Um modelo de objetos para simulação de mecanismos de alocação da CPU

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

Alexandre Sztajnberg
DICC/IME e PEL/FEN - UERJ
alexszt@ime.uerj.br

Resumo. *O escalonamento de processos é um assunto que requer, para o seu entendimento, certa capacidade de abstração. Seria desejável contar com uma ferramenta de apoio para atividades práticas, que pudesse facilitar a compreensão do vários aspectos deste tema. Desenvolvemos um modelo de objetos, na forma de um framework de classes extensível e reutilizável, que modela os principais elementos de um sistema computacional. A partir do framework, desenvolvemos um simulador que fornece aos usuários meios de compor e configurar mecanismos de alocação da CPU e, em seguida, testá-los sob diferentes cenários de funcionamento.*

Abstract. *Understanding processes scheduling mechanisms depends on a certain abstraction capability. It would be interesting to have a tool supporting practical activities, which could facilitate the understanding of the various aspects regarding this area. We developed an object model, resulting in an extensible and reusable framework, which models the main components of a computational system. Using this framework, we created a simulator that allows the users configuring and customizing CPU allocation mechanisms, and to evaluate them under different work scenarios.*

1. Introdução

O escalonamento de processos é um assunto que requer capacidade de abstração. Seria desejável uma ferramenta que auxiliasse a tornar os mecanismos de alocação da CPU em algo mais concreto. Com este objetivo propusemos um modelo de classes onde recriamos, em *software*, os componentes principais de um sistema computacional real. Com base neste modelo desenvolvemos componentes simulando alguns módulos do sistema operacional. Em linhas gerais, simulamos a CPU, os dispositivos de I/O e os diversos elementos inerentes ao sistema operacional como, por exemplo, as filas de processos e as rotinas de tratamento de interrupção. Desenvolvemos, sobre esta base, os módulos de escalonamento de processos e mecanismos de alocação da CPU.

O simulador foi implementado utilizando-se conceitos de encapsulamento, herança e polimorfismo inerentes à tecnologia de objetos. Através desses conceitos, o micronúcleo do sistema operacional, assim como todas as outras classes responsáveis pela parte funcional foram mantidas simples, abstraindo-se das particularidades de cada política de alocação da CPU. Os algoritmos para seleção de uma fila de prontos dentre as múltiplas filas disponíveis e as decisões referentes aos parâmetros de funcionamento são selecionados pelo usuário do simulador. Esses detalhes foram tratados por classes responsáveis por representar cada uma das opções de configuração possível. O simulador descrito neste trabalho foi desenvolvido em Java, 1.2. Utilizamos também o pacote *Swing*, versão 1.0.2, para desenvolver a interface gráfica (Figura 1).

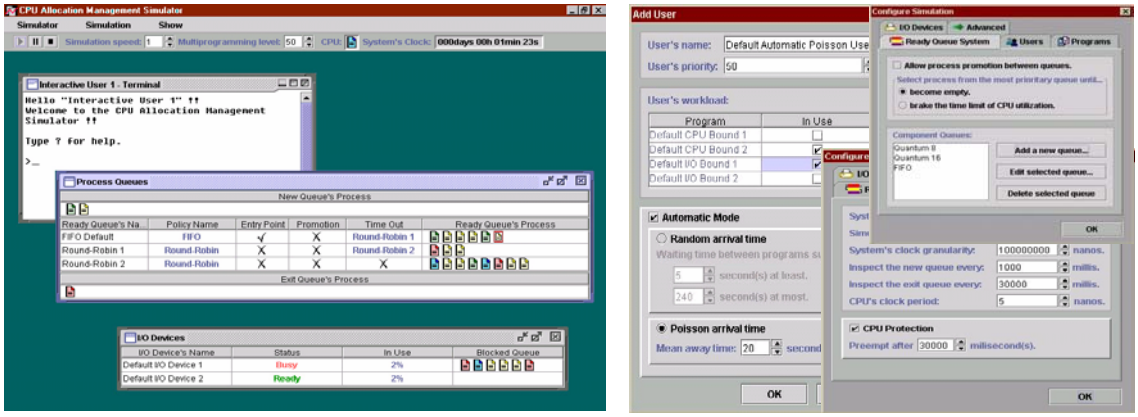

Figura 1. Interface gráfica: (a) *canvas* de exibição, (b) painéis de configuração

O restante do texto está organizado da seguinte forma. Na Seção 2 são apresentados o modelo e a simulação do hardware; na Seção 3, o sistema operacional e nas Seções 4 e 5 as políticas e mecanismos de alocação de CPU. A Seção 6 detalha os usuários virtuais; a Seção 7 trata da camada de ligação de interface gráfica e na Seção 8 listamos trabalhos correlatos. Considerações finais são apresentadas na Seção 9.

2. O Modelo de Objetos

Os componentes que potencialmente trabalham em paralelo em um sistema computacional real como a CPU e os dispositivos de I/O são mapeados em *threads* de execução dedicadas. Foi criado um 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.

As políticas de alocação da CPU foram desenvolvidas segundo uma estratégia baseada em objetos. São disponibilizadas quatro políticas de alocação de CPU diferentes. Uma classe geral foi definida, descrevendo o comportamento de uma política genérica de alocação da CPU. Quatro classes especializam, através de herança, esta classe geral, de forma a caracterizar o comportamento de cada uma das quatro políticas de alocação. O componente de sistema operacional faz chamadas aos métodos da classe geral, e as particularidades de cada política são resolvidas por herança e polimorfismo.

A mesma estratégia foi adotada em outros elementos. Por exemplo: o conjunto de instruções suportado pela CPU é herdado de uma única classe, assim como os tratadores de interrupção e os serviços fornecidos pelo sistema operacional aos processos. Dessa forma torna-se mais fácil a extensão das funcionalidades do simulador, tais como a introdução de novas políticas de alocação da CPU, novas instruções ou novos serviços do sistema operacional. A Figura 2 apresenta o modelo geral do simulador, contendo suas classes principais: o núcleo do sistema operacional, filas de processo, políticas de alocação da CPU, usuários e programas de carga.

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 (Listagem 1).

```
1    enquanto (verdade){
2       novaInstrução = buscaNovaInstrução;
3       executaInstrução( novaInstrução );
4       se ( interruptionFlag ≠ nulo ) então { // Testa interrupção ou exceção
5           executaTratador( interruptionFlag );
6           // Reabilitando as interrupções:
7           enableInterruptions;    } }
```

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

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 que são submetidos ao simulador pelos usuários virtuais. 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 sejam ligadas umas às outras, formando uma lista ligada de instruções.

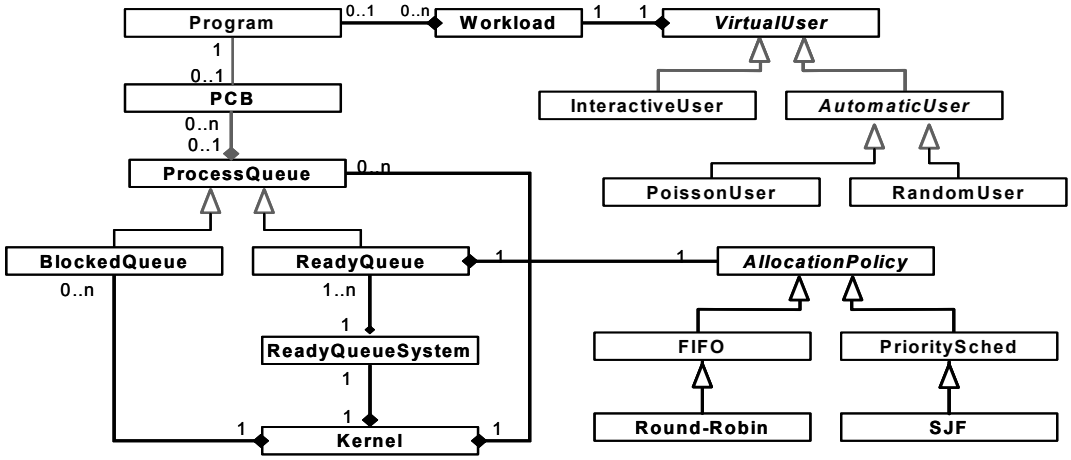

Figura 2. Modelo de objetos do simulador

O simulador também fornece uma instrução especial para a realização de chamadas ao sistema, que tem como parâmetro o 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.

Mecanismo de Interrupção. A opção adotada pelo simulador para implementar o sistema de interrupções 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. A idéia é permitir que o usuário do simulador utilize um número ilimitado de dispositivos periféricos para cada simulação. Utilizamos um método de arbitragem de barramento através de um controlador de pedidos de interrupção [07][08][05]. O controlador de pedidos de interrupções é implementado dentro da própria classe da CPU, através de um mecanismo de semáforos. No simulador, os tratadores de interrupção são implementados através de classes que herdam da classe abstrata *InterruptionHandler* (Figura 3). Estes tratadores fornecem, através do método *execute,* o código que será executado quando ocorrer uma interrupção associada a este tratador.

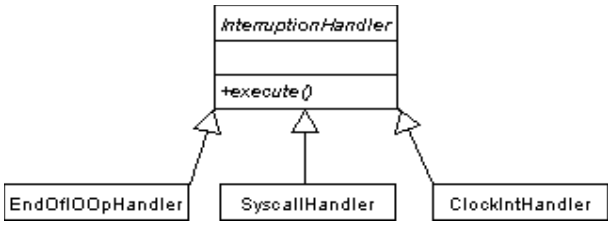

Figura 3. Classe *InterruptionHandler* e suas herdeiras

Temporizador. O temporizador implementado gera interrupções em um intervalo de tempo configurável pelo usuário. 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; (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. A Listagem 2 representação o *loop* principal do temporizador em pseudocódigo.

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

Listagem 2. *Loop* principal do temporizador

Dispositivos de I/O. 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. As operações que o dispositivo de I/O podem executar são representadas por classes que herdam de uma classe abstrata chamada *IOOperation*. Os processos que executam na CPU do simulador podem, através de chamadas ao sistema (instruções *Syscall*), solicitar um serviço de requisição de operação de I/O e serão então bloqueados enquanto a operação se processa.

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

Listagem 3. *Loop* principal em um dispositivo de I/O

Velocidade da Simulação. No simulador utilizamos o método "*sleep*", fornecido pela classe *Thread* para representar o tempo que um componente deve ficar paralisado. Este método requer um parâmetro *p* que indica por quanto tempo se dará esta paralisação. A Listagem 3 esquematiza o *loop* principal de um dispositivo de I/O do sistema. 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. Usando este padrão, permitimos ao usuário regular 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.

3. O Sistema Operacional Virtual

Inspiramo-nos no conceito de micronúcleo para criar um sistema operacional modular, permitindo que os diversos mecanismos de alocação da CPU disponíveis pudessem ser intercambiáveis. O sistema operacional do simulador é formado pelo conjunto de classes pertencentes a um pacote chamado *cpumngtsim.os*. Dentre estas classes, encontramos a classe *Kernel*, que representa o micronúcleo do simulador.

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

Listagem 4. O método *access*

O mecanismo de interrupções implementado pelo simulador utiliza um endereço padrão de desvio para tratamento de interrupções. Este ponto de entrada padrão é representado pelo método *access* da classe *Kernel* (Listagem 4). 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. O método *access* espera como argumento uma classe *InterruptionHandle*, pai da hierarquia de tratadores. Através de polimorfismo, o método *access* será capaz de proporcionar um tratamento diferenciado para cada tipo de interrupção que ocorre no sistema.

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 um objeto que implemente a interface *SystemService* (Figura 4).

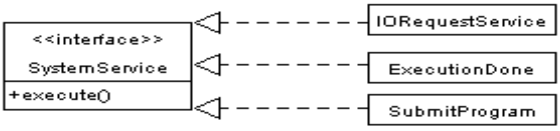

Figura 4. Interface *System Service* e suas implementações

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.

Tratador de Interrupções de Tempo. A implementação do tratador de interrupções de tempo baseada em lista ordenada. A classe *ClockIntHandler* que fornece métodos para registrar e excluir *timers*. 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.

Tratador de Interrupções de I/O. Os tratadores de interrupções acionados no 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 (Listagem 5).

```
01    CPU.testAndSetIntFlag ( new EndOfIOOpHandler( esseDispositivo ) );
```

Listagem 5. Fim de operação de I/O

Uma vez que este tratador entre em execução, ele libera o dispositivo para o atendimento de uma outra requisição, em seguida atualiza a tabela de *status* de dispositivos e o PCB do processo que havia sido bloqueado aguardando o final da operação de I/O. Este processo então é re-encaminhado para sua fila de prontos de origem. Por fim, o tratador varre a tabela de *status* de dispositivo em busca de alguma outra requisição de I/O que esteja pendente.

Programas Virtuais. Os programas reconhecidos pelo sistema operacional virtual do simulador são representados por instâncias da classe *Program*, 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.

Filas de Processos. Dentro do pacote *cpumngtsim.os*, criamos o pacote *queues*, responsável por armazenar as classes que representam as filas de processos, *ProcessQueue*, utilizadas pelo simulador e as classes relacionadas a estas filas. As filas de processos do simulador são representadas por instâncias de *ProcessQueue* ou de classes que herdam de *ProcessQueue*. A classe *ProcessQueue* armazena os PCBs dos processos a ela alocados no campo *pcbVector*, do tipo *java.util.Vector*. Fornece métodos convenientes para manipular esta fila de processos.

Filas de Dispositivo. No simulador, as filas de dispositivo são representadas por instâncias da classe *BlockedQueue*, que herda de *ProcessQueue*. A classe *BlockedQueue* fornece um método para remover um processo da fila a partir do seu PID. 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. 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.

Filas de Processos Prontos. Outra classe pertencente ao pacote *queues* é a *ReadyQueue*, que também estende a funcionalidade de *ProcessQueue,* adicionando uma série de campos e sobrescrevendo os seus métodos *addProcess* e *getProcess*. No simulador, consideramos 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. O campo *policy* que define a política de alocação da CPU que será usada para governar esta fila.

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. Na Listagem 6, notamos que, quando o método *addProcess(pcb)* é invocado, utilizamos o

campo *policy* da fila de prontos para adicionar *pcb* à *pcbVector*. Se *policy*, por exemplo, contém uma política FIFO, *pcb* será adicionado na última posição de *pcbVector*.

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

Listagem 6. Método *addProcess* da classe *ReadyQueue*

Notamos analogamente, que 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, será removido o PCB da primeira posição de *pcbVector*. Se *policy* contém uma política de alocação por prioridade, será retirado o PCB do processo mais prioritário. O método *getProcess* é chamado pelo escalonador quando a CPU está ociosa e é necessário escolher um novo processo para entrar em execução.

4. As Políticas de Alocação da CPU

Para criar uma nova política de alocação da CPU, devemos criar uma classe que herde de *AllocationPolicy*, do pacote *cpumngtsim.os.queues*, fornecendo uma implementação para os seus métodos abstratos: (i) *addingAlgorithm* deve fornecer um algoritmo conveniente para adicionar o PCB de acordo com a política representada pela classe que está implementando o método; (ii) *obtainingAlgorithm* devem fornecer um algoritmo conveniente para escolher, dentre os PCBs pertencentes ao *Vector*, um deles para ser removido da fila. Essa escolha é feita de acordo com a política representada pela classe que implementa este método; (iii) *dispatchAlgorithm* 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 (por exemplo, preparar o temporizador do sistema para avisar quando o *quantum* de tempo tiver expirado); e (iv) *preemptionAlgorithm* deve ser sobrescrito pelas políticas que podem possuir um comportamento preemptivo.

Política FIFO. A política FIFO (*first-in-first-out*) é representada pela classe *FIFO*, que sobrescreve os métodos *addingAlgorithm* e *obtainingAlgorithm* de modo a implementar em *pcbVector* (passados como parâmetro) uma fila; sempre que é solicitado um PCB através de *obtainingAlgorithm,* retornamos o primeiro PCB pertencente à *pcbVector*. Quando é adicionado um PCB*,* o fazemos no final do *Vector*.

Política Round-Robin. A política *round-robin* é representada pela classe *RoundRobin*. 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. O método *dispatchAlgorithm* de *AllocationPolicy* é sobrescrito conforme indicado na Listagem 7. Uma nova instância de *QuantumExpiryEvent* é atribuída ao campo *expiryEvent* (linha 03). No construtor de *QuantumExpiryEvent* indicamos que o processo representado pelo parâmetro *pcb* será interrompido ao término do *quantum* de tempo indicado pelo parâmetro *quantum*. Em seguida, registramos *expiryEvent* em *ClockIntHandler*. Com isso, o temporizador do sistema será programado para que ao final do *quantum* de tempo determinado, seja executado o método *execute* de *expiryEvent*.

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

Listagem 7. Método *dispatchAlgorithm* em *RoundRobin*

Escalonamento por Prioridade. O campo priority da classe PCB é usado pela política de alocação da CPU representada pela classe *PrioritySched*. A classe *PrioritySched* 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. 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*.

Política SJF. A classe SJF (*shortest-job-first* [14]) também 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. Vale observar que, como SJF herda de *PrioritySched*, possui o campo *preemptive* que define se o modo não-preemptivo ou preemptivo de governo está sendo usado. O método *addingAlgorithm* de *AllocationPolicy* 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.

5. Mecanismo de Alocação da CPU

No simulador, consideramos que o sistema operacional utiliza de um *mecanismo de alocação da CPU* para gerenciar a mesma. Este mecanismo 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 é formado por uma ou mais filas de processos prontos que são governadas por uma política de alocação da CPU. Utilizando este modelo, o simulador admite que sempre são utilizadas múltiplas filas com transferências entre elas, permitindo, assim, a configuração de políticas de escalonamento através de composição. Um mecanismo de alocação FIFO, por exemplo, será considerado um caso particular.

Sistema de Filas de Processos Prontos. O sistema de filas de prontos é representado pela classe *ReadyQueueSystem*, 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 sistema de filas de 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 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. Em caso positivo, o seu campo *promotionDestiny* indica qual fila de prontos será o destino dos processos promovidos.

Políticas de Sistemas de Filas de Prontos. As políticas de gerência do sistema de filas de 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*. O primeiro algoritmo corresponde ao algoritmo de alocação preemptiva com prioridades fixas [10]. O segundo é uma alternativa ao algoritmo de alocação preemptiva com prioridades fixas é repartir o tempo de CPU entre as filas [10].

6. Os Usuários Virtuais

A entidade responsável pela geração de processos de carga ao simulador são os usuários virtuais, representados por instâncias de classes que herdam de *VirtualUser*. A classe *VirtualUser* (Figura 5) 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*. A classe *Workload,* por sua vez, possui um campo do tipo *java.util.Vector* onde são armazenados os programas de usuário. Seus métodos se preocupam com a manipulação dos objetos da classe *Program* que fazem parte deste *Vector*.

A classe *VirtualUser* implementa a interface *java.lang.Runnable.* Dentro do método *run* é definido um *loop* infinito responsável por manter o usuário virtual submetendo seus programas à execução (Listagem 8). Mas o método *chooseProgram* é abstrato e precisamos fornecer uma implementação para que o método *run* se torne funcional. Esse método deve ser implementado fornecendo um algoritmo para selecionar um dentre os diversos programas pertencentes ao *workload* do usuário. No momento um usuário resolve submeter algum programa à execução, ele irá usar este método para determinar qual programa será submetido.

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

Listagem 8. Método *run* da classe *VirtualUser*

Usuários Virtuais Automáticos. Na implementação desta classe, o método *chooseProgram* sempre retorna um programa escolhido aleatoriamente dentre os programas do *workload* do usuário virtual. No método *run* de *AutomaticUser* inserimos código para suspender temporariamente a *thread* do usuário virtual, criando assim, intervalos de tempo entre as submissões de programas à execução. Este intervalo de tempo é o valor de retorno do método abstrato *nextAwayTime* que é implementado pelas classes *RandomUser*, *ProgrammedUser* e *PoissonUser*, tratadas adiante.

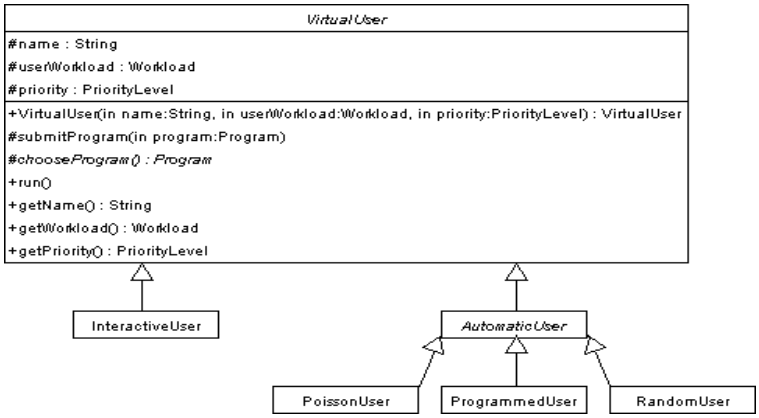

Figura 5. Classe *VirtualUser* e herdeiras

Usuário Automático Aleatório. A classe *RandomUser* 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*. 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*.

Usuário Automático Programado. O usuário automático programado é representado pela classe *ProgrammedUser*. Com o usuário automático aleatório, o usuário real do simulador não tem controle sobre qual programa do seu *workload* será submetido à execução e nem sobre qual o comprimento de cada intervalo de tempo entre as submissões destes programas à execução. Com o usuário automático programado conseguimos determinar previamente tanto cada intervalo de tempo entre submissões, quanto qual programa será submetido ao final de cada intervalo.

Usuário Automático de *Poisson.* O usuário automático de *Poisson* é representado através da classe *PoissonUser*. Seu comportamento é similar ao do usuário automático aleatório. Entretanto, 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. Fornecemos uma implementação para o método *nextAwayTime* em que, a cada invocação, é retornado um valor de tempo próximo a *meanAwayTime*, obedecendo a uma distribuição de *Poisson*.

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 uma implementação do método *chooseProgram* onde através do um terminal, o usuário envia comandos. 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*.

7. A Camada de Ligação com a Interface Gráfica

Para estabelecer a interação entre o framework de simulação e a interface gráfica

(Figura 1) foi implementada uma "camada de ligação", composta por classes que representam eventos e interfaces que permitem o tratamento destes eventos (Figura 6).

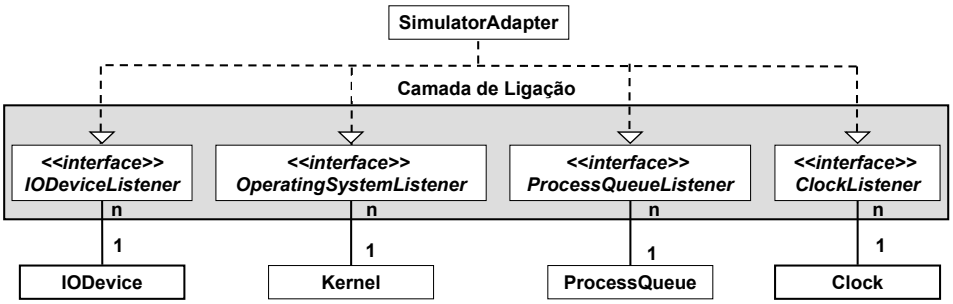

Figura 6. Camada de ligação

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*. Elas funcionam como uma ponte entre os componentes funcionais do simulador (que geram eventos) e a interface gráfica do simulador (que trata os eventos). Desta forma, facilita-se também o mapeamento na interface gráfica das extensões introduzidas no framework de simulação.

8. Trabalhos Correlatos

Um bom número de simuladores relacionados ao escalonamento de CPU pode ser encontrado em buscas na Internet. [17 e 18], por exemplo, simulam detalhes relacionados com processadores reais e permitem a especificação de dependências entre a arquitetura de hardware e módulos de sistema operacional. Outros, tais como [19, 20, 21, 22] têm objetivo didático similar ao nosso, mas não apresentam uma interface amigável ou não apresentam a simulação de forma visual. [23], possui variadas opções de simulação e uma interface gráfica elaborada, e oferece a oportunidade de extensões nos módulos de simulação. A nossa ferramenta, por sua vez, provê um *framework* com esta finalidade e permite a extensão da parte gráfica com a camada de ligação.

9. Conclusão

O uso de classes abstratas e interfaces foram exploradas 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. Nesta linha, está sendo implementada uma extensão para incluir mecanismos de gerência da memória. Basicamente, o núcleo do framework é reusado integralmente e extensões são "plugadas" nas classes que simulam o hardware, nos programas e na interface gráfica.

O simulador foi disponibilizado em 2003 na disciplina de Sistemas Operacionais do DICC/UERJ. Além do seu uso como ferramenta de apoio didático foi feita uma avaliação de usabilidade e semiótica [24]. Nesta avaliação confirmamos a necessidade de se melhorar a ferramenta com *tool-tips*, ajuda o*n-line*. Também se identificou que alguns painéis não obedeciam a um padrão ou poderiam ser aprimorados segundo

critérios de semiótica. Adicionalmente, na avaliação observou-se que os alunos demandavam que a ferramenta pudesse produzir como resultado um relatório no mesmo estilo apresentado nos livros (e cobrado pelo professor nas provas). Assim, adicionou-se à interface existente, através da camada de ligação, um diagrama de *Gantt* e alguns elementos com dados estatísticos sobre a produtividade do sistema enquanto decorre uma simulação, o que não era previsto no projeto inicial do simulador. Melhorias estão sendo introduzidas na ferramenta no contexto de projetos de graduação na UERJ. O simulador esta disponível em *http://www.ime.uerj.br/~alexszt/cpumngtsim/doc/* .

Agradecimentos. Agradecemos o apoio do PIBIC/UERJ. Alexandre Sztajnberg agradece o apoio da RNP/CPqD (Projeto GIGA) a Faperj (E-26/171.130/2005).

Referências

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

[02] Bar, M., *"Keeping the Time",* Byte.com (http://www.byte.com), Fev, 2000.

[03] Costello, A.; Varguese, G., "*Redesigning the BSD Callout and Timer Facilities*", Washington University, http://www.nicemice.net/amc/ , Nov, 1995.

[04] Deitel, H., "*An Introduction to Operating Systems*", Addison-Wesley Publishing Company, 1984.

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

[06] Kinoshita, J.; Cugnasca, C.; Hirakawa, A., "*Experiência 5: Interrupções*", PCS/Escola Politécnica da USP/, http://www.pcs.usp.br/~jkinoshi/, 2001.

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

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

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

[10] Silberschatz, A.; Galvin, P., "*Sistemas Operacionais: conceitos*", 5ª Ed, 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", 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/, Fev, 1996.

[17] Lepreau, J., Flatt. M., et al, "Utah The OSKit Project", http://www.cs.utah.edu/flux/oskit/, 2002.

[18] Singhoff, F. e LISYC Team, "The Cheddar project: a free real time scheduling analyzer", EA LISYC Team, http://beru.univ-brest.fr/~singhoff/cheddar/, 2005.

[19] Neugebauer, R., "Scheduling Simulator", http://www.dcs.gla.ac.uk/~neugebar/schedsim.html, 2001.

[20] Robbins, S., "Simulators for Teaching Operating Systems", http://vip.cs.utsa.edu/simulators/, UTSA, 2005

[21] Reeder, A., Ontko, R., "MOSS Scheduling Simulator", http://www.ontko.com/moss/sched/user_guide.html, Prentice-Hall, Inc., 2001.

[22] Weller, J., "CPU Scheduler Application", http://www.jimweller.net/jims/java_proc_sched/index.html, 2006.

[23] Blumenthal, J, "YASA - Yet another scheduling analyzer", http://yasa.e-technik.uni-rostock.de/overview.php, University of Rostock, 2003.

[24] Ferreira, p., "Uma Proposta de Aplicação do Modelo de Sistema de Ajuda de Engenharia Semiótica para Ambientes de Apoio ao Aprendizado", Trabalho de Conclusão, DICC/UERJ, 2004.