Implementando um Simulador de Mecanismos de Alocação da CPU

Ângelo N. Vimeney
COPPE/UFRJ
avimeney@cos.ufrj.br

Alexandre Sztajnberg
DICC/IME/UERJ
alexszt@ime.uerj.br

Resumo

O escalonamento de processos é um assunto que requer dos alunos dos cursos de graduação em Ciências da Computação uma alta capacidade de abstração. Seria desejável uma ferramenta que auxiliasse na compreensão dos aspectos cuja compreensão é mais difícil quando apresentados apenas pelos métodos de ensino convencionais. Desenvolvemos um simulador de mecanismos de alocação da CPU que se propõe a facilitar a tarefa de avaliar o impacto da implantação de um determinado mecanismo de alocação da CPU sobre um sistema computacional. O simulador fornece aos usuários meios de criar mecanismos de alocação da CPU de forma personalizada e, em seguida, testá-los sobre diferentes condições de funcionamento. Conseguimos um software didático, extensível, reutilizável, modular e independente de plataforma.

Abstract

Process scheduling is a subject that usually requires from the undergraduate students on Computer Science high abstraction capability. It would be desirable to have adequate tools to help the comprehension of aspects that are more difficult to understand when presented by conventional learning methods. We developed a tool to simulate CPU allocation mechanisms of computing systems. The simulator provides to the user the means to create customized CPU allocation mechanisms and, then test them on different operating conditions. The developed software is extensible, reusable, modular and platform-independent.

1. Introdução

O escalonamento de processos é um assunto que requer dos alunos dos cursos de graduação em Ciências da Computação uma alta capacidade de abstração, ficando ainda mais difícil o seu completo entendimento para aqueles que não possuem uma maturidade dentro da área de programação e sistemas operacionais. Desta forma, seria desejável uma ferramenta que auxiliasse o professor na tarefa de tornar os mecanismos de alocação da CPU algo mais concreto, uma ferramenta que ajudasse a compreender os aspectos que são obscuros quando tratados somente pelos métodos de ensino convencionais.

Como uma tentativa de criar esta ferramenta didática, desenvolvemos um simulador de mecanismos de alocação da CPU que permite avaliar o impacto da implantação de um determinado mecanismo de alocação da CPU sobre um sistema computacional. O desenvolvimento do simulador foi fortemente baseado nos conceitos apresentados na literatura clássica da área de sistemas operacionais.

Recriamos em *software* os componentes de um sistema computacional real de forma que se tornasse o mais simples possível fazer com que o simulador se comportasse como tal. Certos componentes do sistema real puderam ser omitidos, uma vez que suas funções não são relevantes no escopo do escalonamento de processos. Outros componentes como a unidade de memória principal, embora imprescindíveis para o funcionamento de qualquer sistema computacional moderno, foram também omitidos, pois a conseqüência direta da sua representação seria a criação de um simulador completo de arquitetura, o que foge do objetivo do projeto. Em linhas gerais, os componentes que foram mais profundamente explorados são a CPU, os dispositivos de I/O e os diversos elementos inerentes ao sistema operacional como, por exemplo, as filas de processos, as rotinas de tratamento de interrupção e as rotinas para o escalonamento de processos.

Os componentes que potencialmente trabalham em paralelo em um sistema computacional real como a CPU e os dispositivos de I/O receberam, na sua representação, fluxos (*threads*) de execução próprios. Os fluxos independentes permitem que esses componentes operem de forma concorrente e passem ao usuário do simulador a impressão de paralelismo. Foi criado um pequeno conjunto de instruções suportado pela CPU virtual. Cada processo possui uma seção de código preenchida por essas instruções. A CPU virtual interpreta cada uma dessas instruções, e pode ser interrompida por dispositivos de I/O, como em um sistema real. Os processos podem gerar exceções, ou fazer chamadas ao sistema, solicitando operações de entrada e saída.

Quando ocorre uma interrupção, ou um processo faz uma chamada ao sistema, a CPU virtual tem seu ciclo normal de interpretação de instruções interrompido e chama o sistema operacional virtual. A grande diferença entre o simulador e uma arquitetura real, ou até mesmo um simulador de arquitetura, é que, neste momento, a arquitetura real ou o simulador de arquitetura desviam seus fluxos de execução para a área de código do sistema operacional, enquanto, no simulador, o código que representa a CPU faz uma chamada a um método que, por sua

vez, contém código para representar o trabalho do sistema operacional. O núcleo do sistema operacional foi todo construído dentro deste único método que é chamado pela CPU nas condições descritas anteriormente.

Procuramos conferir versatilidade ao simulador, através da utilização dos conceitos de encapsulamento, herança e polimorfismo inerentes à linguagem orientada a objetos com a qual foi implementado o simulador. Através desses conceitos, o micronúcleo do sistema operacional virtual, assim como todas as outras classes responsáveis pela parte funcional foram mantidas simples, abstraindo as particularidades de cada política e algoritmos de alocação da CPU. Esses detalhes foram todos tratados por classes responsáveis por representar cada uma das opções de configuração possível.

No caso das políticas de alocação da CPU são disponibilizadas cinco políticas de alocação de CPU diferentes para que o usuário escolha entre uma delas para governar uma determinada fila de processos prontos. Foi escrita uma classe geral, descrevendo o comportamento de uma política genérica de alocação da CPU. Em seguida, foram escritas cinco classes que herdam desta classe geral, cada uma delas especializando-a, de forma a caracterizar o comportamento de cada uma das cinco políticas de alocação em particular. Quando o sistema operacional necessitar utilizar-se de alguma rotina pertencente ao módulo de escalonamento de processos, ele faz chamadas aos métodos da classe geral, tendo a certeza de que a herança e o polimorfismo vão resolver as questões relativas às particularidades de cada política.

O mesmo padrão foi adotado em vários outros momentos. Por exemplo: o conjunto de instruções suportado pela CPU é todo herdado de uma única classe, assim como os tratadores de interrupção e os serviços fornecidos pelo sistema operacional aos processos. É dessa forma que fica garantida a facilidade de extensão das funcionalidades do simulador de modo que possam ser inseridas com o menor esforço novas políticas de alocação da CPU, novas instruções ou novos serviços do sistema operacional. A manutenção também é facilitada, já que podemos trabalhar em separado com cada módulo responsável por uma determinada configuração.

Para que a funcionalidade do simulador possa ser explorada pelo usuário, foi fornecida uma interface gráfica onde as diversas opções de configuração podem ser acessadas de maneira estruturada, permitindo a criação de diversos ambientes de simulação com facilidade (Figura 1). Foram fornecidas opções *default* para os conjuntos de configuração mais avançados, de modo que o usuário pouco familiarizado com as peculiaridades do escalonamento de processos também possa fazer testes usando, para isso, os parâmetros mais comuns, típicos de sistemas reais (Figura 2(a) e (b)). Os usuários mais experientes podem acessar parâmetros avançados (Figura 2(c)) como, por exemplo, de quanto em quanto tempo o sistema operacional vai verificar a presença de novos processos na fila de entrada do sistema ou o comprimento do ciclo de *clock* da CPU, conseguindo assim resultados mais elaborados na sua simulação.

A versão do simulador descrita neste trabalho foi desenvolvida na linguagem Java, versão 1.2. Utilizamos o pacote *Swing*, versão 1.0.2, através do qual foi desenvolvida a interface com o usuário. Como conseqüência, o simulador requer a submissão de suas classes componentes à execução em uma máquina virtual Java.

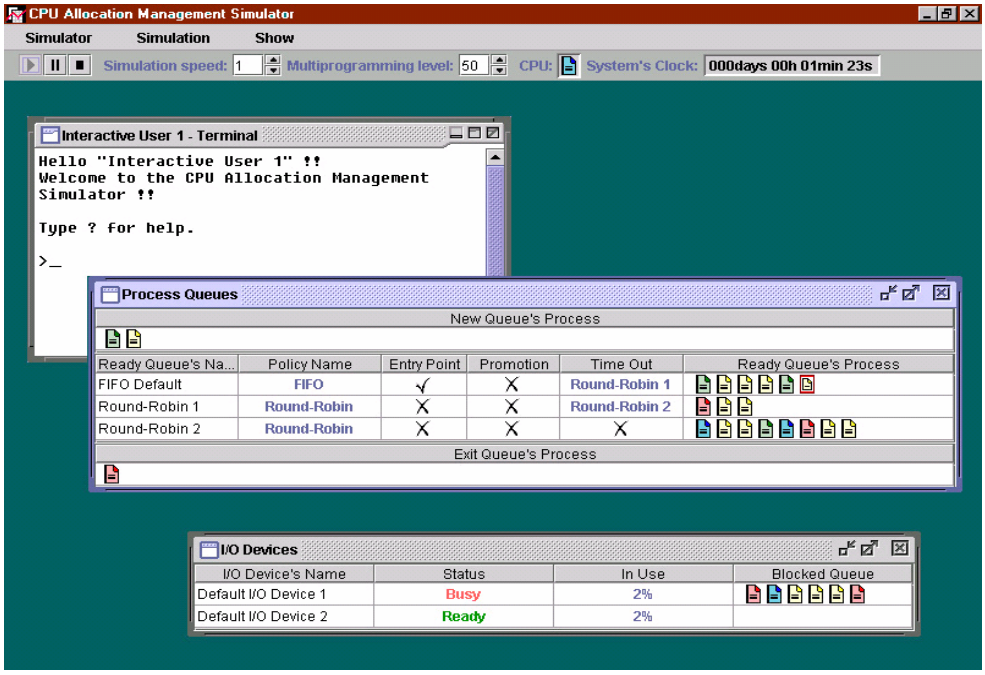

Figura 1 – Interface gráfica: *canvas* de exibição

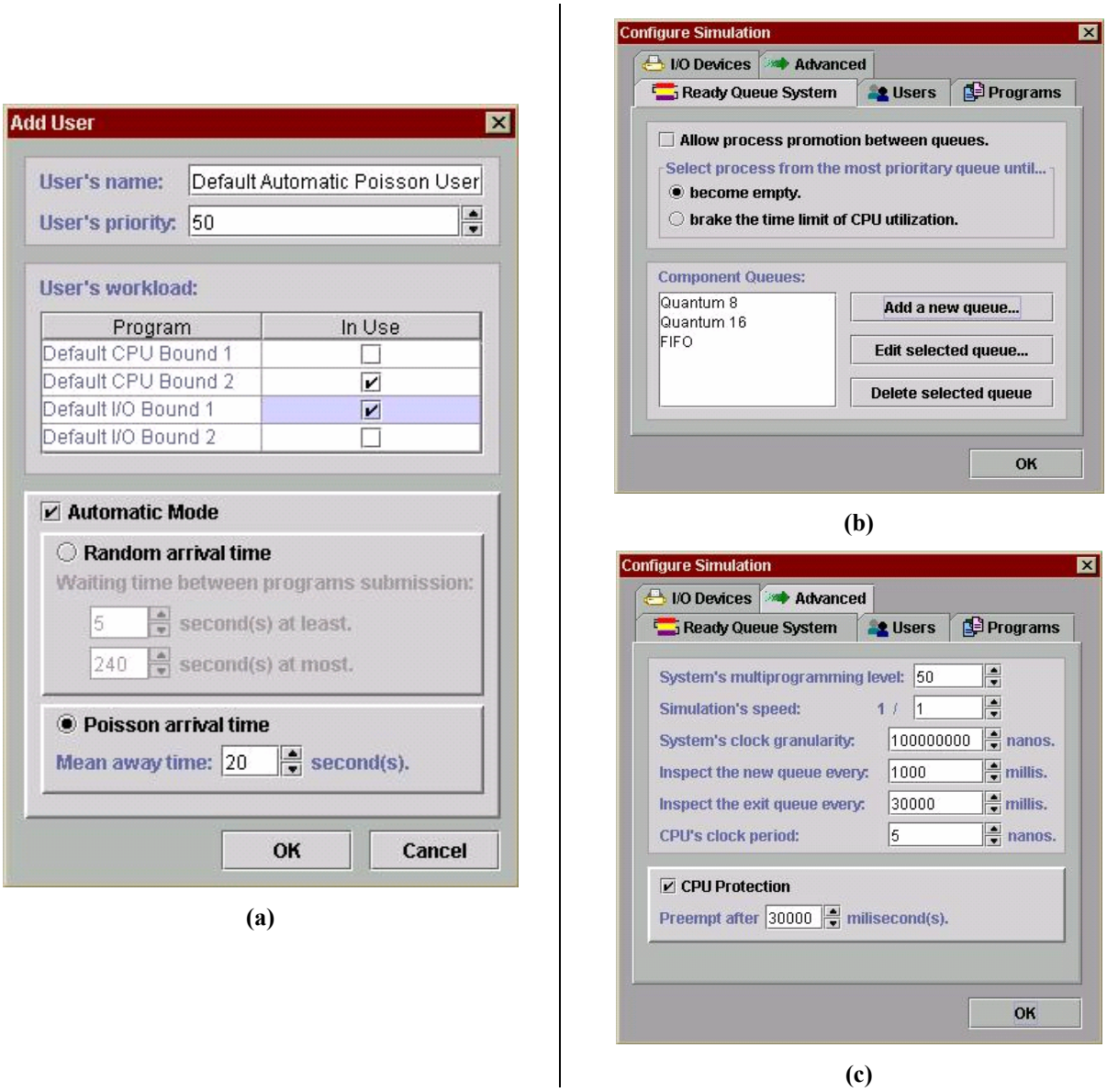

Figura 2 – Interface gráfica: painéis de (a) configuração de usuário, (b) configuração de parâmetros da simulação e (c) aspectos avançados

2. Simulando o *Hardware*

2.1. A CPU e o Conjunto de Instruções Básicas

A CPU do simulador é representada através de uma classe chamada *CPU*, que herda de *java.lang.Thread*, da API Java [11]. Dentro do seu método *run*, é inserido um *loop*, responsável por buscar uma nova instrução, executar esta instrução e verificar a ocorrência de alguma interrupção. Este *loop* é apresentado na Listagem 1. O teste que ocorre na linha 07 é explicado em maiores detalhes na Seção 2.2.

As instruções buscadas e executadas pela CPU do simulador são representadas por instâncias de classes que herdam da classe abstrata *Instruction* e são mantidas armazenadas na área de código dos programas virtuais (Seção 3.5) que são submetidos ao simulador pelos usuários virtuais (Seção 4). A classe abstrata *Instruction* e suas herdeiras estão representadas na Figura 3.

```
01 enquanto (verdade){
02     novaInstrução = buscaNovaInstrução;
03
04     executaInstrução( novaInstrução );
05
06     // Testa interrupção ou excessão:
07     se ( interruptionFlag ≠ nulo ) então {
08
09         executaTratador( interruptionFlag );
10
11         // Reabilitando as interrupções:
12         enableInterruptions;
13     }
14 }
```

Listagem 1 – *Loop* no método *run* da CPU

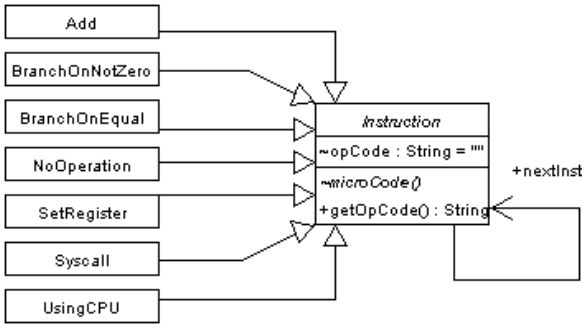

Figura 3 – Classe *Instruction* e suas herdeiras

A classe *Instruction* possui o método abstrato *microcode,* que é chamado pela CPU do simulador no instante da execução de cada instrução. Este método deve ser implementado pelas classes filhas, fornecendo as operações que devem ser realizadas pela CPU durante a execução da instrução que elas representam. Para adicionar novas instruções ao conjunto de instruções básicas da CPU do simulador, basta criar uma nova classe que herde de *Instruction* e fornecer uma implementação para o seu método abstrato *microcode.*

O campo *nextInst*, da classe *Instruction*, permite que as instruções pertencentes à área de código de um mesmo programa virtual sejam ligadas umas às outras, formando uma lista ligada de instruções. Ao final da execução de uma dada instrução "*x"*, a CPU recorre ao campo *x.nextInst* para determinar qual será a próxima instrução a ser executada.

O simulador também fornece uma instrução especial para a realização de chamadas ao sistema, a qual necessita, como parâmetro, do serviço que será solicitado ao sistema operacional. Esta instrução é representada pela classe *Syscall*. Ao instanciar um novo objeto *Syscall,* especificamos, através do seu construtor, o serviço de sistema desejado (Seção 3.2).

Um importante elemento do conjunto de instruções básicas da CPU do simulador é a instrução *UsingCPU*. Através dela, representamos consumo de ciclos de *clock* da CPU. Como parâmetro do seu construtor, informamos a quantidade de ciclos de *clock* desejados, e, no instante da sua execução, a CPU do simulador ficará ocupada por uma quantidade de tempo proporcional à quantidade de ciclos escolhida.

As instruções *Add, SetRegister, BranchOnNotZero* e *BranchOnEqual* são utilizadas pelo simulador para minimizar a quantidade de memória necessária para armazenar os programas virtuais que são executados pela sua CPU.

2.2. O Mecanismo de Interrupções

Dentre as diversas técnicas [06] existentes para implementar a sinalização entre microprocessadores e dispositivos periféricos, optamos pela implementação da técnica de *interrupções*, por satisfazer às necessidades do simulador. Como no simulador várias dispositivos periféricos precisarão interromper a CPU, necessitamos de um mecanismo que permita a identificação de qual destes dispositivos gerou uma interrupção. Como gostaríamos de permitir que o usuário do simulador disponibilizasse um número ilimitado de dispositivos periféricos para cada simulação, a opção de uma CPU dotada de múltiplas linhas de interrupção [09] não seria conveniente.

A opção adotada pelo simulador foi conectar todos os dispositivos a uma única linha de interrupção, e fazer com que a CPU verifique esta linha após a execução de cada instrução. O problema aqui é identificar a origem da interrupção. Para realizar essa identificação, são conhecidos três métodos: *software polling, daisy chaining* e a arbitragem de barramento [09]. Utilizamos no simulador um método de arbitragem de barramento sendo que, como árbitro de barramento usaremos um controlador de pedidos de interrupção [07][08][05].

No simulador, o controlador de pedidos de interrupções é implementado dentro da própria classe da CPU, através de um mecanismo de semáforos. Ele pode tratar um número indeterminado de linhas de requisição, mas não possibilita mascaramento de interrupções e não garante prioridade entre estas linhas. Ao tentar interromper a CPU, a *thread* pertencente a um dispositivo periférico executa um método da classe CPU chamado *testAndSetIntFlag,* onde tenta passar por um semáforo de nome *interruptionEnable,* conforme indicado na linha 02 da Listagem 2.

```
01 método testAndSetIntHandler(
   InterruptionHandler iHandler ){
02     // semáforo inicializado com "1":
03     interruptionEnable.P;
04     interruptionFlag <- iHandler;
05 }
```

Listagem 2 – Método *testAndSetIntHandler*

Se esta *thread* consegue passar pelo semáforo *interruptionEnable,* ela carrega a variável *interruptionFlag* com o valor do tratador que deve ser acionado (os tratadores de interrupção serão tratados adiante). Após a execução de cada instrução, a CPU verifica esta variável e, se o seu valor é diferente de nulo, é acionado o tratador que nela estará armazenado (conforme indicado na Listagem 1, linhas 06 à 13).

Quando a *thread* conseguiu passar pelo semáforo *interruptionEnable* executando nele um *wait*, ela havia acabado de desabilitar as interrupções, uma vez que nenhuma outra *thread* conseguiria cruzar este semáforo. Para então reabilitar as interrupções, ao final da execução do tratador, a CPU chama seu método *enableInterruptions*, representado na Listagem 3.

```
01 método enableInterruptions {
02     interruptionFlag <- nulo;
03     interruptionEnable.V;
04 }
```

Listagem 3 – Método *enableInterruptions*

Quando o método *enableInterruptions* é chamado, é liberada mais uma *thread* pertencente a algum disposi-

tivo periférico que, eventualmente, tenha ficado bloqueada no semáforo *interruptionEnable*, ao tentar interromper a CPU enquanto esta já tratava outra interrupção. Se mais de uma *thread* encontra-se bloqueada neste semáforo, não se pode garantir qual será escolhida para executar, uma vez que na implementação do semáforo *interruptionEnable* utiliza-se o método *notify* da classe *java.lang.Object* [11] o qual não se responsabiliza pelo critério utilizado para determinar qual *thread* será acordada [12].

Como foi dito, quando a CPU testa a linha de interrupção representada pelo campo *interruptionFlag* e verifica que ela foi ativada por algum dispositivo, o fluxo de execução é desviado para a área do sistema operacional e um tratador de interrupções conveniente é executado. No simulador, os tratadores de interrupção são implementados através de classes que herdam da classe abstrata *InterruptionHandler,* conforme ilustrado na Figura 4.

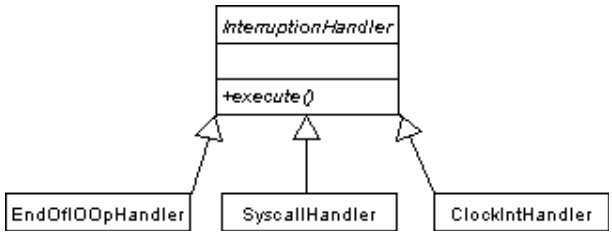

Figura 4 - Classe *InterruptionHandler* e suas herdeiras

Estes tratadores fornecem, através da implementação do método *execute,* o código que será executado quando ocorrer uma interrupção associada a este tratador. Para criar um novo tratador de interrupções, basta criar uma nova classe que herde de *InterruptionHandler* e fornecer uma implementação para o seu método abstrato *execute*. O tratador de interrupções de tempo, por exemplo, é representado pela classe *ClockIntHandler* (Seção 3.3), que herda de *InterruptionHandler* e fornece através da implementação do método *execute* o código que deve ser executado a cada interrupção provocada pelo temporizador do sistema.

Além do tratador de interrupções de tempo, a Figura 4 mostra o tratador de chamadas ao sistema (*SyscallHandler*), acionado pela instrução *Syscall* e o tratador associado as interrupções provocadas pelos dispositivo periféricos ao término de cada uma das operações de I/O (*EndOfIOOpHandler*).

2.3. O Temporizador do Sistema

São duas as técnicas possíveis para a implementação de um sistema de temporização num sistema computacional multiprogramado: a técnica baseada num *hardware* temporizador do tipo *one-shot* e a técnica baseada num *hardware* temporizador do tipo *periodic* [15][02][01]. Em ambos os casos, o papel do *hardware* temporizador é interromper o trabalho da CPU, uma vez decorrida uma determinada quantidade de tempo. A CPU detecta esta interrupção (interrupção de tempo) e transfere o fluxo de execução de instruções para o sistema operacional que, genericamente falando, irá acionar o tratador de interrupções de tempo. No caso do simulador, a implementação do sistema de temporização com os dois tipos de temporizador foram testadas e proporcionaram performance equivalente.

Quando adotamos um temporizador do tipo *periodic* para ser responsável pela geração das interrupções de tempo, devemos determinar o intervalo de tempo em que estas interrupções serão geradas levando em consideração o tempo consumido para que o seu respectivo tratador de interrupções seja executado por completo. Se não tomamos este cuidado, corremos o risco de que uma nova interrupção de tempo seja gerada antes do tratador de interrupções acionado por uma interrupção anterior ter sido finalizado. No caso do simulador, como utilizamos várias *threads* na sua implementação, seria impossível determinar quanto tempo exatamente o tratador de interrupções de tempo levaria para ser executado, pois não temos como garantir que tipo de escalonamento é efetivado pela máquina virtual Java a cada nova execução. Esse fato, a princípio, nos levaria a deixar de lado a implementação via temporizador *periodic* e partir para a implementação via temporizador *one-shot*.

No caso da implementação *one-shot*, como foi dito, o temporizador precisa, explicitamente, de ser reprogramado após o tratamento de cada interrupção. No simulador, isso implica que a *thread* responsável por executar o tratador de interrupções de tempo deveria reprogramar o temporizador instantes antes do final da sua execução. O problema com esta implementação é que, novamente, devido à incerteza em relação ao escalonamento da *threads*, não se pode determinar quanto tempo esta *thread* levaria executando, o que poderia acarretar uma imprecisão indesejada na reprogramação do temporizador e, conseqüentemente, imprecisão na temporização dos eventos.

Na tentativa de se resolver o problema causado pela incerteza no escalonamento das *threads*, poderia ser levantada a hipótese de se interromper a execução de todas as *threads*, exceto a do tratador, de forma que o sistema ficasse paralisado enquanto o tratador não completasse sua execução. Entretanto, esta prática, além de ser altamente sujeita a *dead-locks*, é de difícil implementação por diversos motivos:

- Java não fornece uma API confiável para que se suspendam as threads de modo conveniente [12];
- A implementação via semáforos seria extremamente complexa, adicionando grande *overhead* ao sistema, pois existe grande possibilidade de que muitas *threads* estejam adormecidas no momento em que o tratador tenta bloqueá-las;
- A implementação via aumento da prioridade da *thread* do tratador em detrimento das demais também não é viável pois a máquina virtual Java também não garante o funcionamento adequado de políticas de prioridade entre as *threads* visando a este tipo de utilização [13].

Aos problemas provocados pelas *threads*, some-se a ausência de uma política de prioridades entre as diversas interrupções diferentes que ocorrem na CPU do simulador, o que significa que, uma vez ocorrida uma interrupção de tempo, o seu tratador pode não ser executado de imediato, necessitando esperar a finalização de outro tratador, o que acarretaria em ainda mais imprevisibilidade quanto ao tempo gasto no tratamento de cada interrupção de tempo, tanto na abordagem *periodic* quanto na *one-shot*.

Ainda devemos levar em consideração o fato que, dependendo da eficiência do sistema real que vai rodar o simulador, o tratador de interrupções simulado irá levar mais ou menos tempo para executar. Como os intervalos entre as gerações de interrupções de tempo pelo temporizador são implementados através de suspensão de execução da *thread* do temporizador por alguns instantes, o que não sofre grande variação independentemente do sistema que roda o simulador, se torna mais difícil ainda manter o controle sobre as proporções entre as diversas medidas de tempo dentro do simulador.

Estes problemas criaram um impasse na determinação de qual tipo de temporizador seria adotado pelo simulador. No caso do *one-shot* deveríamos arcar com a imprecisão nas temporizações, acarretada pela imprevisibilidade do tempo de execução do tratador; no caso do *periodic* cairíamos no problema de determinar qual seria o intervalo no qual as interrupções de tempo seriam geradas. Escolhemos, então, adotar um temporizador *periodic* e criamos uma solução alternativa para resolver o problema da impossibilidade de determinação do intervalo de tempo de geração de interrupções ideal, evitando, deste modo, as imprecisões da abordagem *one-shot*.

O temporizador implementado gera interrupções em um intervalo de tempo configurável pelo usuário e leva em consideração tanto as diferenças de velocidade entre os sistemas reais que rodarão o simulador quanto a imprevisibilidade do tempo gasto na execução do tratador de interrupções de tempo. Ele é composto por uma única *thread* que fica executando um *loop* infinito onde desenvolve três papéis importantes:

1) Contar o tempo decorrido, o que é feito através de atualizações a uma variável que mantém a hora atual e chamadas consecutivas a um método que suspende a sua execução por um intervalo de tempo determinado.
2) Gerar *threads* filhas, que são as responsáveis por interromper a CPU de modo a executar-se o tratador de interrupções de tempo (é importante notar que não é a própria *thread* do temporizador que irá interromper a CPU e executar o tratador, e sim estas *threads* filhas geradas pelo temporizador).
3) Regular a velocidade da simulação de modo que cada *thread*-filha gerada tenha tempo suficiente para cumprir sua tarefa antes que uma nova *thread*-filha seja instanciada.

Na Listagem 4 podemos observar a representação do *loop* principal do temporizador em pseudocódigo.

Quando geramos uma nova *thread*-filha de interrupção da CPU (segundo papel), incrementamos uma variável de controle que conta quantas *threads* deste tipo estão ativas. Quando cada uma destas *threads* termina sua execução, esta variável de controle é decrementada. Deste modo, temos conhecimento de quantas *threads* de interrupção estão ativas no sistema. O ideal é que o valor armazenado por esta variável seja sempre zero ou um. Quando o valor é maior que um, significa que uma nova *thread*-filha de interrupção de tempo foi instanciada com objetivo de interromper a CPU e uma ou mais *threads* anteriores ainda não tinham terminado a sua execução.

```
01 enquanto (verdade) {
02   // Primeiro papel:
03   suspendeExecução(
          granularidadeDoTemporizador ×
          velocidadeDaSimulação);
04     atualizaVariávelDeTempo;
05     // Terceiro papel:
06     ajustaVelocidadeDaSimulação;
07     // Segundo papel:
08     geraNovaThreadDeInterrupção;
09 }
```

Listagem 4 – *Loop* principal do temporizador

Quando o temporizador está desempenhando o seu terceiro papel (ajustar a velocidade da simulação), o número de *threads* filhas ativas é levado em consideração de modo que, sempre que este valor é maior do que um, é efetuada uma diminuição na velocidade da simulação, fornecendo assim mais tempo para as *threads* filhas executarem. Quando a *thread* do temporizador volta a desempenhar o seu primeiro papel e suspende sua própria execução, a velocidade da simulação é um dos fatores utilizados na determinação de por quanto tempo se dará esta suspensão, o que indiretamente vai dar maior ou menor tempo para que as *threads* filhas de interrupção da CPU possam executar.

Utilizar a velocidade da simulação como artifício para coordenar o tempo necessário para executar um tratador e o intervalo de geração das interrupções se mostrou uma técnica interessante, pois a velocidade da simulação é um fator externo a toda a problemática da simulação, garantindo que, pelo menos em média, seja executado um tratador de interrupções de tempo a cada interrupção de tempo, sem que seja alterado o intervalo em que são geradas as interrupções. Resumidamente, o que esta técnica faz é: já que o sistema real que roda o simulador não é capaz de executar os tratadores de interrupção rápido o suficiente para que as interrupções "não se atropelem", diminuímos a velocidade da simulação até que ela se torne compatível com a capacidade de processamento deste sistema real.

2.4. Os Dispositivos de I/O

A execução de processos se dá em um ciclo no qual se alternam a execução pela CPU e a espera por I/O [10]. Para ser capaz de representar estes ciclos, o simulador

fornece dispositivos de I/O virtuais. Os processos que executam na CPU do simulador podem, através de chamadas ao sistema (instruções *Syscall*, descritas na Seção 2.1) solicitar um serviço de requisição de operação de I/O (Seção 3.2) e serão então bloqueados enquanto a operação se processa.

Os dispositivos de I/O são representados pelo simulador através da classe *IODevice*, que implementa a interface *java.lang.Runnable* [11], permitindo que cada dispositivo seja executado por uma *thread* própria. Cada *thread* permanece num *loop* infinito onde aguarda para executar uma nova operação de I/O, executa a operação e em seguida interrompe a CPU, avisando sobre o final da operação.

As operações que o dispositivo de I/O podem executar são representadas por classes que herdam de uma classe abstrata chamada *IOOperation*. As classes que representam operações de I/O devem implementar o método abstrato *execute* de *IOOperation*. Este método será chamado pelo dispositivo de I/O no momento da execução desta operação.

O simulador oferece por *default* duas operações de I/O, representadas pelas classes *IORead* e *IOWrite*. Ambas possuem um campo usado para determinar quanto tempo levará a operação de I/O. Fica a cargo do usuário do simulador determinar o valor destes dois campos, possibilitando que sejam criados dispositivos mais velozes ou mais lentos, aumentando, assim, a possibilidade de diferentes cenários de simulação.

2.5. A Velocidade da Simulação

No simulador existem alguns componentes que, periodicamente, efetuam "pausas" de modo a representar alguma característica de um sistema real. Por exemplo: num sistema real, os dispositivos de I/O, periodicamente, realizam operações de leitura ou escrita. Essas operações, por sua vez, levam algum tempo até serem completadas, e este tempo precisa ser representado de alguma forma na simulação. De forma análoga, a CPU consome tempo executando instruções, e os usuários do sistema consomem tempo efetuando diversas atividades fora do escopo do sistema, entre as submissões de seus programas à execução.

Durante a implementação do simulador, quando encontramos um componente deste tipo, utilizamos o método "*sleep*", fornecido pela classe *Thread* do pacote *java.lang* [11] para representar o tempo que o componente deve ficar paralisado. Este método, requer um parâmetro que indica por quanto tempo se dará esta paralisação. Vamos chamar este parâmetro de *p*. A Listagem 5 esquematiza o *loop* principal de um dispositivo de I/O do sistema, como exemplo.

No simulador, o parâmetro "*p"* pode ser expresso por um número na forma $a \times b$, onde "*a"* é efetivamente o tempo que deve ser gasto pelo componente, e "*b*" é um fator usado para ajustar a velocidade da simulação. Ou seja: $p = a \times b$, e a velocidade da simulação é $1/b$. Se na Listagem 5, o dispositivo de I/O leva 5 unidades de tempo para executar uma operação, e a velocidade da simulação é 1/2, a variável "*a"* valeria 5 e "*b*" valeria 2, assim teríamos $p = 10$.

```
01 enquanto( verdade )
02   {
03   aguardaRequisiçãoDeOperação;
04
05   // Simulando a execução da operação:
06   Thread.sleep( p );
07
08   sinalizaFinalDaOperação;
09   }
```

Listagem 5 – *Loop* principal em um dispositivo de I/O

Usando este padrão, permitimos ao usuário diminuir ou aumentar a velocidade da simulação sem que para isso, sejam diretamente alterados valores como o tempo gasto em cada operação de I/O, ou o comprimento do ciclo de *clock* da CPU, por exemplo. Basta atualizar o fator "*b*", e automaticamente cada componente irá efetuar uma pausa maior ou menor, mas sempre proporcional ao fator "*a*".

3. O Sistema Operacional Virtual

"Um sistema operacional é principalmente um gerenciador de recursos, e o principal recurso gerenciado é o *hardware* computacional" [04]. Para gerenciar o *hardware* virtual do simulador, criamos um sistema operacional virtual que será descrito nesta seção. Começaremos analisando o micronúcleo do sistema operacional virtual e em seguida iremos apresentar os demais subsistemas que colaboram para que o núcleo cumpra seu principal objetivo neste simulador: gerenciar a CPU.

3.1. O Micronúcleo

O conjunto de serviços oferecido pelo sistema operacional do simulador é bastante reduzido em relação ao de um sistema operacional real, praticamente limitando-se a se preocupar com a questão do escalonamento de processos. Assim mesmo, nos inspiramos no modelo de micronúcleo para criar um sistema operacional modularizado, permitindo que os diversos mecanismos de alocação da CPU disponíveis pudessem ser intercambiáveis sem que, para isso, o núcleo do sistema operacional precisasse conhecer estes mecanismos. Utilizamos os conceitos de herança e polimorfismo inerentes à linguagem orientada a objetos através da qual foi implementado o simulador, para viabilizar esta idéia.

O sistema operacional do simulador é formado pelo conjunto de classes pertencentes a um pacote chamado *cpumngtsim.os*. Nas classes deste pacote são implementadas as políticas de alocação da CPU, os tratadores de interrupção, os serviços que podem ser acessados através de chamadas ao sistema, as filas de processos e outros componentes necessários ao gerenciamento do sistema computacional virtual. Dentre estas classes, encontramos a classe *Kernel* (Figura 5), que representa o micronúcleo do sistema operacional do simulador.

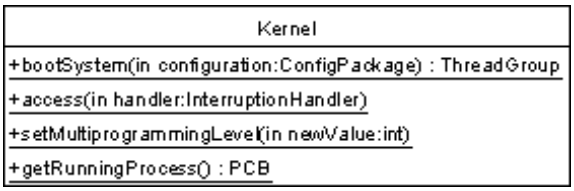

Figura 5 – Classe *Kernel*

Os parâmetros de uma simulação são encapsulados numa classe *ConfigPackage* e são enviados como argumento do método *bootSystem* da classe *Kernel.* Baseado nestes parâmetros, o núcleo então dispara as *threads* responsáveis pela simulação e retorna ao chamador uma classe *java.lang.ThreadGroup* [11] contendo referências para estas *threads* recém-criadas.

```
01 método access( InterruptionHandler iHandler ){
02     salvaContexto( processoEmExecução );
03     iHandler.execute;
04     próximoParaExecutar =
   sistemaDeFilasDeProntos.obterPróximo;
05     carregaContexto( próximoParaExecutar );
06 }
```

Listagem 6 – O método *access*

O mecanismo de interrupções implementado pelo simulador não utiliza interrupções vetoradas, dando preferência a um endereço padrão de desvio para tratamento de interrupções. Este ponto de entrada padrão é representado pelo método *access,* também da classe *Kernel.* Este método está especificado em pseudocódigo na Listagem 6.

Sempre que ocorre uma interrupção, a CPU desvia o fluxo de execução de instruções para o ponto de entrada padrão fazendo uma chamada ao método *access* que recebe como parâmetro o tratador de interrupções que deve ser acionado (essa chamada foi representada em pseudocódigo na linha 08 da Listagem 1, Seção 2.1).

É interessante notar que o método *access* espera como argumento uma classe *InterruptionHandle* que é a classe pai da hierarquia de tratadores. Isso significa que, através de polimorfismo, o método *access* será capaz de proporcionar um tratamento diferenciado para cada tipo de interrupção que ocorre no sistema.

Como foi visto na Seção 2.2, ao interromper a CPU, carregamos, no seu campo *interruptionFlag*, o tratador de interrupções para o qual devemos desviar o fluxo de execução de instruções no momento da ocorrência desta interrupção. É desta forma que a CPU, ao detectar esta interrupção, sabe qual argumento usar na chamada ao método *access.*

Conforme indicado na Listagem 6, uma vez invocado o método *access,* ele salva o contexto do processo que vinha executando (linha 02), executa o devido tratador (linha 03) e através de métodos convenientes, existentes no sistema de filas de prontos configurado pelo usuário do simulador, obtém o próximo processo, digamos "*p*", que deverá entrar em execução ao final do tratamento da interrupção (linha 04). Por fim, basta recarregar o contexto de "*p*" de forma a retomar sua execução (linha 05).

Vale notar que para classe *Kernel*, ou seja, o micronúcleo do sistema operacional virtual do simulador, todos os algoritmos de escalonamento são transparentes. A decisão sobre qual será o processo selecionado para execução recai sobre as classes responsáveis por implementar as filas de processos (Seção 3.6) e as políticas de alocação da CPU (Seção 3.9).

3.2. Os Serviços do Sistema Operacional

Quando um processo precisa requisitar um serviço de sistema, ele utiliza uma instrução *Syscall.* O construtor da instrução *Syscall* recebe como argumento uma instância de uma classe que implemente a interface *SystemService.* A interface *SystemService* assim como suas implementações é mostrada na Figura 6.

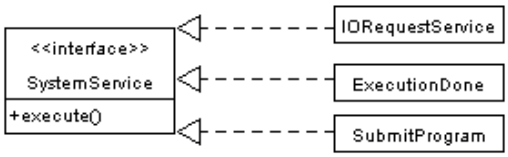

Figura 6 – Interface *System Service* e suas implementações

Observamos que esta interface define um único método, o *execute*, que deve ser implementado pelas classes que se propuserem a definir um serviço de sistema. No momento da execução de uma instrução *Syscall*, a CPU do simulador comporta-se como tivesse sido interrompida, desviando o fluxo de execução de instruções para a área do sistema operacional através de uma chamada "*Kernel.access(new SyscallHandler)*" onde *SyscallHandler* é o tratador de chamadas ao sistema. O método *execute* será invocado por este tratador de chamadas ao sistema, desencadeando a execução do serviço.

Para requisitar uma operação de I/O, um processo deve possuir na sua área de código uma instrução *Syscall* instanciada usando um objeto da classe *IORequestService* como argumento. *IORequestService* é a classe que representa o serviço do sistema operacional de requisição I/O e possui dois parâmetros no seu construtor: um número de porta e uma classe *IOOperation*. O número de porta serve para indicar qual será o dispositivo de I/O alvo da requisição e *IOOperation* indica qual operação será solicitada a este dispositivo (Seção 2.4).

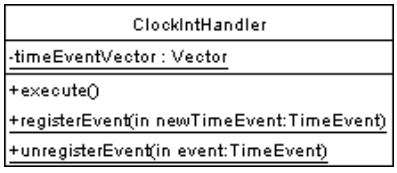

Figura 7 – Classe *ClockIntHandler*

O método *execute* de *IORequestService* é responsável por encaminhar o processo que fez a requisição para a fila de processos bloqueados referente ao disposi-

tivo instalado na porta especificada, escrever nos registradores do dispositivo os dados para dar início à realização da operação de I/O, atualizar as informações da tabela de *status* de dispositivo do sistema operacional e sinalizar ao escalonador do sistema a necessidade de selecionar um novo processo para execução já que o atual está bloqueado.

Figura 8 – Classe *TimeEvent*

Outra classe que implementa a interface *SystemService* é a *ExecutionDone*. Ela representa o serviço do sistema operacional que deve ser requisitado por todos os processos do sistema ao concluírem sua execução. O código executado pelo seu método *execute* é bem simples: resume-se a encaminhar o processo terminado para a fila de saída do sistema, atualizar informações de contabilização e assim como o *IORequestService* sinalizar ao escalonador do sistema a necessidade de selecionar um novo processo para execução, já que o atual foi concluído.

Por fim, o último serviço de sistema oferecido pelo simulador é o de criação de um novo processo. Ele é representado pela classe *SubmitProgram* que através do seu método *execute,* se responsabiliza por criar um novo *PCB* (*Process Control Block*) para armazenar as informações do processo que está sendo criado e o encaminha para a fila de entrada do sistema. Este serviço é solicitado pelos usuários virtuais do simulador no momento em que submetem seus programas à execução.

3.3. O Tratador de Interrupções de Tempo

Realizamos a implementação do tratador de interrupções de tempo baseada em lista ordenada, já que as vantagens oferecidas pelas outras estruturas de dados mais elaboradas [03][16], como a escalabilidade, não seriam aproveitadas pois no simulador muito poucos *timers*[1] são utilizados. O nosso tratador de interrupções de tempo (Figura 7) foi implementado através da classe *ClockIntHandler* que fornece métodos para registrar e excluir *timers*.

Um usuário que deseje utilizar a facilidade de *timer* deve criar uma nova classe que herde da classe abstrata *TimeEvent*, (Figura 8) e implementar o seu método *execute*. No seu construtor, existe um parâmetro que indica após quanto tempo esse *timer* vai expirar. A classe filha vai encapsular os dados e métodos necessários a resolução do problema do usuário, e pode ser registrada no tratador de interrupções de tempo do simulador através do método *registerEvent* fornecido pela classe *ClockIntHandler*.

A cada *tick* do temporizador do sistema, é gerada uma interrupção de tempo e o fluxo de execução é desviado para a área do sistema operacional, quando então o tratador de interrupções de tempo começa a ser executado. No caso do simulador, o seu tratador possui dois papéis simples:

1) Ajustar o campo *time* do primeiro *timer* da lista, de modo a refletir a passagem do tempo.
2) Verificar se existem *timers* expirados, chamando seus métodos *execute* em caso positivo.

A hora atual é mantida pelo próprio temporizador do simulador, livrando o tratador deste papel.

O método *registerEvent* do tratador varre a lista de *timers* a partir das primeiras posições, procurando a posição correta para adicionar o novo *timer*. Para cada elemento da lista, o método compara seu campo *time* com o do novo *timer,* que será registrado. Se o campo do elemento da lista for menor ou igual ao do novo *timer*, subtraímos do campo do novo *timer* o valor do campo *time* do elemento da lista e continuamos a busca. Se ele for maior, já encontramos a posição onde o novo *timer* será inserido, e a busca está finalizada.

O método *unregisterEvent* varre a lista procurando pelo *timer* passado como parâmetro. Se ele é encontrado, será removido da lista.

3.4. O Tratador de Interrupções de I/O

Como foi dito na Seção 2.2, os tratadores de interrupções provocadas por final de operação de dispositivos de I/O são representados por instâncias da classe *EndOfIOOpHandler*. O construtor desta classe requer como argumento o dispositivo que receberá o tratamento. Desta forma, quando um dispositivo de I/O conclui uma operação a ele requisitada, deve tentar interromper a CPU conforme indicado na Listagem 7. Através da passagem para o construtor de uma referência ao dispositivo que terminou a operação de I/O, podemos instanciar um tratador consciente do dispositivo que deverá ser atendido.

```
01    // "esseDispositivo" contém uma
      referência para o dispositivo que está
      tentando interromper a CPU:
02
03    CPU.testAndSetIntFlag( new
      EndOfIOOpHandler( esseDispositivo ) );
04
```

Listagem 7 – Fim de operação de I/O

Uma vez que este tratador entre em execução, ele irá primeiramente liberar o dispositivo em questão para o atendimento de uma outra requisição, em seguida atualiza a tabela de *status* de dispositivo do sistema operacio-

[1] Conforme em [03][16], chamamos de *timer* o recurso oferecido pelo sistema operacional que permite a programação de eventos que ocorrerão dentro de um intervalo de tempo predeterminado pelo usuário.

nal e o PCB do processo que havia sido bloqueado aguardando o final da operação de I/O. Este processo então é reencaminhado para sua fila de prontos de origem, ou seja, a última fila de prontos em que ele esteve antes de ser escalado para execução pela última vez.

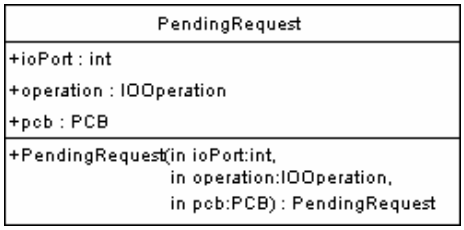

Figura 9 – Classe *PendingRequest*

Por fim, o tratador varre a tabela de *status* de dispositivo em busca de alguma outra requisição de I/O que esteja pendente aguardando a disponibilidade do dispositivo de I/O para atendê-la. As requisições de I/O pendentes são representadas através de instâncias da classe *PendingRequest*, pertencente ao pacote *cpumngtsim.os* (Figura 9).

O campo *ioPort* de *PendingRequest* armazena a porta para a qual está sendo feita a requisição. No simulador, a porta nada mais é do que um índice para acessarmos o *array* onde são armazenados os diversos dispositivos de I/O que estão sendo utilizados na simulação. O campo *operation*, indica qual operação será solicitada ao dispositivo e o campo *pcb* indica qual processo gerou esta requisição pendente.

As requisições pendentes, ou seja, os objetos *PendingRequest* são manipulados por instâncias da classe *DeviceStatusInfo*, também pertencente ao pacote *cpumngtsim.os* (Figura 10). A classe *DeviceStatusInfo* representa um conjunto de informações de *status* de um determinado dispositivo de I/O. Ela possui um campo do tipo *java.util.Vector* onde são armazenados diversos objetos *PendingRequest* que representam as requisições pendentes para este dispositivo. O campo *busy* indica se este dispositivo está ocupado no momento e o campo *pid* indica o PID do processo que está sendo atendido, caso o dispositivo esteja ocupado.

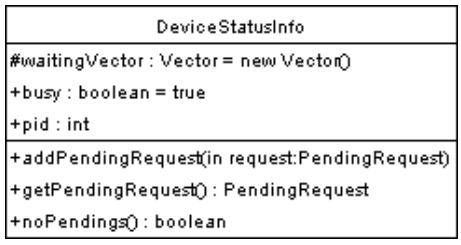

Figura 10 – Classe *DeviceStatusInfo*

A tabela de *status* de dispositivo do sistema operacional é implementada através de um *array* contendo uma entrada para cada dispositivo de I/O presente na simulação, onde em cada uma delas é armazenada uma instância da classe *DeviceStatusInfo*.

3.5. Os Programas Virtuais

Os programas reconhecidos pelo sistema operacional virtual do simulador são representados por instâncias da classe *Program* (Figura 11), pertencente ao pacote *cpumngtsim.os*. A classe *Program* possui um campo *name* que define o nome do programa e um campo *codeArea,* do tipo *Instruction* que aponta para a primeira instrução pertencente à lista encadeada de instruções que compõem a área de código deste programa. Para economizar memória, todos os programas virtuais disponibilizados por *default* aos usuários do simulador compartilham diversas seções da área de código.

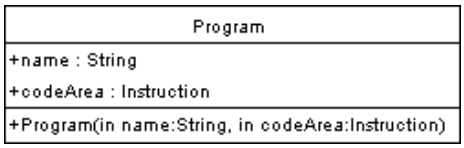

Figura 11 – Classe *Program*

Para possibilitar esse compartilhamento, são fornecidos quatro blocos principais de código, compostos pelas instruções descritas na Seção 2.1, chamados de blocos *A, B, C e D*. Os blocos *A* e *B* utilizam intensamente os dispositivos de I/O, enquanto os blocos *C* e *D* utilizam intensamente a CPU do simulador. Os programas então executam estes blocos principais por diversas vezes, em seqüência pré-determinada.

3.6. As Filas de Processos

Dentro do pacote *cpumngtsim.os*, criamos o pacote *queues*, responsável por armazenar as classes que representam as filas de processos utilizadas pelo simulador e as classes relacionadas a estas filas. Dentro do pacote *cpumngtsim.os.queues*, encontramos a classe *ProcessQueue*. As filas de processos do simulador são representadas por instâncias de *ProcessQueue* ou de classes que herdam de *ProcessQueue*, como veremos adiante.

A classe *ProcessQueue* armazena os PCBs dos processos a ela alocados no campo *pcbVector*, do tipo *java.util.Vector*. Em seguida, fornece uma série de métodos convenientes para manipular esta fila de processos, permitindo entre outras ações, adicionar e remover processos.

3.7. As Filas de Dispositivo

No simulador, as filas de dispositivo são representadas por instâncias da classe *BlockedQueue*, que herda de *ProcessQueue*, conforme indicado na Figura 12. A classe *BlockedQueue* estende a funcionalidade de *ProcessQueue* fornecendo um método para remover um processo da fila a partir do seu PID.

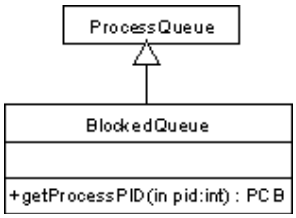

Figura 12 – Classe *BlockedQueue*

A classe *Kernel* contém um campo do tipo *array* onde cada posição é preenchida com uma instância de *BlockedQueue*. O comprimento deste *array* é igual ao número de dispositivos de I/O existentes no sistema. Quando um processo faz uma solicitação de I/O para um dispositivo que se encontra ocupado atendendo a outro processo, seu PCB é inserido na fila de dispositivo pertencente à posição do *array* referente ao dispositivo desejado.

3.8. As Filas de Processos Prontos

Outra classe pertencente ao pacote *queues* é a *ReadyQueue* (Figura 13). Assim como *BlockedQueue*, a classe *ReadyQueue* também estende a funcionalidade de *ProcessQueue,* desta vez, adicionando uma série de campos e sobrescrevendo os seus métodos *addProcess* e *getProcess*.

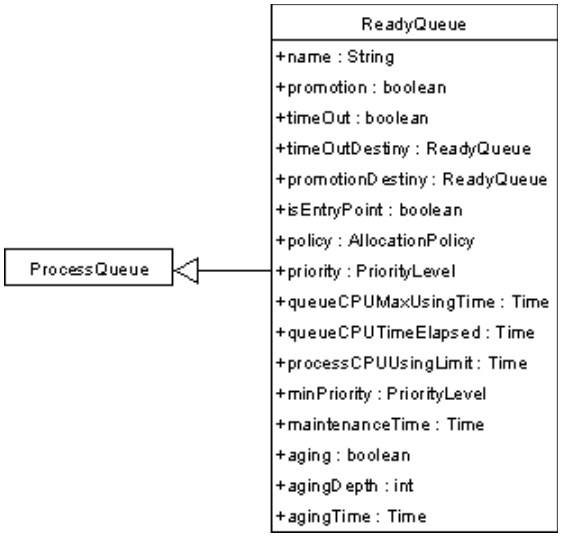

Figura 13 – Classe *ReadyQueue*

No simulador, dizemos que as filas de processos prontos são governadas pelas *políticas de alocação da CPU*. As políticas de alocação da CPU conhecem os algoritmos utilizados para adicionar e remover processos das filas de prontos. Por este motivo acrescentamos à classe *ReadyQueue* o campo *policy* que define a política de alocação da CPU que será usada para governar esta fila.

O campo *name*, de *ReadyQueue*, permite que o usuário do simulador dê um nome para a fila de prontos, facilitando sua identificação no sistema. O campo *maintenanceTime* é tratado na Seção 3.15.

Os demais campos guardam informações importantes ou para permitir que a fila de prontos seja governada eficientemente pela política de alocação da CPU determinada para ela, ou para facilitar o funcionamento do sistema de filas de prontos no qual a fila está instalada.

Os métodos *addProcess* e *getProcess* são sobrescritos para permitir que a política de alocação da CPU responsável por esta fila possa efetivamente operar sobre o campo *pcbVector* onde são armazenados os PCBs dos processos pertencentes a ela. A Listagem 8 e Listagem 9 mostram, respectivamente, uma visão simplificada dos métodos *addProcess* e *getProcess* conforme a implementação de *ReadyQueue.*

Na Listagem 8, notamos que, quando o método *addProcess(pcb)* é invocado, utilizamos o campo *policy* da fila de prontos para efetivamente adicionar *pcb* à *pcbVector*. Se *policy*, por exemplo, contém uma política FIFO (*first-in-first-out*), *pcb* será adicionado na última posição de *pcbVector*.

```
01 método addProcess(PCB pcb) {
02     policy.addingAlgorithm(pcb, pcbVector);
03 }
```

Listagem 8 – Método *addProcess* da classe *ReadyQueue*

Na Listagem 9, notamos que, analogamente, quando o método *getProcess* é invocado, utilizamos o campo *policy* da fila de prontos para efetivamente retirar um PCB de *pcbVector*. Se *policy*, por exemplo, contém uma política FIFO (*first-in-first-out*), será removido o PCB pertencente à primeira posição de *pcbVector*. Se *policy* contém uma política de alocação por prioridade, será retirado o PCB pertencente ao processo mais prioritário.

O método *getProcess* é chamado pelo escalonador da CPU quando a CPU está ociosa é necessário escolher um novo processo para entrar em execução.

```
01 método PCB getProcess {
02     retorna policy.obtainingAlgorithm(
   pcbVector );
03 }
```

Listagem 9 – Método *getProcess* da classe *ReadyQueue*

Sobrescrevendo os métodos conforme indicado, deixamos toda a responsabilidade sobre como adicionar e remover PCBs às filas de prontos para os métodos *addingAlgorithm* e *obtainingAlgorithm*, pertencentes à política de alocação que governa a fila de prontos.

3.9. As Políticas de Alocação da CPU

Ainda dentro do pacote *cpumngtsim.os.queues*, encontramos a classe abstrata *AllocationPolicy* (Figura 14). Para criar uma nova política de alocação da CPU, devemos criar uma classe que herde de *AllocationPolicy*, fornecendo uma implementação para os seus métodos abstratos *addingAlgorithm* e *obtainingAlgorithm.*

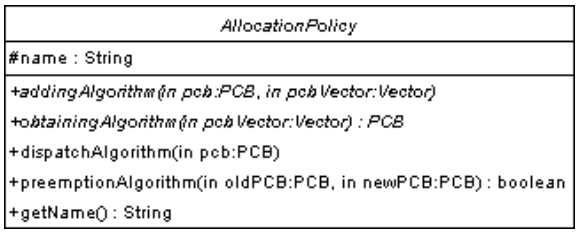

Figura 14 – Classe *AllocationPolicy*

O método *addingAlgorithm* possui dois parâmetros: um *PCB* e um *java.util.Vector*. As implementações deste método devem fornecer um algoritmo conveniente para adicionar o PCB passado como primeiro argumento ao *Vector* passado como segundo argumento, de acordo com a política representada pela classe que está implementando o método.

Quando uma fila de prontos recebe uma invocação solicitando que lhe seja adicionado um PCB, esta fila irá chamar *addingAlgorithm* fornecendo o PCB em questão como primeiro argumento e o seu campo *pcbVector* como segundo argumento.

O método *obtainingAlgorithm* possui um único parâmetro: um *java.util.Vector*. As implementações deste método devem fornecer um algoritmo conveniente para escolher, dentre os PCBs pertencentes ao *Vector* passado como argumento, um deles para ser removido da fila. Essa escolha é feita de acordo com a política representada pela classe que implementa este método.

Quando uma fila de prontos, recebe uma invocação solicitando que dela seja removido um PCB (esta invocação é feita pelo escalonador da CPU, no momento da escolha de um novo processo para executar), esta fila irá chamar *obtainingAlgorithm* fornecendo o seu campo *pcbVector* como argumento.

Além destes dois métodos abstratos, a classe *AllocationPolicy* ainda possui outros dois métodos importantes para o funcionamento de uma política de alocação da CPU. São eles o método *dispatchAlgorithm* e *preemptionAlgorithm*.

Em *AllocationPolicy*, o método *dispatchAlgorithm* não executa nenhum código. Este método deve ser sobrescrito pelas políticas de alocação que necessitam de algum código específico para o despachante do sistema operacional executar ao colocar um novo processo em execução. O método *dispatchAlgorithm* espera como parâmetro o PCB do processo que será colocado em execução. Algumas políticas podem não ter necessidade de realizar nenhuma operação especial no momento do despacho de novos processos, mas políticas como a *round-robin* precisam, por exemplo, de preparar o temporizador do sistema para retirar o processo de execução assim que o seu *quantum* de tempo tiver expirado.

O método *preemptionAlgorithm* deve ser sobrescrito pelas políticas que podem possuir um comportamento preemptivo. Nestas políticas, certamente será necessário um algoritmo para decidir quando um determinado processo que estava em execução deve ceder a CPU a um outro que acabou de entrar na fila de prontos do sistema. Este método deverá então ser sobrescrito de modo a satisfazer essa necessidade.

Periodicamente, o sistema operacional verifica se existem processos aguardando, na fila de entrada do sistema[2], a sua transferência para alguma fila de prontos. Se existem processos nesta situação, o sistema operacional irá decidir qual fila de prontos[3] irá hospedar estes processos e transferi-los para ela. Uma vez decidida qual fila irá hospedar o processo, o sistema irá invocar o método *preemptionAlgorithm* da política que governa esta fila para que esse método informe se o novo processo que está vindo da fila de entrada deve tomar o lugar do processo que já se encontrava em execução.

Políticas não preemptivas, não necessitam sobrescrever este método. No entanto, políticas que dão suporte ao modo preemptivo de governo, devem sobrescrevê-lo fornecendo o código que irá decidir qual dos dois processos em questão deve tomar posse da CPU: o que já estava em execução (representado pelo parâmetro *oldPCB*) ou o processo que está vindo da fila de entrada (representado por *newPCB*).

3.10. Política FIFO

A mais simples das políticas de alocação suportadas pelo simulador é a política FIFO. Ela é representada pela classe FIFO, que sobrescreve os métodos *addingAlgorithm* e *obtainingAlgorithm* de modo a implementar no *Vector pcbVector* (passado como parâmetro a estes métodos) uma fila, ou seja, sempre que é solicitado um PCB através de *obtainingAlgorithm,* retornamos o primeiro PCB pertencente ao *Vector*. Quando é adicionado um PCB através de *addingAlgorithm,* o fazemos no final do *Vector*. Os demais métodos de *AllocationPolicy* não são usados por FIFO e por isso não foram sobrescritos.

3.11. Política Round-Robin

Outra política de alocação da CPU oferecida pelo simulador é a política *round-robin*, representada pela classe *RoundRobin* (Figura 15). Por comportar-se, basicamente, como a política FIFO, a classe *RoundRobin* herda da classe *FIFO* estendendo sua funcionalidade através do mecanismo de *quantum* de tempo implementado utilizando os recursos de temporização descritos na Seção 2.3 e na Seção 3.3.

---

[2] No simulador, utilizamos uma fila de entrada, onde processos recém submetidos a execução são armazenados esperando que o sistema operacional transfira-os para uma fila de prontos.

[3] O sistema irá decidir para *qual* fila de prontos será transferido o processos pois o simulador prevê a existência de mecanismos de alocação da CPU que utilizam mais de uma fila de prontos (Seção 3.14 - O Mecanismo de Alocação da CPU).

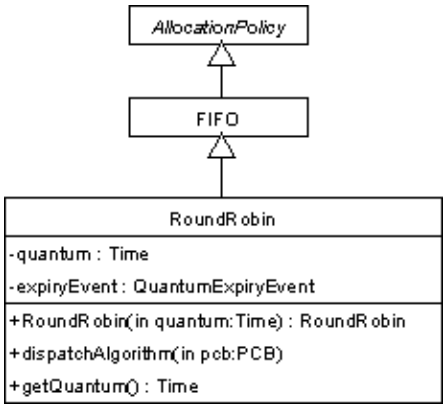

Figura 15 – Classe *RoundRobin*

O comprimento do *quantum* de tempo é armazenado no campo *quantum*. No campo *expiryEvent* será armazenada uma instância da classe *QuantumExpiryEvent.* A classe *QuantumExpiryEvent* herda de *TimeEvent* (Seção 3.3) e fornece uma implementação para o método *execute*, responsável por tirar de execução um processo que eventualmente venha a ultrapassar o *quantum* de tempo reservado para a sua utilização da CPU.

O método *dispatchAlgorithm* de *AllocationPolicy* é sobrescrito conforme indicado na Listagem 10. Uma nova instância de *QuantumExpiryEvent* é atribuída ao campo *expiryEvent* na linha 04. No construtor de *QuantumExpiryEvent* indicamos que o processo representado pelo parâmetro *pcb* será interrompido ao término do *quantum* de tempo indicado pelo parâmetro *quantum*.

```
01 método dispatchAlgorithm( PCB pcb ) {
02     ClockIntHandler.unregisterEvent(
   expiryEvent );
03
04     expiryEvent = new QuantumExpiryEvent(
   quantum, pcb );
05
06    ClockIntHandler.registerEvent(
   expiryEvent);
07 }
```

Listagem 10 – Método *dispatchAlgorithm* em *RoundRobin*

Em seguida, na linha 06, registramos *expiryEvent* em *ClockIntHandler* (Seção 3.3). Com isso, o temporizador do sistema será programado para que ao final do *quantum* de tempo determinado, seja executado o método *execute* de *expiryEvent*.

Examinando agora a linha 02, verificamos que antes de programar o temporizador do sistema para interromper o processo que será despachado, tentamos remover a instância de *QuantumExpiryEvent* que havia sido armazenada pelo próprio método *dispatchAlgorithm* no campo *expiryEvent* em uma invocação anterior. Tomamos essa providência para evitar que, no caso de um processo liberar a CPU antes do final do seu *quantum* de tempo, o tratador de interrupções de tempo não invoque o método *execute* de uma instância antiga de *QuantumExpiryEvent* indevidamente.

3.12. Escalonamento por Prioridade

O campo *priority* da classe PCB é usado pela política de alocação da CPU representada pela classe *PrioritySched* (Figura 16). A classe *PrioritySched* (Figura 16) implementa uma política de escalonamento por prioridade onde os processos com o maior valor no campo *priority* do seu PCB têm maior prioridade para uso da CPU do que os com valor menor neste campo.

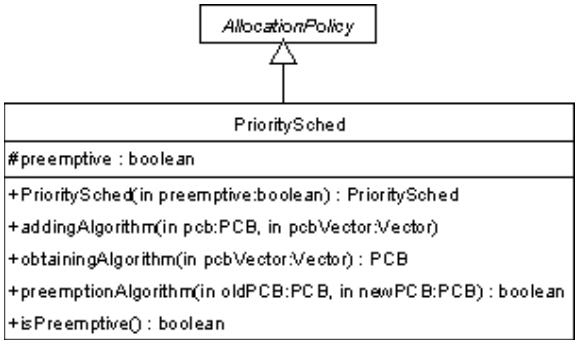

Figura 16 - Classe *PrioritySched*

O método *addingAlgorithm* de *AllocationPolicy* (Seção 3.9) é sobrescrito de forma que, quando um fila de prontos o invoca passando como parâmetros o PCB *pcb* e o *java.util.Vector pcbVector*, ele irá adicionar *pcb* ao *pcbVector* de forma que nas primeiras posições de *pcbVector* fiquem sempre os PCBs mais prioritários.

O método *obtainingAlgorithm* de *AllocationPolicy* é sobrescrito de forma que, quando um fila de prontos o invoca passando como parâmetro o *java.util.Vector pcbVector*, ele irá retornar ao chamador o primeiro PCB de *pcbVector*. Isso corresponde a retornar o PCB do processo mais prioritário, uma vez que *addingAlgorithm* mantém *pcbVector* ordenado por prioridade.

O escalonamento definido em *PrioritySched* admite tanto o modo preemptivo de governo quando o não-preemptivo. Quem determina qual o modo adotado é o campo *preemptive*. Se o modo preemptivo está sendo usado, precisamos determinar quando um processo tem poder de interromper a execução de um outro que já tinha obtido a posse da CPU. Para isso, sobrescrevemos o método *preemptionAlgorithm* definido em *AllocationPolicy*. A implementação do método *preemptionAlgorithm* por *PrioritySched* retorna *true* (ou seja, devemos interromper a execução de um processo por preempção) somente quando a prioridade do processo que pretende obter a posse da CPU é maior do que a do processo que já estava em execução.

3.13. Política SJF

De modo análogo à classe *PrioritySched*, a classe SJF (*shortest-job-first* [14]) (Figura 17) implementa uma política de escalonamento por prioridade, sendo que, em vez de basear-se no campo *priority* da classe PCB, baseia-se no campo *nextCPUUseEstimate*, também da classe PCB no qual fica armazenada uma estimativa de duração da próxima fase de uso da CPU para o processo dono deste PCB. Esta política é denominada "SJF" .

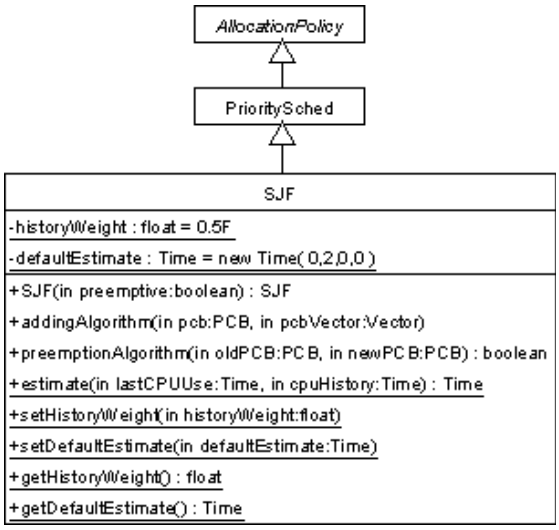

Figura 17 – Classe SJF

O método *addingAlgorithm* de *AllocationPolicy* (Seção 3.9) é sobrescrito de forma análoga à *PrioritySched*, sendo que, em SJF, os PCBs são mantidos organizados em ordem inversa de estimativas de duração de próximas fases de uso da CPU, através de seus campos *nextCPUUseEstimate*. Assim, *obtainingAlgorithm* sempre devolve o PCB pertencente ao processo com menor estimativa.

Vale observar que, como SJF herda de *PrioritySched*, possui o campo *preemptive* que define se o modo preemptivo de governo está sendo usado. Se o modo preemptivo está ativado, o método *preemptionAlgorithm* de SJF devolve *true* (ou seja, devemos interromper a execução de um processo por preempção) somente quando a estimativa de duração de próxima fase de uso da CPU do processo que pretende obter a posse da CPU é menor do que a do processo que já estava em execução.

3.14. O Mecanismo de Alocação da CPU

No simulador, consideramos que para o sistema operacional cumprir a tarefa de gerenciamento da CPU ele se utiliza de um *mecanismo de alocação da CPU*. Este mecanismo de alocação da CPU é formado por um *sistema de filas de processos prontos* e uma *política de gerenciamento de sistemas de filas de processos prontos*. Por sua vez, o sistema de filas de processos prontos, ou *sistema de filas de prontos* é formado por uma ou mais filas de processos prontos que são governadas por uma política de alocação da CPU.

Esse modelo conceitual foi criado para, através das características inerentes à orientação a objetos (encapsulamento, herança e polimorfismo), facilitar o tratamento da diversidade de estratégias que um sistema operacional pode adotar para a gerência da CPU. Utilizando este modelo, o simulador pode considerar que sempre são utilizadas múltiplas filas com transferências entre elas. Um mecanismo de alocação da CPU que utilize somente uma fila FIFO, por exemplo, será considerado um caso particular onde o sistema de filas de prontos é formado por uma única fila.

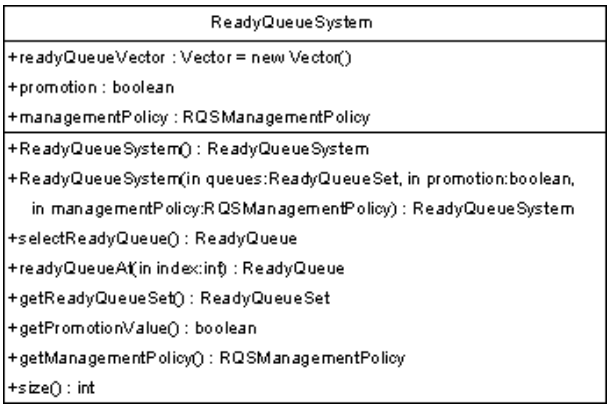

Figura 18 – Classe *ReadyQueueSystem*

3.15. O Sistema de Filas de Processos Prontos

O sistema de filas de prontos é representado pela classe *ReadyQueueSystem* (Figura 18), pertencente ao pacote *cpumngtsim.os.queues*. Esta classe possui um campo de nome *readyQueueVector*, do tipo *java.util.Vector*, onde ficam armazenadas as filas de prontos componentes de um determinado sistema de filas de prontos. As filas de prontos são adicionadas a este *Vector* pelo construtor de *ReadyQueueSystem* que recebe como um dos parâmetros de entrada uma instância da classe *ReadyQueueSet* que nada mais é do que um conjunto de filas de processos prontos.

Cada fila de prontos possui um campo *aging* que indica se a *técnica de envelhecimento de processos* [10] está sendo utilizada por esta fila. Em caso positivo, os campos *agingTime* e *agingDepth,* ambos da classe *ReadyQueue*, indicam, respectivamente, de quanto em quanto tempo será efetuado o envelhecimento dos processos pertencentes a esta fila e de quantas unidades será incrementada a prioridade dos processos no momento do envelhecimento.

A classe *ReadyQueue* também possui um campo *promotion* que indica se ela pode promover processos (transferir processos para uma fila mais prioritária). Em caso positivo, o seu campo *promotionDestiny* indica qual fila de prontos será o destino dos processos promovidos.

Outro campo da classe *ReadyQueue* é o *minPriority*. Ele indica qual a prioridade mínima que um determinado processo precisa possuir para poder entrar nesta fila de processos prontos. Vale lembrar que a prioridade de um processo é indicada no campo *priority* da classe PCB.

Voltando à classe *ReadyQueueSystem,* observamos que ela possui um campo de nome *promotion* que indica se a promoção de processos é permitida neste sistema de filas de prontos. Uma fila de processos prontos irá promover processos somente se as seguintes condições forem satisfeitas:

- O campo *promotion* desta fila de prontos indica que a fila pode promover processos.

- O campo *promotion* do sistema de filas de prontos indica que a promoção de processos está habilitada.
- O grau de prioridade do processo que será transferido deve ser maior ou igual ao valor indicado no campo *minPriority* da fila de prontos de destino, ou seja, deve ser maior ou igual ao campo *minPriority* da fila indicada no campo *promotionDestiny* da *ReadyQueue* onde este processo se encontra originalmente.

Já sabemos como processos menos prioritários podem ganhar prioridade e chegar a serem transferidos para filas de prontos mais prioritárias. Agora vamos ver como processos podem ser transferidos de filas mais prioritárias para filas menos prioritárias.

Em *ReadyQueue,* criamos o campo *timeOut* que indica se processos selecionados para executar a partir de uma determinada fila de prontos terão um tempo limite de execução. Em caso positivo, no campo *processCPUMaxUsingTime,* também de *ReadyQueue,* estará armazenando o tempo máximo de CPU que pode ser utilizado em benefício de um processo selecionado para execução a partir desta fila de prontos.

No campo *clockAtBeginning* da classe PCB armazenamos o valor do relógio do sistema no momento do início da última fase de CPU dos processos. Através do valor deste campo, o sistema operacional verifica se um determinado processo ultrapassou o tempo limite de execução determinado para ele. Se o tempo foi ultrapassado, este processo será encaminhado para uma nova fila de processos prontos, certamente de prioridade mais baixa. Esta nova fila de prontos é indicada no campo *timeOutDestiny* da fila de prontos de origem deste processo (a fila de prontos de origem é a fila onde este processo se encontrava antes de ser escalado para executar. Essa informação é mantida no campo *originQueue* do PCB de cada processo do sistema).

Quando uma classe *ReadyQueue* é instanciada, seu construtor programa o mecanismo de temporização do sistema operacional para que, de tempo em tempos, ocorra um evento de manutenção desta fila de prontos. Este evento é representado através de uma instância da classe *QueueMaintenanceEvent.* A classe *QueueMaintenanceEvent* herda de *TimeEvent* (Seção 3.3) e fornece uma implementação para o método *execute*, responsável por verificar se cada um dos processos pertencentes a esta fila de prontos precisa ser encaminhado para a fila indicada em *timeOutDestiny*. O método também verifica se existem processos em condição de receber mais prioridade (no caso da técnica de envelhecimento estar sendo utilizada) ou serem transferidos para fila indicada em *promotionDestiny.* O intervalo de tempo em que ocorrerão os eventos de manutenção de uma fila de prontos é configurado através do campo *maintenanceTime*, presente na classe *ReadyQueue*.

Precisamos conhecer mais dois campos da classe *ReadyQueue*: o campo *priority* e o campo *entryPoint*. O campo *priority* indica o grau de prioridade desta fila com relação às outras filas de prontos pertencentes ao mesmo sistema de filas de prontos. O campo *entryPoint* indica se a fila de prontos permite a entrada de processos vindo da fila de entrada. Quando um processo entra no sistema, será o quanto antes encaminhado para alguma das filas de processos prontos pertencentes ao sistema de filas de prontos. A fila que irá receber esse processo é escolhida segundo os seguintes critérios:

- O campo *entryPoint* indica que a fila de prontos permite a entrada de processos vindo da fila de entrada.
- O grau de prioridade do processo que será transferido deve ser maior ou igual ao valor indicado no campo *minPriority* da fila de prontos.
- O campo priority da fila possui o maior valor possível.

Quando o escalonador do sistema precisa escolher um novo processo para ser colocado em execução, deve primeiramente decidir de qual das filas de prontos pertencentes ao sistema de filas de prontos será retirado este processo. A política de gerenciamento do sistema de filas de prontos é responsável por tomar esta decisão. O escalonador da CPU invoca o método *selectReadyQueue* da classe *ReadyQueueSystem* e o método, por sua vez, solicita que a política de gerenciamento armazenada no campo *managementPolicy* selecione uma das filas de *readyQueueVector*. O escalonador recebe esta fila como retorno da sua invocação e agora pode solicitar diretamente a ela que seja escolhido o melhor processo segundo a política de alocação que a governa (isto é feito através do método *getProcess* de *ReadyQueue* como foi visto na Seção 3.8).

Outra função da política de gerenciamento de sistemas de filas de prontos é saber como adicionar filas de prontos em *readyQueueVector*. Como foi dito no início desta seção, as filas de processos prontos são adicionadas a este *Vector* pelo construtor de *ReadyQueueSystem* que recebe como um dos parâmetros de entrada um conjunto de filas de processos prontos. Entretanto, o construtor o faz também com intermédio da política de gerenciamento armazenada em *managementPolicy.*

3.16. As Políticas de Sistemas de Filas de Prontos

Já vimos que as principais funções da política de gerência do sistema de filas de prontos são saber como adicionar filas de prontos ao campo *readyQueueVector* de *ReadyQueueSystem* e saber como selecionar uma entre as possíveis várias filas pertencentes à *readyQueue Vector* para que seja retirado desta fila um processo para entrar em execução. As políticas de gerência do sistema de filas de processos prontos são representadas por classes que herdam *de RQSManagementPolicy*, pertencente ao pacote *cpumngtsim.os.queues*, fornecendo uma implementação para seus dois métodos abstratos *addingAlgorithm* e *selectionAlgorithm*.

O método *addingAlgorithm* requer dois parâmetros: uma *ReadyQueue* (a fila que se pretende adicionar

ao sistema de filas de prontos) e um *java.util.Vector* (o *Vector* usado para armazenar as filas de prontos componentes do sistema de filas de prontos). As classes que representam políticas de gerência do sistema de filas de prontos devem fornecer o algoritmo usado para adicionar a fila de prontos ao *Vector*. Duas políticas de gerência são oferecidas por *default* pelo simulador: *UntilEmpty* e *UntilTimeOut.* Essas classes fornecem uma implementação para método *addingAlgorithm* onde as filas de maior prioridade são sempre adicionadas às primeiras posições do *Vector*.

A diferença de comportamento entre as políticas *UntilEmpty* e *UntilTimeOut* se deve à implementação do método abstrato *selectionAlgorithm* de *RQSManagementPolicy*. O método *selectionAlgorithm* é invocado pelo método *selectReadyQueue* do sistema de filas de prontos para que a sua política de gerência de encarregue de selecionar qual a fila de onde será retirado o próximo processo a ganhar posse da CPU.

Na implementação de *selectionAlgorithm* por *UntilEmpty* é fornecido um algoritmo que irá selecionar sempre a fila de maior prioridade dentre as filas disponíveis no *Vector* passado como parâmetro até que esta se torne vazia. Quando esse evento ocorre, então, o algoritmo passa a selecionar a próxima fila de maior prioridade até que esta se torne vazia, e assim por diante. Quando a fila de menor prioridade do sistema se torna vazia, o algoritmo irá continuar selecionando-a. Se, a qualquer momento, uma fila de maior prioridade que estava vazia recebe um novo processo, ela será selecionada. Este algoritmo corresponde ao algoritmo de alocação preemptiva com prioridades fixas [10]. Uma alternativa ao algoritmo de alocação preemptiva com prioridades fixas é repartir o tempo de CPU entre as filas [10]. Essa alternativa é representada pela classe *UntilTimeOut*.

Cada fila de prontos possui um tempo limite de utilização da CPU, indicado no campo *queueCPUMaxUsingTime* da classe *ReadyQueue*. À medida que processos retirados de uma determinada fila de prontos vão consumindo tempo de CPU, incrementamos o valor de um outro campo de *ReadyQueue*, chamado *queueCPUTimeElapsed*. Quando o valor de *queueCPUTimeElapsed* é maior ou igual ao de *queueCPUMaxUsingTime*, dizemos que a fila de prontos ultrapassou seu tempo limite de utilização da CPU.

O algoritmo *UntilTimeOut* fornecido como implementação do método *selectionAlgorithm* de *UntilTimeOut* seleciona sempre a fila de maior prioridade dentre as filas disponíveis no *Vector* passado como parâmetro, até que esta ultrapasse seu tempo máximo de utilização da CPU. Quando esse evento ocorre, então o algoritmo passa a selecionar a próxima fila de maior prioridade até que esta rompa seu tempo máximo de utilização da CPU, e assim por diante. Quando a fila de menor prioridade do sistema rompe seu tempo, o algoritmo zera a contagem de tempo de CPU de todas as filas do sistema e volta a selecionar a fila mais prioritária de todas.

4. Os Usuários Virtuais

Já discutimos como executar e gerenciar processos, conhecemos os programas que dão origem a estes processos, mas ainda não apresentamos a entidade responsável pela geração destes processos. Vamos então apresentar os usuários virtuais do simulador, representados por instâncias de classes que herdam de *VirtualUser*, implementando seus métodos abstratos.

4.1. A Classe *VirtualUser*

A classe *VirtualUser* (Figura 19) possui um campo *name* que define o nome do usuário, um campo *userWorkload* que define o *workload* do usuário e um campo *priority* que define a prioridade do usuário no sistema. O campo *userWorkload* é do tipo *Workload* (Figura 20). A classe *Workload,* por sua vez, possui um campo do tipo *java.util.Vector* onde são armazenados os programas que o usuário virtual poderá submeter à execução. Seus métodos se preocupam com a manipulação dos programas (representados através de instâncias da classe *Program*, detalhada na Seção 3.5) que fazem parte deste *Vector*.

A classe *VirtualUser* possui um método abstrato de nome *chooseProgram* que deve ser implementado fornecendo um algoritmo para selecionar um dentre os, possivelmente diversos, programas pertencentes ao *workload* do usuário. No momento que este usuário resolve submeter algum programa à execução, ele deve chamar o método *chooseProgram* para determinar qual programa será submetido.

A classe *VirtualUser* implementa a interface *java.lang.Runnable* [11], por este motivo, possui um método *run*. Dentro deste método, é definido um *loop* infinito responsável por manter o usuário virtual submetendo seus programas à execução, conforme indicado na Listagem 11. Mas o método *chooseProgram* é abstrato, precisamos fornecer uma implementação para que o método *run* se torne funcional. É isso que fazem as classes *InteractiveUser* e *AutomaticUser*.

```
01 método run {
02     enquanto ( verdade ){
03         submitProgram( chooseProgram );
04     }
05 }
```

Listagem 11 – Método *run* da classe *VirtualUser*

4.2. O Usuário Virtual Interativo

A classe *InteractiveUser* implementa um "usuário virtual interativo". Isso significa que através deste usuário virtual, o usuário real do simulador pode interagir com o sistema, por exemplo, submetendo, a qualquer momento, programas por ele mesmo escolhidos. A classe *InteractiveUser* fornece então uma implementação do método *chooseProgram* onde através do um terminal, o usuário do simulador envia comandos para o usuário virtual. Se o comando enviado corresponde ao nome de algum dos programas pertencentes ao *workload* do usuário virtual interativo, consideramos que este programa

foi escolhido e o retornamos concluindo o método *chooseProgram*.

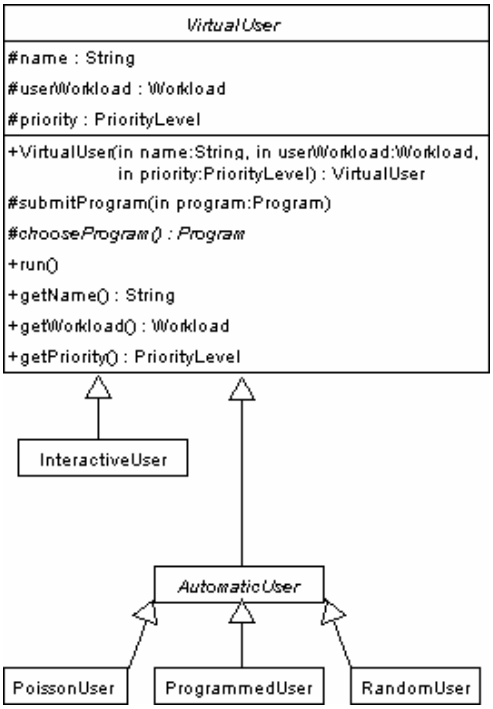

Figura 19 – Classe *VirtualUser* e suas herdeiras

4.3. Os Usuários Virtuais Automáticos

Observando a Figura 19, notamos a existência de outra classe que herda de *VirtualUser*, fornecendo uma implementação para o método *chooseProgram*: é a classe *AutomaticUser* (Figura 21). Na implementação desta classe, o método *chooseProgram* sempre retorna um programa escolhido aleatoriamente dentre os programas pertencentes ao *workload* do usuário virtual.

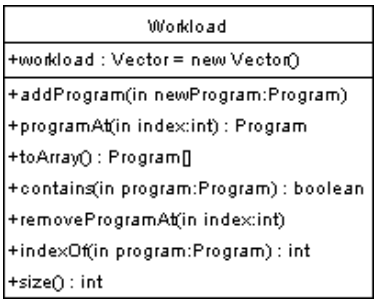

Figura 20 – Classe *Workload*

O método *run* de *AutomaticUser* sobrescreve o de *VirtualUser* conforme indicado na Listagem 12. A diferença entre o método *run* de *AutomaticUser* e o original é que agora inserimos uma linha de código para suspender temporariamente a *thread* do usuário virtual de modo que possamos determinar o intervalo de tempo entre as submissões de programas à execução. O comprimento deste intervalo de tempo é o valor de retorno do método abstrato *nextAwayTime* que é implementado pelas classes *RandomUser*, *ProgrammedUser* e *PoissonUser*, tratadas nas próximas seções.

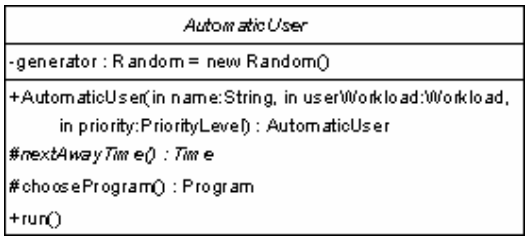

Figura 21 – Classe *AutomaticUser*

4.4. Usuário Automático Aleatório

A classe *RandomUser* (Figura 22) representa o que chamamos de usuário virtual aleatório. Este usuário irá submeter seus programas à execução em intervalos de tempo aleatórios, dentro de uma faixa de valores configurada pelo usuário real do simulador. O limite inferior desta faixa é armazenado no campo *minAwayTime* e o limite superior desta faixa é determinado por *minAwayTime* + *awayTimeRange*.

```
06 método run{
07     enquanto ( verdade ){
08         suspendeExecuçãoTemporariamente(
   nextAwayTime );
09         submitProgram( chooseProgram );
10     }
11 }
```

Listagem 12 – Método *run* da classe *AutomaticUser*

A classe *RandomUser* fornece uma implementação para o método *nextAwayTime* da classe abstrata *AutomaticUser* onde através do gerador de números pseudo aleatórios *generator* é retornado um valor entre *minAwayTime* e *minAwayTime* + *awayTimeRange*.

4.5. Usuário Automático Programado

O usuário automático programado é representado pela classe *ProgrammedUser* (Figura 23). Enquanto no usuário automático aleatório o usuário real do simulador não tinha controle total de qual programa do seu *workload* seria submetido à execução em dado instante e qual o comprimento exato de cada intervalo de tempo entre as submissões destes programas à execução, com o usuário automático programado conseguimos determinar previamente tanto o comprimento de cada intervalo de tempo entre submissões quanto qual programa será submetido ao final de cada intervalo.

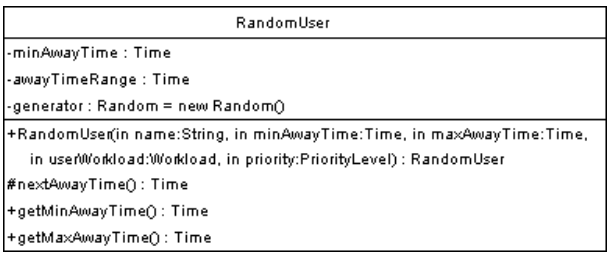

Figura 22 – Classe *RandomUser*

Para viabilizar este controle, criamos a classe *ProgramSubmission* que representa uma submissão de programa à execução. Essa classe possui um campo

chamado *program*, que determina qual programa será submetido, e um campo chamado *awayTime,* que indica após quanto tempo desde o início da simulação ou da última submissão o programa *program* será submetido.

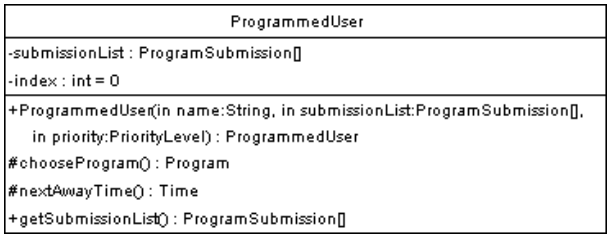

Figura 23 – Classe *ProgrammedUser*

A classe *ProgrammedUser* possui um campo *submissionList* que mantém um *array* de instâncias de *ProgramSubmission*, representando a seqüência de submissões que este usuário irá fazer. Mantemos um campo *index* que aponta, inicialmente, para a primeira posição deste *array*.

Fornecemos uma implementação para o método *nextAwayTime* onde retornamos a cada invocação o valor do campo *awayTime* da instância armazenada na *submissionList*, apontada pelo campo *index*. Sobrescrevemos o método *chooseProgram* fazendo com que ele retorne o valor do campo *program* da instância armazenada na *submissionList*, apontada pelo campo *index*. Em seguida incrementamos o valor de *index*.

4.6. Usuário Automático de *Poisson*

Uma estratégia comumente adotada para expressar matematicamente o intervalo entre as chegadas de novos processos ao sistema operacional é usar a aproximação deste fenômeno a uma distribuição exponencial de tempos de chegada, ou seja, uma distribuição de *Poisson* [09].

O usuário automático de *Poisson* é representado através da classe *PoissonUser* (Figura 24). Seu comportamento é bastante similar ao comportamento do usuário automático aleatório (*RandomUser*), sendo que, em vez dos campos *minAwayTime* e *maxAwayTime,* presentes em *RandomUser,* definimos o campo *meanAwayTime* onde é armazenado o tempo médio de chegada de processos, ou seja, o tempo médio em que este usuário virtual irá submeter seus programas à execução.

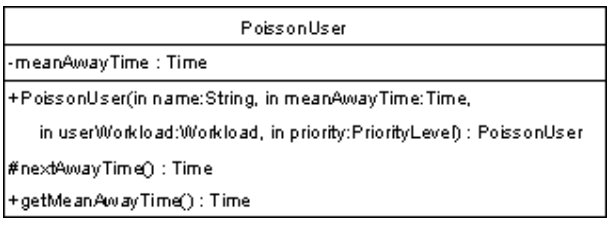

Figura 24 – Classe *PoissonUser*

Fornecemos uma implementação para o método *nextAwayTime* em que, a cada invocação, é retornado um valor de tempo próximo a *meanAwayTime*. Chamando o valor retornado de $t = meanAwayTime + x$, pela definição da distribuição de *Poisson*, sabemos que a probabilidade $P(t)$ de retornarmos um valor t, é $P(t) = 1 - e^{meanAwayTime * t}$.

Assim como em *RandomUser*, o programa que será submetido a cada intervalo de tempo por *PoissonUser* é escolhido aleatoriamente, segundo a implementação do método *chooseProgram* original de *AutomaticUser*.

5. Integração da Interface Gráfica com o Simulador

Gostaríamos de obter um sistema modular onde sua parte funcional pudesse trabalhar independentemente da interface gráfica. No entanto, seria impossível manter uma interface gráfica consistente sem um mínimo de interação com a parte funcional. Para estabelecer esta interação, foi implementada uma "camada de ligação", composta por classes que representam eventos e interfaces que permitem o tratamento destes eventos. No simulador, eventos importantes para manter a interface gráfica atualizada são gerados por objetos no decorrer de uma simulação. Esses eventos são representados pelas classes *ClockEvent*, *SpeedEvent*, *QueueEvent, DeviceEvent e OperatingSystemEvent*.

ClockEvent e *SpeedEvent* são gerados pelo temporizador do sistema e ocorrem, respectivamente, quando o temporizador termina a contagem de uma fatia de tempo e quando o temporizador altera a velocidade da simulação. Para que uma classe possa tratar estes dois eventos, ela deve implementar a interface *ClockListener* e, em seguida, ser registrada no temporizador do sistema através do método *addClockListener (ClockListener listener)* da classe *Clock*.

Quando um processo é adicionado ou removido de uma fila de processos, esta fila gera um *QueueEvent*. Para que uma classe possa tratar este evento, ela deve implementar a interface *ProcessQueueListener* e ser registrada através do método *addProcessQueueListener (ProcessQueueListener listener)* do objeto *ProcessQueue*, o qual representa a fila de processos desejada.

DeviceEvent é gerado por dispositivos de I/O quando estes começam o atendimento a alguma requisição ou quando eles terminam de atender a uma requisição tornando-se disponíveis. Para tratar este evento, uma classe deve implementar a interface *IODeviceListener* e ser registrada no dispositivo de I/O desejado através do método *addIODeviceListener(IODeviceListener listener)* da classe *IODevice*.

Por fim, *OperatingSystemEvent* é gerado pela classe *Kernel* quando o sistema operacional começa a executar, ou quando um novo processo é colocado em execução, fazendo com que o sistema operacional saia de execução. Para que uma classe possa tratar este evento, ela deve implementar a interface *OperatingSystemListener* e, em seguida, ser registrada no sistema operacional através do método *bootSytem* da classe *Kernel*.

As quatro interfaces apresentadas acima herdam de *java.util.EventListener* para manter o padrão *Java* e situam-se também no pacote *cpumngtsim.userinterface.event*. Elas funcionam como uma ponte entre os componentes funcionais do simulador (que geram eventos) e a interface gráfica do simulador (que trata os eventos). Na implementação do simulador, optamos por, em vez de criar quatro classes para implementar cada uma destas interfaces, criar uma só classe que implementa as quatro interfaces de forma integrada. Esta classe é a *SimulatorAdapter*, que responde aos eventos do simulador atualizando componentes gráficos de interação com o usuário.

6. Conclusão

6.1. Modularidade e extensão

Três fatores colaboraram para que o simulador se transformasse em um sistema modularizado. Primeiro, o encapsulamento, característico das linguagens orientadas a objetos, portanto característico da linguagem *Java*, usada na implementação do simulador. Depois, o escopo de nomes fornecido pelo agrupamento de classes em pacotes *Java*. Por fim, a preocupação com a representação da realidade com a maior fidelidade possível, seguida como princípio de projeto. Esses três fatores em conjunto proporcionaram que, por exemplo, nossos dispositivos de I/O trabalhassem independentemente da CPU e do sistema operacional, comunicando-se com eles através apenas de um mecanismo de interrupções. Sendo assim, poderíamos modificar todo o código da CPU e do sistema operacional tendo a certeza de que nossos dispositivos de I/O continuariam a funcionar. No caso da interface gráfica, a facilidade de modificação se deve à modularidade conquistada graças à utilização do conceito de *listener*.

Os mecanismos de classes abstratas e interfaces foram explorados na implementação do simulador. Isto significa que o seu mecanismo básico de funcionamento trabalha com invocações a métodos cujas implementações serão fornecidas por classes que precisam simplesmente seguir um padrão preestabelecido em uma interface ou em uma classe abstrata. Essas classes podem, facilmente, ser elaboradas por programadores que tenham acesso à documentação do projeto.

A portabilidade foi alcançada quando resolvemos adotar a linguagem *Java* como a linguagem de implementação do simulador. Um programa compilado para a máquina virtual *Java* está apto a rodar em qualquer sistema com uma máquina virtual *Java* instalada. Adicionalmente, fornecemos uma versão *applet* do simulador que permite que usuários de qualquer plataforma utilizem o simulador através de um navegador equipado com o *Java Plug-in*.

6.2. Aspectos Funcionais

Conseguimos fornecer ao usuário a possibilidade de configuração desde o *hardware* do sistema computacional até as especificações de funcionamento particulares a cada política de alocação da CPU. Possibilitamos a construção de mecanismos de alocação com uma ou mais filas e transferência entre filas, bem como a parametrização do funcionamento de cada uma das filas individualmente.

Na versão descrita neste trabalho, o usuário do simulador pode acompanhar a simulação em tempo real, observando a evolução dos processos através das filas do sistema e o revezamento entre os processos de usuário e o sistema operacional dentro da CPU. Entretanto, a coleta dos dados sobre produtividade do sistema ainda não é feita, impossibilitando a geração do relatório.

Outra pendência é a interface gráfica para configuração dos programas virtuais submetidos à execução pelos usuários virtuais. Apesar de estes usuários já submeterem programas *default* à execução, não é possível criar novos programas ou alterar os programas existentes.

6.3. Aspectos didáticos

O simulador dá suporte a uma grande variedade de decisões de projeto relacionadas a mecanismos de alocação da CPU. Cobre diferentes técnicas e políticas, todas encontradas na literatura didática especializada em sistemas operacionais. Sendo assim, o simulador de fato se tornou um guia de referência sobre o assunto.

Os painéis de configuração dispõem, de forma estruturada, as diferentes possibilidades de ajuste dos elementos componentes de um módulo de escalonamento de processos de um sistema operacional real. Essa disposição fornece ao usuário uma visão geral sobre o assunto e ajuda na percepção da interdependência entre as diferentes técnicas usadas na implementação desta parte fundamental dos sistemas operacionais. Através da navegação por estes painéis, o usuário é apresentado a um modelo que torna mais concretos aspectos discutidos apenas teoricamente através dos métodos de ensino convencionais.

Um ponto que prejudica o aspecto didático do simulador é a ausência de um sistema de ajuda e *tool-tips* na interface gráfica. Embora essa ausência obrigue o usuário a uma exploração mais profunda da interface, familiarizando-o com as diversas possibilitadas de configuração, a grande quantidade de opções e interdependências entre elas gera dificuldades que podem acabar desestimulando a utilização do *software*.

6.4. Aspectos técnicos

A idéia de recriar, em *software*, com a maior verosimilhança possível, todos os elementos de um sistema computacional real envolvidos com a questão do escalonamento de processos, era bastante promissora. Seríamos poupados de projetar uma arquitetura para nosso sistema virtual, uma vez que nossa arquitetura seria a mesma de um sistema real, bastando copiá-la. Ganharíamos ainda a modularidade herdada do sistema real e estaríamos confiantes em que o simulador iria funcionar, uma vez que o

sistema real tomado como modelo de fato funcionava. Entretanto, alguns problemas surgiram.

Associamos uma *thread* a cada elemento que pudesse potencialmente trabalhar em paralelo no mundo real. Desta forma, todos os dispositivos de I/O possuem *threads* próprias, bem como os usuários virtuais, a CPU e o temporizador do sistema. Considerando-se a configuração *default* do simulador, uma simulação usaria 2 dispositivos de I/O, 6 usuários virtuais, a CPU e o temporizador, somando-se 10 *threads* concorrentes. A esse número ainda poderíamos acrescentar as *threads* de gerência da interface gráfica (mantidas pela própria máquina virtual *Java*) e a *thread* de interrupção de tempo (gerada pelo temporizador do simulador periodicamente).

Uma das conseqüências deste elevado número de *threads* concorrentes é que, como não é possível prever o resultado do escalonamento efetuado pela máquina virtual *Java*, o simulador realiza simulações não determinísticas. Isso quer dizer que o usuário do simulador pode elaborar um determinado conjunto de configurações e obter resultados diferentes para simulações efetuadas com este mesmo conjunto.

Essa característica poderia ser bem razoável quando lembramos que, num sistema real, muitas vezes, também não é possível prever o estado do sistema algum tempo após a submissão de uma carga de trabalho. Entretanto, como assumimos que o simulador deveria funcionar como uma ferramenta didática, o não-determinismo se torna uma característica negativa.

Suponhamos que um professor tenha sugerido, como exercício, a modelagem de um determinado mecanismo de alocação da CPU através do simulador para realizar em seguida uma simulação com uma carga de trabalho preestabelecida. Devido ao não-determinismo, o professor não teria um resultado padrão para usar como referência na correção do exercício, pois cada aluno poderia obter um resultado diferente e, ainda assim, correto. Agora, suponhamos que um aluno tente modelar, no simulador, um mecanismo de alocação sugerido em um livro didático. Mesmo que ele realize corretamente a modelagem, possivelmente não conseguirá realizar simulações que coincidam com os resultados teóricos descritos pelo livro. Estes são apenas exemplos das várias situações em que um simulador determinístico seria bastante desejável do ponto de vista didático.

Outro problema com a utilização de inúmeras *threads* foram as imprecisões inseridas nas simulações. Elas são provocadas por dois fatores: o controle via *threads* de eventos disparados graças a uma condição temporal e impossibilidade de atualização do tempo de suspensão das *threads* após uma alteração da velocidade da simulação.

No PCB dos processos virtuais, armazenamos várias informações de tempo, todas calculadas a partir da leitura de uma variável interna do temporizador do sistema que controla a passagem do tempo. Essas informações são importantes para a tomada de uma série de decisões no escalonamento de processos. Como a contagem de tempo é realizada por uma *thread*, a corretude destas informações depende de que esta *thread* seja escalada para execução antes que a leitura da variável desatualizada seja realizada.

Devido também à imprevisão no escalonamento das *threads*, podemos ter eventos ocorrendo fora de ordem. Suponhamos que dois usuários virtuais estejam prestes a submeter programas à execução. O primeiro estava configurado para submeter no instante x, o outro no instante x+1. Uma *thread* qualquer termina sua execução no instante x+2, quando então a máquina virtual *Java* tem que decidir qual a próxima *thread* a ser escalada. Por algum motivo de escalonamento da máquina virtual, o usuário programado para submeter seu programa em x+1 poderia ser escolhido para executar, passando, erroneamente, a vez do primeiro. Problemas semelhantes poderiam ocorrer também envolvendo dispositivos de I/O virtuais ou qualquer outro elemento controlado por uma *thread*.

O outro fator gerador de imprecisão, como foi dito, é a impossibilidade de atualização[4] do tempo de suspensão das *threads* após uma alteração da velocidade da simulação. As *threads* dos dispositivos de I/O e as *threads* dos usuários virtuais estão, na maior parte do tempo suspensas representando, respectivamente, o trabalho dos dispositivos e a espera para a submissão de um novo programa. Quando o temporizador do sistema ou o usuário real do simulador decide alterar a velocidade da simulação, o tempo de suspensão das *threads* não é atualizado. Sendo assim, poderíamos ter, num mesmo instante, *threads* trabalhando cientes da nova velocidade de simulação e *threads* ainda suspensas com parâmetros referentes à velocidade anterior.

Ainda relacionado com a utilização das *threads,* outro problema foi a implementação de uma classe que faz chamadas a métodos *deprecated* da interface *Java*. Necessitávamos de uma forma de finalizar e pausar uma simulação. Logo, necessitávamos de uma forma de finalizar e suspender temporariamente o trabalho normal das *threads* componentes da simulação. Para evitar a utilização dos métodos *deprecated* poderíamos trocar os *loops* infinitos que mantêm as *threads* em execução por *loops* condicionados a um teste que verifica se a simulação ainda deve continuar. Adicionalmente, deveríamos inse-

[4] Dois problemas existem para a atualização do tempo de suspensão. Primeiramente, não temos como saber quanto tempo a *thread* já permaneceu suspensa para calcularmos o tempo que ela ainda necessitaria ficar suspensa dada a nova velocidade. Depois, ainda que fosse possível este cálculo, necessitaríamos interromper as *threads* suspensas através de uma chamada a seus métodos *interrupt*, e dentro de blocos de código *catch(interruptedException ie){}*, novamente suspender as *threads* pelo novo tempo calculado. Dentro deste bloco, poderia ocorrer nova alteração na velocidade, e retornaríamos aos mesmos problemas iniciais.

rir pontos de teste para verificar se a simulação foi pausada. Em caso positivo, utilizaríamos semáforos que bloqueariam as *threads* até que a simulação fosse prosseguida.

Essa estratégia, além de adicionar *overhead*, faria com que o efeito de pausar e finalizar a simulação demorasse a ser percebido pelo usuário, pois as *threads* deveriam alcançar os pontos de teste para serem finalizadas ou bloqueadas. Como muitas destas *threads* normalmente permanecem suspensas por períodos longos de tempo, o retardo seria ainda mais considerável. Ainda deveríamos considerar os problemas de *deadlock* que ocorreriam dependendo da forma em que as *threads* fossem finalizadas ou bloqueadas. Por estes motivos, acabamos utilizando os métodos *suspend*, *stop* e *resume* da interface *Java*. Essa opção, provavelmente, foi a responsável pelos problemas percebidos algumas vezes em que pausamos ou finalizamos uma simulação nesta versão do simulador.

Por fim, mais uma opção que pode ter contribuído para imprecisões, agora não relacionada à questão da utilização de *threads*. Possibilitamos ao usuário do simulador determinar com precisão de até nano segundos o comprimento de um ciclo de *clock* da CPU e o tempo de leitura e escrita de um bloco de dados pelos dispositivos de I/O. Em seguida, mapeamos 1 nano segundo "virtual" com 1 nano segundo real, confiando na precisão do método fornecido pela interface *Java* com esta finalidade.

Entretanto, sabemos que esta precisão depende dos mecanismos de temporização adotados pela plataforma que executa a máquina virtual *Java* e, ainda que ele fornecesse, precisão razoável de nano segundos, haveria diferenças entre as plataformas, alterando o comportamento do simulador. Em paralelo a esta questão, somente o *overhead* de se programar o temporizador e efetuar alguns cálculos preliminares já faria com que determinada ação que deveria demorar poucos nano segundos demorasse muito mais do que isso.

Poderíamos pensar em resolver o problema através da multiplicação por um fator de correção todas as medições de tempo. Essa técnica, porém, faria com que valores grandes de tempo (como o intervalo em que usuários ficam aguardando para submeter à execução um novo programa) sofressem uma variação em absoluto muito grande, próxima à variação que sofreriam os valores em nano segundos. Seria difícil arcar com essa disparidade, pois o usuário real do simulador acabaria sendo obrigado a esperar muito tempo para ver os programas entrando em execução, enquanto os valores em nano segundos, por sua vez, ainda não estariam satisfatoriamente dilatados.

Poderíamos pensar em usar fatores de correção diferentes para ordens de grandezas de tempo diferentes. Entretanto, não teríamos parâmetros confiáveis para determinar quando obtivemos fatores razoáveis, ainda mais quando levamos em consideração as diferenças entre plataformas. Seria, então, praticamente impossível calcular estes fatores satisfatoriamente, o que, provavelmente, pioraria as imprecisões.

6.5. Novas versões

Durante os testes realizados na primeira versão do simulador verificou-se que o uso de *multi-threading* causava em grande parte os problemas discutidos na seção anterior. Uma nova versão do simulador foi, então, desenvolvida com um código completamente seqüencial. A nova abordagem diminuiu de 98 para 51 o número de classes *Java* utilizadas na implementação, sendo um indicativo da grande simplificação obtida. Com a arquitetura mais simples, o simulador tornou-se mais eficiente, o que se refletiu na velocidade de processamento das simulações.

Fora a questão da eficiência, na versão 3.0 os problemas detectados no simulador foram, em grande parte, resolvidos, restando no entanto a adição de ajuda e *tool-tips* na interface para complementar a ferramenta. Adicionalmente, a nova versão cria um diagrama de *Gantt* e mantém a interface gráfica atualizada com dados estatísticos sobre a produtividade do sistema enquanto decorre uma simulação, o que não era previsto no projeto inicial do simulador.

Problemas de interface e usabilidade da ferramenta estão sendo examinados e uma série de recomendações estão sendo produzidas no contexto de um outro projeto de graduação no DICC/IME/UERJ [17].

6.6. Simulador de Gerência de memória

Está sendo implementada uma extensão do simulador de mecanismos de alocação da CPU que irá permitir a avaliação e comparação da produtividade de um sistema computacional de acordo com o emprego de diferentes mecanismos de gerência da memória. Este trabalho será desenvolvido como projeto final de graduação no DICC/IME/UERJ, e já apresenta alguns protótipos funcionais, demonstrando que a facilidade de extensão e reuso, almejadas pelo projeto original, de fato foram alcançadas.

7. Referências

[01] BALL, S.; BARR, M. *"Introduction to Counter/Timer Hardware",* Embedded Systems Programming (http://www.embedded.com/), setembro, 2002.

[02] BAR, M., *"Keeping the Time",* Byte.com (http://www.byte.com), semana de 7 de fevereiro, 2000.

[03] COSTELLO, A.; VARGUESE, G., "*Redesigning the BSD Callout and Timer Facilities*", Washington University, http://www.nicemice.net/amc/ ou http://www-cse.ucsd.edu/users/varghese/, novembro, 1995.

[04] DEITEL, H., "*An Introduction to Operating Systems*", revisão da primeira edição, Addison-Wesley Publishing Company, julho, 1984.

[05] Intel Corporation, "*8259A Programmable Interrupt Controller*", dezembro, 1988.

[06] KINOSHITA, J.; CUGNASCA, C.; HIRAKAWA, A., "*Experiência 5: Interrupções*", Escola Politécnica da USP/Departamento de Engenharia de Computação e Sistemas Digitais, http://www.pcs.usp.br/~jkinoshi/, 2001.

[07] KOZIEROK, C., *"Interrupt Controllers",* The PC Guide (http://www.pcguide.com), abril, 2001.

[08] Microsoft Corporation, "*The Importance of Implementing APIC-Based Interrupt Subsystems on Uniprocessor PCs*", Windows Platform Development Site (http://www.microsoft.com/hwdev/), janeiro, 2003.

[09] SHAY, W., "*Sistemas Operacionais*", Makron Books, 1996.

[10] SILBERSCHATZ, A.; GALVIN, P., "*Sistemas Operacionais: conceitos*", 5ª edição, Prentice Hall, 2000.

[11] Sun Microsystems, *"Java 2 Platform, Standard Edition, v1.2.2 API Specification",* pacote *java.lang,* http://java.sun.com/products/jdk/1.2/docs/api/java/lang/package-summary.html, 1998.

[12] Sun Microsystems, "*Java 2 SDK Standard Edition Documentation Version 1.2.1"*, http://java.sun.com/products/jdk/1.2/docs/index.html, 1998.

[13] Sun Microsystems, "*The Java Tutorial", trilha Essential Java Classes, Threads: Doing Two or More Tasks At Once,* Seção *Understanding Thread Priority*, http://java.sun.com/docs/books/tutorial/essential/threads/priority.html, 2003.

[14] TANENBAUM, A.S., "*Modern Operating Systems*", Prentice-Hall, 1992.

[15] Tics Realtime Articles - *"Different Timing Mechanisms and How They are Used",* http://www.cris.com/~Tics/tutorials.html.

[16] VARGUESE, G.; LAUCK, T., *"Hashed and Hierarchical Timing Wheels: Efficient Data Structs for Implementing a Timer Facility"*, http://www-cse.ucsd.edu/users/varghese/, fevereiro, 1996.

[17] FERREIRA, P. P, "Avaliação Semiótica de um Simulador de Escalonamento de CPU", Projeto Final, DICC/IME/UERJ, Março, 2005.