTION_RUNTIME_ERROR_OUT_OF_MEMORY);

       // Parte o bloco em dois e retorna o novo bloco
       if(size < block.memPoolSize)
                newBlock = block.split(size);

       newBlock.allocated = true;
       memBlock.add(memBlock.indexOf(block), newBlock);
```

```
        return newBlock.startAddress;
}

/**
 * Retorna um menor bloco livre que tenha
 * tamanho maior ou igual ao size
 *
 * @param size
 * @return
 * @throws ProcessException
 */
private MemBlock getBestFit(int size)
{
        int diff = 0;
        int lastBestFit = MEM.length;

        MemBlock newBlock = null;

        for( MemBlock block : memBlock)
   {
      if( fits(block, size) )
      {
                diff = block.memPoolSize - size;

         if(diff < lastBestFit )
         {
            lastBestFit = diff;
            newBlock = block;
```

```
        }
      }
        }
        return newBlock;
}

/**
 * Retorna o maior bloco livre que tenha
 * tamanho maior ou igual ao size
 * @param size
 * @return
 * @throws ProcessException
 */
private MemBlock getWorstFit(int size)
{
        int diff = 0;
        int lastWorstFit = 0;

        MemBlock newBlock = null;

        for( MemBlock block : memBlock)
   {
      if( fits(block, size) )
      {
         diff = block.memPoolSize - size;

                  if( diff > lastWorstFit )
         {
```

```
            lastWorstFit = diff;
            newBlock = block;
         }
      }
       }
       return newBlock;
}

/**
 * Permite saber se um determinado tamanho de memoria cabe num bloco de
 * memoria livre.
 * @param block
 * @param size
 * @return
 */
private boolean fits(MemBlock block,int size)
{
       return block.memPoolSize >= size && !block.allocated;
}

/**
 * Permite libertar memoria que foi alocada por um processo.
 * Funde blocos de memoria caso estes estejam ligados e libertados, criando
 * um bloco de memoria livre maior.
 * @param size
 * @param pcb
 */
public void memfree(int size, PCB pcb)
```

```
  {
     MemBlock block = getMemBlockByStartAddress(pcb.getBaseVarAddress());
     //assert(block != null);

     for(int i = 0; i < size; i++)
     {
        MEM[ pcb.getBaseVarAddress() + i] = 0;
     }
     //libertar a memoria
     block.allocated = false;
     //fundir blocos livres
     //obter o bloco anterior
     MemBlock startblock = getMemBlockByEndAddress(pcb.getBaseVarAddress() - 1);
     //obter o seguinte.
     MemBlock endblock = getMemBlockByStartAddress(pcb.getBaseVarAddress() +
pcb.getAllocatedMemSize());

     if(startblock  != null && !startblock.allocated)
     {
        block.startAddress = startblock.startAddress;
        block.memPoolSize = block.memPoolSize + startblock.memPoolSize;
        memBlock.remove(startblock);
     }
     if(endblock != null && !endblock.allocated)
     {
        block.endAddress = endblock.endAddress;
        block.memPoolSize = block.memPoolSize + endblock.memPoolSize;
        memBlock.remove(endblock);
     }
```

```
}

/**
 * Permite copiar o conteudo de uma array para um endereco especifico na
 * memoria.
 * @param address
 * @param data
 */
public void memCopy(int address, int data[])
{
   System.arraycopy(data, 0, MEM, address, data.length);
}

/**
 * Copia o conteudo a partir de uma posicao de memoria para outra posicao
 * de memoria.
 * @param originAdress
 * @param destinyAddress
 * @param amountToCopy
 */
public void copyToAddress(int originAdress, int destinyAddress, int amountToCopy)
{
        for(int i = 0, j = destinyAddress; i < amountToCopy; i++, j++){
                MEM[j] = MEM[originAdress + i];
        }
}

/**
```

```
 * Obtem o bloco de memoria que coincide com o endereco inicial.
 * @param address
 * @return
 */
private MemBlock getMemBlockByStartAddress(int address)
{
   for( MemBlock block : memBlock)
   {
      if(block.startAddress == address)
      {
         return block;
      }
   }
   return null;
}

/**
 * Obtem o bloco de memoria que coincide com o endereco final.
 * @param address
 * @return
 */
private MemBlock getMemBlockByEndAddress(int address)
{
   for( MemBlock block : memBlock)
   {
      if(block.endAddress == address)
      {
         return block;
```

```
      }
    }
    return null;
}

/**
 * Retorna a lista de blocos de memoria livres e alocados.
 * @return
 */
public LinkedList<MemBlock> getMemBlock()
{
    return memBlock;
}

/**
 * Atribui uma nova lista de blocos de memoria.
 * @param memBlock
 */
public void setMemBlock(LinkedList<MemBlock> memBlock)
{
    this.memBlock = memBlock;
}

/**
 * Retorna o algoritmo de gestao de memoria a ser aplicado na alocacao de
 * memoria.
 * @return
 */
```

```
public int getMemAlgo()
{
   return memAlgo;
}

/**
 * Define qual o algoritmo de gestao de memoria que ira ser usado para a
 * alocacao de memoria.
 * @param memAlgo
 */
public void setMemAlgo(int memAlgo)
{
   this.memAlgo = memAlgo;
}


@Override
public String toString()
{
   String output = "";
   //array de memoria

   if(Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_SPECIAL_DEBUG)
   {
        output += "MEMORIA [";
      for(int i = 0; i < MEM.length; i++)
      {
         output += MEM[i] + " ";
```

```
            }
            output += "]\n";

            //array dos blocos
            output += "BLOCOS: {";
            for( MemBlock block : memBlock)
            {
               output += block.toString();
            }
            output += "}\n";
        }
        else if(Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_NORMAL ||
                Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_DEBUG)
        {
            output += "MEMORIA [";
            for(int i = 0; i < MEM.length; i++)
            {
               output += MEM[i] + " ";
            }
            output += "]\n";
        }

        return output;
    }
}
```

Class PCB

```
/**
```

```
 * Contem os dados de um processo.
 *
 */
public class PCB extends AbstractPCB
{
   private int pid;
   private int dataid;
   private int PC;
   private int creationTime;
   private int baseAddress;
   private int baseVarAddress;
   private int allocatedMemSize;
   private int currentPipelineCycle;
   private int tempoDeEspera;
   private int state = PipelineI.PROCESS_STATE_NOT_CREATED;

   public PCB()
   {
      this.pid = PCB.pidNumber++;
      this.currentPipelineCycle = 0;
      this.tempoDeEspera = 0;
      this.creationTime = 0;
      this.PC = 0;
      this.baseAddress = 0;
      this.baseVarAddress = 0;
      this.allocatedMemSize = 0;
      this.dataid = 0;
   }
```

```
public PCB(int state)
{
   this.pid = PCB.pidNumber++;
   this.state = state;
   this.currentPipelineCycle = 0;
   this.creationTime = 0;
   this.PC = 0;
   this.baseAddress = 0;
   this.tempoDeEspera = 0;
}

/**
 * O novo processo criado pela funcao fork e chamado de processo filho.
 * Todas as variaveis do processo sao duplicadas dentro do sistema
 * operativo.
 * @return
 */
public PCB fork()
{
   // Aloca memoria para o que resta a ser processado
   int newPosition = 0;
   int diff = this.PC - this.baseAddress;
   int newSize = this.allocatedMemSize - diff;
   try
   {
      newPosition = MemFactory.getInstance().memAlloc(newSize);
   }
```

```
    catch (ProcessException e)
    {
       e.printStackTrace();
    }
    // Cria o novo processo
    PCB newProcess = new PCB(PipelineI.PROCESS_STATE_NEW);
    newProcess.setBaseVarAddress(newPosition);
    newProcess.setBaseAddress(newPosition + MemFactory.MEM_VARIABLE_SIZE);
    newProcess.setPC(newProcess.getBaseAddress());
    newProcess.setAllocatedMemSize(newSize);

    // Copia as variaveis
    MemFactory.getInstance().copyToAddress(
                        this.baseVarAddress,
                        newProcess.getBaseVarAddress(),
                        MemFactory.MEM_VARIABLE_SIZE
                        );
    // Copia os dados
    MemFactory.getInstance().copyToAddress(this.PC, newProcess.getBaseAddress(),
newSize - MemFactory.MEM_VARIABLE_SIZE);

    return newProcess;
  }

  /**
   * Retorna o tamanho de memoria alocada.
   * @return
   */
  public int getAllocatedMemSize()
```

```
{
    return allocatedMemSize;
}

/**
 * Atribui o espaco ocupado em memoria para este processo.
 * @param allocatedMemSize
 */
public void setAllocatedMemSize(int allocatedMemSize)
{
    this.allocatedMemSize = allocatedMemSize;
}

/**
 * Obtem o id da posicao da array que contem os dados com as instrucoes
 * deste processo para processamento.
 * @return
 */
public int getDataid()
{
    return dataid;
}

/**
 * Atribui o id da posicao da array que contem os dados com as
 * instrucoes deste processo para processamento.
 * @param dataid
 */
```

```
public void setDataid(int dataid)
{
    this.dataid = dataid;
}

/**
 * Contem o endereço de memória onde começa a primeira variável.
 * @return
 */
public int getBaseVarAddress()
{
    return baseVarAddress;
}

/**
 * Atribui o endereço de memória onde começa a primeira variável.
 * @param baseVarAddress
 */
public void setBaseVarAddress(int baseVarAddress)
{
    this.baseVarAddress = baseVarAddress;
}

/**
 * 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;
    }

    /**
     * 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 o tempo em que o processo e criado.
 * @return
 */
public int getCreationTime()
{
    return creationTime;
}

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

/**
 * Retorna o valor do Program Counter.
 * @return
 */
public int getPC()
```

```
{
    return PC;
}

/**
 * Atribui um endereco de memoria ao Program Counter.
 * @param PC
 */
public void setPC(int PC)
{
    assert(PC < baseAddress || PC >= this.allocatedMemSize);
    this.PC = PC;
}

/**
 * Obtem o endereco de memoria base do programa.
 * @return
 */
public int getBaseAddress()
{
    return baseAddress;
}

/**
 * Atribui o endereco de memoria base do programa.
 * @param baseAddress
 */
public void setBaseAddress(int baseAddress)
```

```
    {
        this.baseAddress = baseAddress;
    }

    @Override
    public String toString()
    {
        return new String();
    }
}
```

Class Pipeline

```
/**
 * 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;
    private int numCycles = 0;
    private LinkedList<PCB> preProcessCreationQueue;
    public static PipelineStage[] pipelineStages;

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

```
    //gestao de espaco livre em memoria
    MemBlock memblock = new MemBlock();
    memblock.endAddress = 299;
    memblock.memPoolSize = 300;
    MemFactory.getInstance().getMemBlock().add(memblock);



    //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
     */
    @Override
    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);

                //alocar memoria.
                int data[] = FileInput.getDadosOfProcess( pcb.getDataid());
                int address = MemFactory.getInstance().memAlloc(data.length + 10);

                MemFactory.getInstance().memCopy(address +
MemFactory.MEM_VARIABLE_SIZE, data);
```

```
            pcb.setBaseVarAddress(address);
            pcb.setBaseAddress( address + MemFactory.MEM_VARIABLE_SIZE);
            pcb.setPC(address + MemFactory.MEM_VARIABLE_SIZE);
            pcb.setAllocatedMemSize(data.length + MemFactory.MEM_VARIABLE_SIZE);

            pipelineStages[Pipeline.PIPELINE_STAGE_NEW].addProcessToQueue(pcb);

            if(Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_NORMAL)
                    System.out.print("Inicio do processo "+pcb.getPID()+",
"+MemFactory.getInstance()+"\n");

            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_RUN].update(numCycles);
        pipelineStages[PipelineI.PIPELINE_STAGE_READY].update(numCycles);
```

```
    pipelineStages[PipelineI.PIPELINE_STAGE_EXIT].update(numCycles);

    if(Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_SPECIAL_DEBUG)
    {
        System.out.print(this);
        System.out.print(MemFactory.getInstance());
    }
    if(Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_DEBUG)
    {
        if(numCycles < 10)
        {
            System.out.println(MemFactory.getInstance());
        }
    }
    numCycles++;
}

@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 += "-------------------------------\n";
        return output;
    }
}
```

Interface PipelineI

```
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

```
/**
 * 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.add(process);
    }

    /**
     * Obtem o tamanho da lista de processos.
     * @return
     */
```

```
    public int getQueueSize()
    {
       return queue.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(PCB pcb : queue)
      {
        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 = "";


        return output;
    }
}
```

Class ProcessException

```
class ProcessException extends Exception
{
   public static final String PROCESS_EXCEPTION_RUNTIME_ERROR = "Ocurreu um erro ao
correr o processo, tempo invalido";
   public static final String PROCESS_EXCEPTION_RUNTIME_ERROR_UNKNOWN_OPCODE
= "Ocurreu um erro ao correr o processo, cpu opcode desconhecido";
   public static final String PROCESS_EXCEPTION_RUNTIME_ERROR_OUT_OF_MEMORY =
"Erro ao alocar memoria, memoria insuficiente";


   public ProcessException()
   {
      super();
   }


   public ProcessException(String s)
```

```
    {
        super(s);
    }
}
```

Class StageBlocked

```
/**
 * Gere os processos do stage blocked.
 *
 */
public class StageBlocked extends PipelineStage
{
    public static Device dev;
    public StageBlocked(int stageType)
    {
        super(stageType);
        queue = new LinkedList<PCB>();
        dev  = new Device(3);
    }

    /**
    * 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.
        //existe processos neste device
        if(queue.size() > 0)
        {
           PCB pcb = queue.getFirst();
           //processo esta no ciclo correcto
           if(pcb.getCurrentPipelineCycle() < cycles)
           {

              dev.run();
              if(dev.isFinished())
              {
                 pcb.setState(PipelineI.PROCESS_STATE_READY);
                              pcb.setCurrentPipelineCycle(cycles);

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

   }
}
```

Class StageExit

```
/**
 * Gere os processos do stage exit.
 *
```

```
 */
public class StageExit extends PipelineStage
{
   public StageExit(int stageType)
   {
      super(stageType);
      queue = 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)
      {
         //processo esta a espera a pelo menos 1 ciclo
         if(pcb.getCurrentPipelineCycle() < cycles)
         {
            pcb.setCurrentPipelineCycle(cycles);
            numProcessesOnCorrectCycle++;
         }
```

```
        }

        if(numProcessesOnCorrectCycle == Pipeline.numProcesses)
        {
            Cpu_scheduler2.running = false;
            queue.clear();
        }
    }
}
Class StageNew
/**
 * Gere os processos do stage new.
 *
 */
public class StageNew extends PipelineStage
{
    public StageNew(int stageType)
    {
        super(stageType);
        queue = 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.size() > 0 &&
Cpu_scheduler2.pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_READY].getQueueSize
() <= Pipeline.READY_STAGE_PROCESS_LIMIT)
      {
         //apenas avancar o processo se for o ciclo seguinte
         if(queue.getFirst().getCurrentPipelineCycle() < cycles)
         {
            PCB Process = queue.removeFirst();
            Process.setCurrentPipelineCycle(cycles);
            Process.setState(PipelineI.PROCESS_STATE_READY);

Cpu_scheduler2.pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_READY].addProcessT
oQueue( Process );
            Process = null;
         }
      }
   }
}
```

Class StageReady

```
/**
 * Gere os processos do stage ready.
 *
 */
public class StageReady extends PipelineStage
{
```

```
public StageReady(int stageType)
{
   super(stageType);
   queue = 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)
   {
      if(pcb.getCurrentPipelineCycle() < cycles)
      {
         pcb.setTempoDeEspera( pcb.getTempoDeEspera() + 1);
      }
   }

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

```
            process.setState(PipelineI.PROCESS_STATE_RUN);
            process.setCurrentPipelineCycle(cycles);

Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_RUN].addProcessToQueue(process);
        }
    }
}
Class StageRun
/**
 * Gere os processos do stage run.
 *
 */
public class StageRun extends PipelineStage
{
    public static int CPU_CYCLES_PER_PROCESS = 4;

    public static final int OPCODE_ZERO      = 0x0;
    public static final int OPCODE_ADD       = 0x1;
    public static final int OPCODE_SUB       = 0x2;
    public static final int OPCODE_IF        = 0x3;
    public static final int OPCODE_BACK      = 0x4;
    public static final int OPCODE_FORW      = 0x5;
    public static final int OPCODE_FORK      = 0x6;
    public static final int OPCODE_DISK_SAVE = 0x7;
    public static final int OPCODE_COPY      = 0x8;
    public static final int OPCODE_EXIT      = 0x9;

    private int currentProcessCyles = 0;
```

```
public StageRun(int stageType)
{
   super(stageType);
   queue = 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 ProcessException
{
   //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.size() > 0 &&
     queue.getFirst().getCurrentPipelineCycle() < cycles)
```

```
{
   PCB process = queue.getFirst();

   //incrementar o tempo de cpu para este processo.
   currentProcessCyles = inc(currentProcessCyles);
   process.setCurrentPipelineCycle(cycles);

   Integer opcode = MemFactory.MEM[ process.getPC() ];
   int high = (opcode % 100 / 10);
   int low = (opcode % 10);
   switch(high)
   {
      case OPCODE_ADD:
      {
         MemFactory.MEM[ process.getBaseVarAddress() + low]++;
         process.setPC( process.getPC() + 1);
      }break;
      case OPCODE_ZERO:
      {
         MemFactory.MEM[ process.getBaseVarAddress() + low] = 0;
         process.setPC( process.getPC() + 1);
      }break;
      case OPCODE_SUB:
      {
         MemFactory.MEM[ process.getBaseVarAddress() + low]--;
         process.setPC( process.getPC() + 1);
      }break;
      case OPCODE_IF:
```

```
{
   if(MemFactory.MEM[ process.getBaseVarAddress() + low] == 0)
   {
      process.setPC( process.getPC() + 1);
   }
   else
   {
      process.setPC( process.getPC() + 2);
   }
}break;
case OPCODE_BACK:
{
   process.setPC( process.getPC() - low);
}break;
case OPCODE_FORW:
{
   process.setPC( process.getPC() + low);
}break;
case OPCODE_FORK:
{
   // Incrementar primeiro o PC para evitar
   // que o fork seja feito infinitamente pelo processo filho
   process.setPC( process.getPC() + 1);
   PCB newProcess = process.fork();
   newProcess.setCurrentPipelineCycle(cycles);
   Pipeline.numProcesses++;
```

```
Pipeline.pipelineStages[Pipeline.PIPELINE_STAGE_NEW].addProcessToQueue(newProces
s);


            if(Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_NORMAL)

                  System.out.print("Inicio do processo "+newProcess.getPID()+",
"+MemFactory.getInstance()+"\n");

         }break;

         case OPCODE_DISK_SAVE:

         {

            process.setState(PipelineI.PROCESS_STATE_BLOCKED);


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

            resetCPUCycles();

            process.setPC( process.getPC() + 1);

         }break;

         case OPCODE_COPY:

         {

            MemFactory.MEM[ process.getBaseVarAddress()] = low;

            process.setPC( process.getPC() + 1);

         }break;

         case OPCODE_EXIT:

         {

            process.setState(PipelineI.PROCESS_STATE_EXIT);


Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_EXIT].addProcessToQueue(queue.rem
oveFirst());

            resetCPUCycles();


            if(Cpu_scheduler2.printMode == Cpu_scheduler2.PRINT_MODE_NORMAL)
```

```
                    System.out.println("Fim do processo "+process.getPID()+",
"+MemFactory.getInstance());


                //libertar a memoria aqui?
                MemFactory.getInstance().memfree( process.getAllocatedMemSize(), process);
            }break;


            default:
                throw new ProcessException(
ProcessException.PROCESS_EXCEPTION_RUNTIME_ERROR_UNKNOWN_OPCODE);
        }


        //fim de cpu burst, enviar para o estado ready.
        if(currentProcessCyles == CPU_CYCLES_PER_PROCESS)
        {
            resetCPUCycles();
            process.setState(PipelineI.PROCESS_STATE_READY);

Pipeline.pipelineStages[PipelineI.PIPELINE_STAGE_READY].addProcessToQueue(queue.re
moveFirst());
            return;
        }


    }
  }


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

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