DENIZ PEDROZO DE ALMEIDA

AEO, UM ESCALONADOR DE OBJETOS PARA O SISTEMA OPERACIONAL AURORA.

FLORIANÓPOLIS – SC

MAIO DE 2002

UNIVERSIDADE FEDERAL DE SANTA CATARINA
FACULDADES INTEGRADAS CÂNDIDO RONDON
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA DA COMPUTAÇÃO

Deniz Pedrozo de Almeida

AEO, UM ESCALONADOR DE OBJETOS PARA O SISTEMA OPERACIONAL AURORA

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Luiz Carlos Zancanella - Dr. Sc.
(Orientador)

Florianópolis, maio de 2002.

AEO, UM ESCALONADOR DE OBJETOS PARA O SISTEMA OPERACIONAL AURORA

Deniz Pedrozo de Almeida

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, Área de Concentração - Sistemas de Computação e aprovada, em sua forma final, pelo Programa de Pós-Graduação em Ciência da Computação.

_____
Prof. Dr. Sc. João Bosco Mangueira Sobral

Banca Examinadora:

_____
Prof. Dr. Sc. Luiz Carlos Zancanella

_____
Prof. Dr. Sc. Rômulo Silva de Oliveira

_____
Prof. Dr. Sc. Luís Fernando Friedrich

_____
Prof. Dr. Sc. Olinto José Varela Furtado

Um vencedor diz:

- *“Pode ser difícil, mas é possível”.*

Um perdedor diz:

- *“Pode ser possível, mas é difícil”.*

O eterno perdedor diz:

- *“É impossível”.*

Agradecimento

Ao Criador de todos os sistemas existentes no universo - DEUS - que é a causa de toda a energia e propósito de cada ser. Sem Ele nada teria finalidade.

Um sentimento de gratidão vem externar às duas grandes instituições (Universidade Federal de Santa Catarina – UFSC e Faculdades Integradas Cândido Rondon – UNIRONDON) por permitirem a realização deste curso de mestrado, antecedendo a portaria do Ministério da Educação coibindo a realização deste tipo convênio em todo território o nacional poucos meses depois de aprovado este curso, haja vista que o mesmo foi de grande importância ao Estado de Mato Grosso, para as duas instituições e, principalmente, aos futuros mestres oriundos desta parceria.

Ao **Prof. Dr. Luiz Carlos Zancanella**, por permitir-me fazer parte deste projeto de grande importância ao meio acadêmico, científico e tecnológico. Por sua paciência e sapiência em nos orientar a todo instante em que necessitamos. Tornou-se muito mais que um orientador, tornou-se um amigo, e graças a um conjunto de boas qualidades foi possível o término deste projeto.

À minha **mãe Irinita Jovina Púlquério de Almeida**, heroína, amiga e uma grande vencedora, que em todos os instantes soube compreender a trabalhosa jornada deste trabalho.

Aos amigos que direta e indiretamente ofertaram sua parcela de contribuição, em especial a estes: **Emiliano Soares Monteiro, Márcia Cristina de Menezes Butakka , Valéria Cristina Pinto Ferraz, Tânia e Rejane**.

Índice

Lista de Abreviaturas

AE	Algoritmo de Escalonamento
AEO	Algoritmo de Escalonamento de Objetos
AP	Apontador de Pilha
AVLO	Access Virtual List Object
CI	Contador de Instrução
CPU	Unidade Central de Processamento
FCFS	First-Come, First-Served
FIFO	First-In-First-Out
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
POO	Programação Orientada a Objetos
MFQ	Multi-level Feedback Queues
PCB	Process Control Block
PC	Program Counter
RE	Registrador de Estado
RC	Reflexão Computacional
RR	Round-Robin
PSW	Status Word
SD	Sistemas Distribuídos
SJF	Job mais curto primeiro
SO	Sistemas Operacionais
SOD	Sistema Operacional Distribuído
SP	Stack Pointer
UML	Unified Modeling Language - Linguagem de Modelagem Unificada

Lista de figuras

Resumo

Este trabalho está inserido no contexto do Sistema Operacional Aurora, um sistema operacional orientado a objetos e reflexivo. Visa solucionar o problema de escalonamento de objetos, de modo a compartilhar, de forma justa e eficiente, o uso do processador entre os objetos ativos.

O modelo implementado neste projeto é produto de estudo e análise de várias políticas de escalonamento, o algoritmo obtido visou a uma solução capaz de se enquadrar adequadamente às necessidades do Sistema Operacional AURORA. Este modelo de escalonamento (Algoritmo de Escalonamento por Múltiplas Filas com Realimentação) apresenta em si o problema de envelhecimento de objetos na fila, ao qual, neste trabalho, foi dada uma solução.

Abordar-se-á a implementação do algoritmo **AEO (Algoritmo de Escalonamento de Objetos)**, que é o objeto de pesquisa deste trabalho, com ênfase à solução do problema de envelhecimento de objetos na fila, por conseguinte modelou-se com uso de UML (Unified Modeling Language – Linguagem de Modelagem Unificada) e implementou-se na linguagem de programação C++, com uso do IDE (Integrated Development Environment - Ambiente Integrado de desenvolvimento) C++ Buider 6.0.

Abstract

This stydy is inserted in the context of AURORA Operating System, an operating system oriented to objects and reflexive. It aims at solving the problem of scheduling of objects, in order to share, in a just and efficient way, the use of the processor among the octive objects.

The model implemented in this project is a product of study and analysis of several scheduling politics, the algorithm obtained allowed a solution capable of fit appropriately to the needs of AURORA Operating System . This scheduling model (Algorithm of Scheduling for Multiple Lines with Feedback) presents the problem of objects aging in the line, to which, in this study, a soluction was proposed.

It was proposed the implementation of the algorithm AEO (Algorithm of Objects Scheduling) with emphasis to the solution of the problem of objects aging in the line that is the object of this study. It was modeled with use of UML (Unified Modeling Language). It was implemanted through the programming language C++, with use of IDE (Integrated Development Environment) – C++ Builder 6.0.

1. Introdução

Enfocamos o gerenciamento de tarefas e thread, bem como a motivação e estrutura do tema pesquisado.

É crescente a necessidade de melhores computadores em diversos seguimentos de atividades. Atividades estas que buscam soluções aos problemas de menor a maior complexibilidade, desde um simples editor de texto a grandes sistemas de armazenamentos de dados; simples consulta de reserva em um hotel até a delicadas operações realizadas por uma equipe médica. As respostas esperadas são as mesmas: rapidez de processamento, facilidade de uso, confiabilidade, e, principalmente, segurança.

Para isso, necessita-se cada vez mais de um melhor aproveitamento dos computadores e arquitetura para atender a essas necessidades. A presença do sistema operacional é fundamental para melhor gerenciar tarefas, e possuir papel importante, visto ser ele o responsável pelo uso eficiente do processador. TANENBAUM (2000), como vários autores, tem afirmado a importância do escalonamento de processos em sistemas multiprogramados. *"O escalonamento de CPU é a base dos sistemas operacionais multiprogramados"*. *"A chave da multiprogramação é o escalonamento*" STALLINGS (1998).

SHAY (1996) ressalva que "*multiprogramação não significa necessariamente que muitos processos sejam executados simultaneamente"*, e sim, que *"somente um processo poderá ser executado de cada vez"*, dando oportunidade a todos os processos que aguardam oportunidade e momento para fazerem uso da CPU, considerando que o sistema computacional tenha apenas um processador. Os processos criados e parcialmente executados devem ser muitos bem gerenciados, para que a eficiência do sistema seja alcançada, considerando que cada processo terá um pequeno tempo para ser processado. E esse gerenciamento visa à distribuição *"justa"* do processador a todos os processos que estão na fila de prontos, isso para que nenhum usuário seja "tão" prejudicado, embora haja processos que não gozem de privilégios semelhantes: dependendo do tipo de gerenciamento (política de escalonamento), pode um processo ter maior prioridade em relação a outro. *"Na realidade, porém, a satisfação de um usuário pode vir somente à custa da satisfação de outro. O resultado é um usuário satisfeito e outro aborrecido, reclamando muito"* SHAY (1996). Mas o trabalho será na possibilidade de que essas insatisfações sejam minimizadas, dando mais prioridade aos processos que mais necessitem de fazer uso do processador. Esse gerenciamento é conhecido como **Escalonamento de tarefas,** como também **Escalonamento de**

Threads, (conceitualmente são iguais, "*muitas vezes usados indistintamente*" SILBERSCHATZ (2000), diferenciando o que está sendo escalonado; serão tratados mais adiante) e para esse escalonamento há várias políticas ou **Algoritmos de Escalonamento**.

Foram analisadas várias políticas (algoritmo de escalonamento), entre as quais, uma foi escolhida e implementada. Estes modelos de escalonamento foram adaptados para suportar o modelo de objetos (criando assim, um escalonador de objetos). O resultado foi a criação do **AEO (Algoritmo de Escalonamento de Objetos).**

Alguns autores têm citado a qualidade do **Algoritmo de Escalonamento por Múltiplas Filas com Realimentação**. SILBERSCHATZ (2000) afirma: *"A definição de escalonador por múltiplas filas com realimentação torna o algoritmo de escalonamento de CPU mais geral"*. SHAY (1996): "*O sistema de filas múltiplas de realimentação é um método flexível que se adapta automaticamente à carga de trabalho e às alterações nos padrões de comportamento dos processos*". MACHADO (1997) também afirma: "*O escalonamento por múltiplas filas com realimentação é um algoritmo de escalonamento genéricos que pode ser implementado em qualquer tipo de sistema operacional*".

Todo sistema operacional necessita de um algoritmo de escalonamento ou um escalonador, que é um conjunto de rotinas que gerenciam tempo aos processos e threads no uso do processador.

Para alguns autores processo e thread conceitualmente é a mesma coisa, porém há diferença entre eles.

Resumidamente conceitua-se **processos** como programas em execução, porém é um pouco mais que isso, ou seja, é o ambiente onde o programa se executa. *"Também inclui a atividade corrente, conforme representado pelo valor do* ***contador do programa*** *e o conteúdo dos registradores do processo. Um processo geralmente inclui a* ***pilha*** *de processo que contém dados temporários (como parâmetros de métodos, endereços de retorno e variáveis locais) e uma* ***seção de dados****, que contém variáveis globais",* SILBERSCHATZ (2000).

O sistema operacional cria o processo mediante uma estrutura de bloco de controle de processo (Process Control Block – PCB) o PCB encontram-se todas as

informações do processo, ou seja, sua identificação, prioridade, estado, recursos alocados pelo processo e informações do programa do qual se originou.

Um **thread** ou **processo leve** (*leghtweight process*) é uma unidade básica de CPU, contendo seu identificador, contador, registradores e uma pilha. Um thread compartilha com outros threads do mesmo processo sua sessão de código, de dados e outros recursos do sistema.

Um processo com apenas um thread denominado de processo pesado, ou seja, é o processo tradicional (com apenas um fluxo de execução).

Processos com muitos threads é denominado por multithread, isso quer dizer que podem ter processos com vários threads, cada um possui seu PC (Program Counter) e concorrem naturalmente ao processador como se fossem um único processo.

"A grande diferença entre sub-processos é em relação ao espaço de endereçamento. Enquanto sub-processos possuem, cada um, espaços independentes e protegidos, threads compartilham o mesmo espaço de endereço do processo, sem nenhuma proteção, permitindo assim que um thread possa alterar dados de outro thread. Apesar dessa possibilidade, threads são desenvolvidos para trabalhar de forma cooperativa, voltada para desempenhar uma tarefa em conjunto", MACHADO (1997).

Um objeto em execução será tratado como um thread. Baseado neste conceito será adotado um algoritmo de escalonamento existente ou, até mesmo, acrescentar pequenas mudanças ao algoritmo para adaptá-lo às necessidades do sistema operacional AURORA, se necessário.

1.1 Motivação

O objetivo deste trabalho é o estudo e elaboração deste escalonador para que melhor se enquadre ou que atenda às necessidades do Sistema Operacional AURORA.

Este sistema operacional é baseado na Tese de Doutorado em Ciência da Computação do Prof. Dr. Sc. Luiz Carlos ZANCANELLA (1997), "**Estrutura Reflexiva para Sistemas Operacionais Multiprocessados**". Foi elaborada uma pesquisa sobre diversos tipos de algoritmos de escalonamento e, finalmente, um

algoritmo foi escolhido. Este algoritmo foi alterado para suportar e controlar envelhecimento de objetos (activity) na fila. Neste trabalho encontra-se também a forma como este escalonador de objetos será inserido no sistema operacional AURORA (orientado a objetos, distribuído e reflexivo), o algoritmo foi denominado "**AEO - (Algoritmo de Escalonamento de Objetos)**".

A presente dissertação está dividida em sete capítulos, o primeiro, a própria introdução, em que são apresentados os motivos que conduziram ao estudo proposto, os demais, os objetivos e um resumo geral.

Faz-se no Capítulo 2 um histórico de escalonamento, um estudo sobre algumas políticas de escalonamento de processos, definindo-se os conceitos básicos sobre escalonamento, os algoritmos de escalonamento mais utilizados, as principais características de um escalonamento. Encerra-se o capítulo com uma breve análise das características de operações necessárias à implementação de um escalonador de múltiplas filas com realimentação para o gerenciamento do controle de objetos em sistemas operacionais orientados a objetos.

Apresenta-se no Capítulo 3 os sistemas operacionais mais conhecidos.

Um breve estudo sobre computação reflexiva é exposto no capítulo 4, os seus fundamentos e uma pequena descrição do sistema operacional AURORA (tese de doutorado do Dr. Prof. Sc. Luiz Carlos ZANCANELLA (1997)), tese esta que tornou possível elaborar e implementar o AEO.

É explanado no Capítulo 5 o "**AEO, um escalonador para o sistema operacional orientado a objetos AURORA**", proposto nesta dissertação. Inicialmente, descreve-se os motivos que conduziram ao desenvolvimento desse escalonador, apresentando-se uma breve introdução com as principais características de escalonamento. Em seguida, **"como"** esse escalonador foi desenvolvido e apresentação das estratégias de inserção e retirada de objetos da fila, com as devidas alterações, para serem utilizadas nessa nova abordagem, e, por fim, uma análise do novo escalonador apresentado.

No capítulo 6 é feita a análise conclusiva das principais vantagens obtidas com o escalonador para o sistema operacional AURORA, bem como validação e simulações

efetuadas com o escalonador e a apresentação final de algumas sugestões de trabalhos futuros.

2. Escalonamento

Inicialmente uma breve história sobre o escalonamento de tarefas, os conceitos básicos e os principais algoritmos de escalonamento, e também, um estudo sucinto sobre ***job****, processos e tarefas, destacando-se os diferentes conceitos de alguns autores e os possíveis estados que uma tarefa ou processo pode assumir. Finaliza-se destacando-se o* ***Escalonador de Tarefas por Múltiplas Filas com Realimentação****, e as principais características que ele deve ter para ser utilizado no gerenciamento do controle de processo.*

2.1 Evolução dos Computadores

Desde o surgimento da humanidade até os dias atuais, principalmente, percebe-se a grande necessidade de manipulação e gerenciamento de informações. Das escritas nas cavernas aos papiros no Egito, dos couros aos papéis em diferentes civilizações, das madeiras tecidas aos mais sofisticados discos magnéticos com imensas capacidades para armazenamentos de informações. De suma importância foram alguns dos inventos da história da humanidade, os quais tiveram um pouco de participação para os atuais computadores, ou seja, das primeiras calculadoras surgidas aos gigantescos computadores de décadas atrás, e o mais importante, as poderosas máquinas hoje existentes, com grande capacidade de processamento e armazenamento.

Das afirmações de conceituados estudiosos, Prof. Dr. Sc. Jorge Muniz BARRETO (2000): *"Atualmente parece que cabe a Wilhem Schickard (1592-1635), professor de Matemática, Astronomia e Hebreu na Universidade de Heidelberg, o mérito de ter inventado e construído a* ***primeira calculadora*** *mecânica em 1623, provavelmente para facilitar seus cálculos astronômicos"*. Como também o Prof. Dr. Andrew S. Tanenbaum TANENBAUM (2000), em seu livro "*Sistema Operacional Moderno*", atribui ao matemático inglês Charles Babbage (1792-1871), projetor e construtor do **primeiro computador digital**, não concluído em função de, na época, não haver associado à sua máquina analítica um sistema operacional e, é claro, um escalonador.

A evolução dos computadores esteve quase adormecida entre 1842 e 1945 BARRETO (2000). Porém reaparecem os primeiros computadores eletromecânicos no início da Segunda Guerra Mundial, estes eram extremamente grandes em relação aos existentes hoje, ocupavam imensos espaços físicos e faziam meros cálculos matemáticos. E em décadas posteriores foram marcados com importantes inovações tecnológicas.

As **linguagens de programação** vieram a facilitar a vida dos programadores, pois as programações dos computadores eram feitas diretamente no hardware, em pequenos painéis, exigindo-se profundo conhecimento dele. As primeiras linguagens de programação foram o *Assembly* e *Fortran*.

Novas utilidades foram dadas aos computadores, tornando-se difíceis os trabalhos manuais, inspirando novas tecnologias. Com esse anseio apareceram os primeiros

sistemas operacionais. A IBM desenvolve o sistema operacional OS/360, caracterizado pela portabilidade em qualquer computador, desenvolvido especialmente para a série 360, em 1964, que revolucionou a indústria de informática, "*pois produzia uma linha de computadores pequena, poderosa e, principalmente, compatível. Isso permitiu que uma empresa adquirisse um modelo mais simples e barato e, conforme suas necessidades, mudasse para modelos com mais recursos, sem comprometer suas aplicações já existentes*", MACHADO (1997).

Apesar da grande importância do avanço tecnológico para a época, exigia-se cada vez mais melhores performances nestes computadores, tornando fundamental o surgimento dos escalonadores de tarefas. Na série 360, utilizava-se o conceito de processo, ou seja, o escalonador era uma pequena rotina que fazia parte do sistema e tinha por responsabilidade ativação e encaminhamentos de jobs, estes eram processos que ficavam armazenados em disco à espera de execução. Quando os processos passaram a ter dependência do tempo, houve também necessidade de aperfeiçoamento na política de escalonamento, e assim aconteceu: os processos passaram a ser tratados como tarefas.

Os sistemas operacionais foram melhorando, aos computadores atribuíram-se novas atividades impulsionadas pelo setor comercial e com essas melhorias e necessidades se destacam também a evolução do escalonamento de tarefas. Os recursos e CPU (Unidade Central de Processamento) passaram a ser usados por vários usuários ao mesmo tempo e precisavam ser compartilhados entre esses usuários. A CPU necessitava ficar quase 100% de seu tempo ocupado. Para isso, o escalonador deveria garantir a eficiência da CPU. Com esta eficiência adquirida no escalonador, certa categoria de usuário foi prejudicada no uso da CPU, percebeu-se, então, a necessidade de eficiência e distribuição "**justa**" a todos os usuários de forma que ninguém viesse a ser prejudicado e, também, não perder essa eficiência.

A condução para origem de diferentes algoritmos foi quase automática, buscando, assim, o melhor desempenho do sistema para cada tipo de usuário. Porém as necessidades de um sistema não se resumiam apenas a um tipo de usuário, e sim, a inúmeros tipos de usuários, como também diferentes características entre eles. Quanto

mais justa a distribuição de tempo e uso do processador pela tarefa, melhor é o algoritmo.

2.2 Tarefas, Processos e Thread

O estudo sobre escalonamento induz ao estudo dos conceitos de *jobs*, tarefas, processos e thread. Alguns autores SHAY (1996), MACHADO (1997), SILBERSCHATZ (2000) e OLIVEIRA (2000) os tratam como se eles representassem o mesmo conceito, ou seja, um programa em execução, porém com importantes características, vistas mais abaixo. Atualmente o conceito de *job* é pouco utilizado, visto que o mesmo é quase sempre relacionado com os antigos sistemas em lote, em que um *job* representava um programa ou um conjunto de programas postos para executar, TANENBAUM (1999). Um processo representa um programa ou tarefas em execução, sendo constituído do código executável, dos dados referentes ao código, da pilha de execução, do valor do contador de programa (registrador PC, do inglês *Program Counter*), do valor do apontador de pilha (registrador SP, do inglês *Stack Pointer*) e de outras informações necessárias à execução do programa, TANENBAUM (1999). Muitos outros autores tais como: MACHADO (1997) e SILBERSCHATZ (2000) apresentam esse mesmo conceito, destacando que um processo é uma abstração de um programa sendo executado. Para SHAY (1996) o conceito de processo é equivalente ao de tarefa. Então, é possível concluir que a maioria dos procedimentos adotados para tratamento dos processos pode ser aplicada também para tratamento das tarefas.

O sistema operacional cria o processo mediante uma estrutura de bloco de controle de processo (Process Control Block – PCB). No PCB encontram-se todas as informações do processo, ou seja, sua identificação, prioridade, estado, recursos alocados pelo processo e as informações do qual se originaram.

Um **thread** ou **processo leve** (*leghtweight process*) é uma unidade básica de CPU, com seu identificador, contador, registradores e uma pilha. Um thread compartilha com outras threads do mesmo processo sua sessão de código, de dados e outros recursos do sistema.

Um processo com apenas um thread é denominado de processo pesado, ou seja, com apenas um fluxo de execução.

Processos com muitos threads são denominados de multithread, isso quer dizer que pode haver processo com vários threads, cada um possui seu PC (Program Counter) e concorre naturalmente ao processador como se fosse um único processo.

2.3 Estados das Tarefas

A mudança de comportamento de um processo é constante, da sua criação à destruição. A definição do estado de uma tarefa é atribuída à sua atividade atual. Na figura abaixo são descritos os estados tradicionais dos sistemas operacionais, ou seja, **Novo**, **Pronto**, **em Espera**, **em Execução** e **Terminado**. Em algumas versões do sistema operacional Linux há também o estado de ZUMBI.

Na mudança de um estado para outro existem princípios: O estado "*Novo*" e "*Terminado*" representam, respectivamente, a **criação** e **término** do processo. Após o processo criado, este deve obrigatoriamente ir para o estado de "*Pronto*". O estado de "*Terminado*" somente acontece quando o processo está "*em Execução*". E, por fim, o estado de "*em Espera*" acontece quando há uma solicitação de entrada/saída. Após sair desse estado, deve obrigatoriamente retornar ao estado de "*Pronto*".

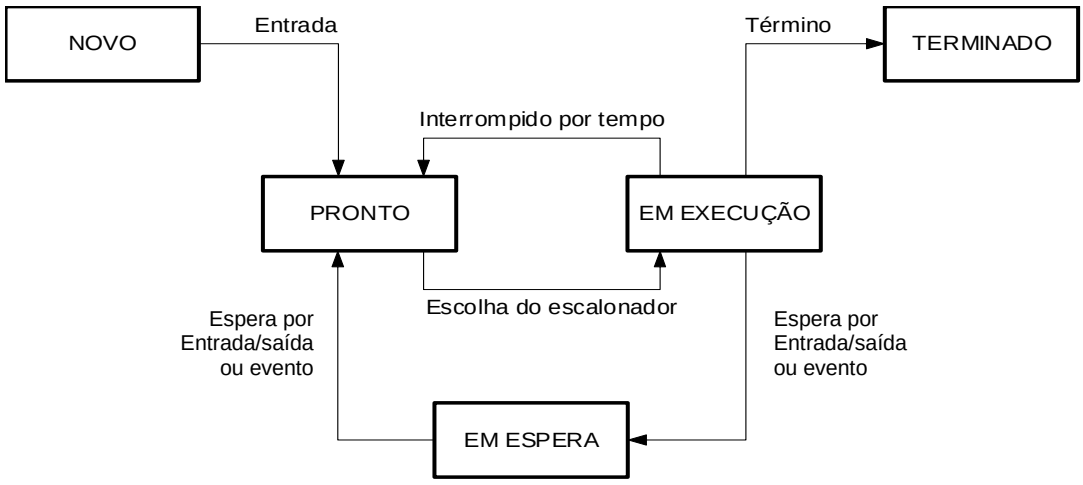


Figura 2.1 Estados das tarefas e transições entre estados.

Enquanto houver a existência do processo, em um dos estados seguintes deve estar a processo.

- **Novo**: Origem, ativação de uma nova tarefa.
- **Em execução**: são executadas pela CPU as instruções da tarefa. Em computador com um único processador se processa uma tarefa por vez.
- **Em espera**: aguarda por algum evento externo ou recurso do sistema para dar continuidade à sua existência. Por exemplo, uma solicitação de Entrada/saída, uma espera de data ou hora do sistema.

  Alguns sistemas subdividem o estado de espera, ou seja, em estado de bloqueado e espera. Esta diferença é que uma tarefa em estado de bloqueado aguarda para ser autorizado pela utilização de algum recurso, enquanto em estado de espera aguarda pelo término de uma operação em um recurso que já foi garantido, MACHADO (1997).
- **Pronto:** o estado é dito “pronto” quando a tarefa está na fila aguardando por sua vez ao uso da CPU, gerenciado pelo sistema operacional.
- **Encerrado:** é terminada a execução da tarefa.

2.4 Bloco de Controle das Tarefas

Para cada processo criado é associado a ele um pacote de informações denominado de bloco de controle de processo (PCB – Process Control Block) ou bloco de controle de tarefa, SILBERSCHATZ (2000). A figura 2.2 descreve as principais informações manipuladas no PCB.

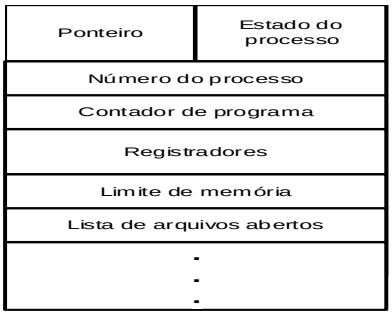


Figura 2.2 Bloco de controle de processo (PCB).

- Estado do processo: O estado do processo pode ser novo, pronto, em execução, em espera, em suspenso ou bloqueado e terminado. No sistema operacional UNIX há também o estado zumbi, SHAY (1996).
- Contador de programa (PC – Program Counter): O endereço da próxima instrução da tarefa é indicado pelo PC.
- Registradores de CPU: dispositivo de alta velocidade localizada no CPU. Em função da arquitetura de cada processador os números de registradores variam. Os principais registradores são: PC ou contador de instrução (CI) responsável pelo próximo endereço de instrução, "*apontador de pilha (AP) ou stack pointer (SP), onde há o endereço de memória do topo da pilha, que é a estrutura de dados onde o sistema mantém informações sobre tarefas que estavam sendo processadas e tiveram que ser interrompidas por algum motivo"* MACHADO (1997), registrador de estado (RE) ou status word (PSW), responsável pelo armazenamento de informações sobre a execução do programa, como ocorrência de *carry* e *overflow*. Essas informações são alteradas todas as vezes que o processo faz uso da CPU para que continue onde parou.
- Informações de escalonamento de CPU: Essas informações incluem prioridade da tarefa e quaisquer outras informações e argumentos de escalonamento.
- Informações de gerência de memória: Controle de registradores de bases e limites, tabela de paginação e, dependendo do sistema usado pelo sistema operacional, a tabela de segmentação.
- Informações de contabilização: Informação de quantidade de CPU, Jobs, tarefas ou processos etc.
- Informações de status de Entrada/Saída: Informações de listas de dispositivos de Entrada/Saída alocadas para este processo, uma lista de arquivos abertos etc, SILBERSCHATZ (2000).

Estas informações podem variar de tarefa a tarefa, pois cada tarefa possui comportamento diferente.

2.5 Threads

O modelo de processo tradicional, discutido anteriormente, considerava categoricamente que um processo é um programa que realiza, do começo ao final, um único fluxo de execução, permite que o processo execute uma tarefa por vez, que não é o caso de um thread. Na figura abaixo há uma pequena descrição de dois processos em execução, uma tarefa A e outra tarefa B. Caracterizam-se tarefas: sair do estado de execução, salvar o estado no PCB, recarregar logo em seguida o estado do PCB da tarefa B, ficando enquanto isso a tarefa A inativa.

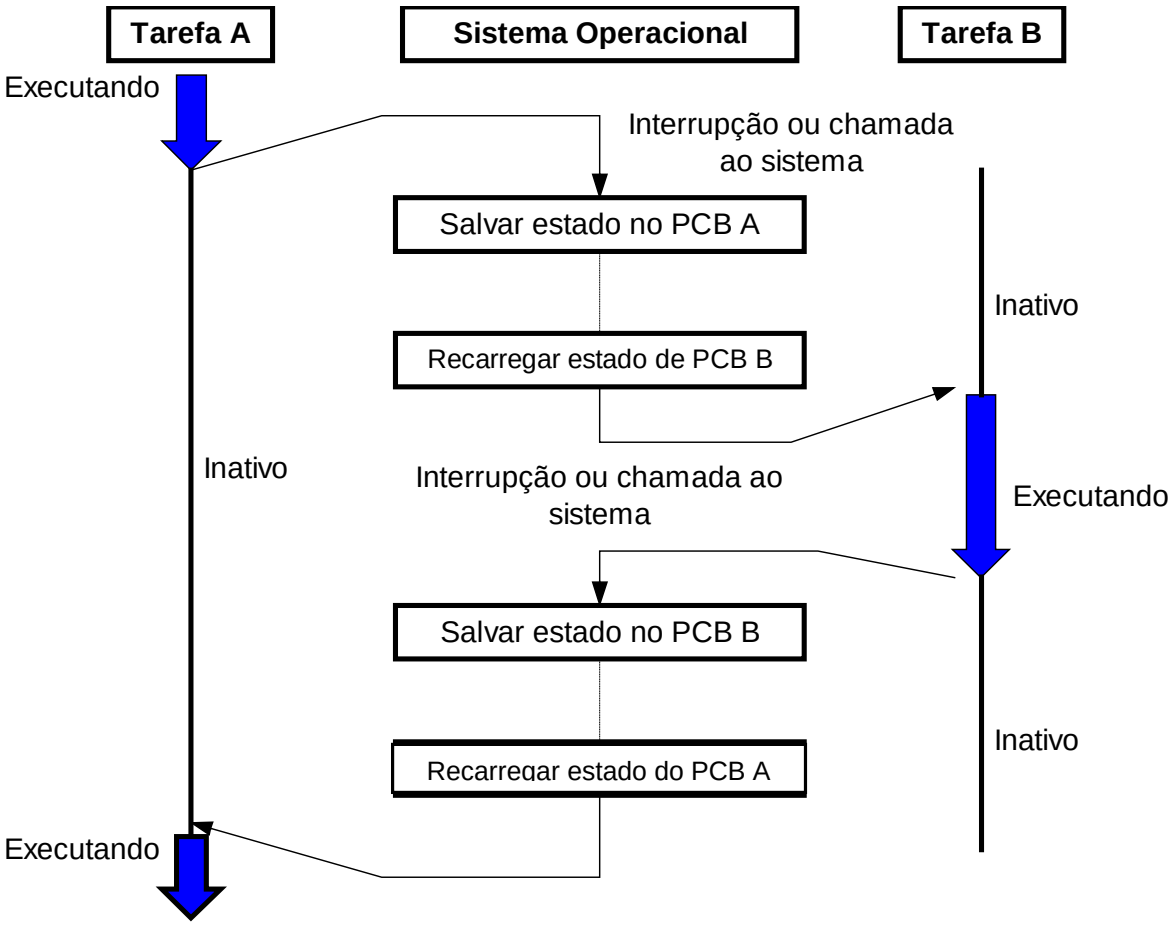


Figura 2.3 Diagramas mostrando a CPU alternando entre processos, SILBERSCHATZ (2000).

Nos sistemas operacionais modernos, houve necessidade de mudança no conceito de processo, acrescentando novos atributos, permitindo que um processo tenha

múltiplos fluxos de execução ou threads. Nesta visão, pode o processador executar mais de uma tarefa ao mesmo tempo.

2.6 Escalonamento de Processos

A multiprogramação não existiria se o sistema operacional não pudesse compartilhar tempo aos inúmeros processos que esperam pela sua vez de execução, ou seja, fazer uso da CPU. Na multiprogramação deve-se manter a CPU quase 100% em uso, e para que isso aconteça, deve haver um conjunto de procedimentos para distribuir tempo a cada processo que aguardar na fila de espera para fazer uso do processador. Para tanto é de fundamental importância uma boa política de escalonamento destes processos que se encontram na fila.

2.6.1 Filas de Escalonamento

"À medida que os processos entram no sistema, são colocados em uma fila de jobs. Essa fila consiste em todos os processos do sistema", SILBERSCHATZ (2000). Normalmente mantidos em uma lista encadeados, os processos em estado de pronto e aguardando a vez de serem executados são colocados em uma fila, denominados filas de processos prontas **(ready queue)**. Há sempre um apontador para o primeiro e último processo da fila, com finalidade de incluir ou excluir processos da fila.

Mas há outras filas no sistema, ou seja, fila para processos encerrados, fila para processos interrompidos ou fila para espera pela ocorrência de determinado evento. Os processos que esperam por determinados recursos de Entrada/Saída são denominados de filas de dispositivos de Entrada/Saída. A figura 2.4 mostra uma fila de processos no estado de prontos, aguardando sua vez para ir ao processador.

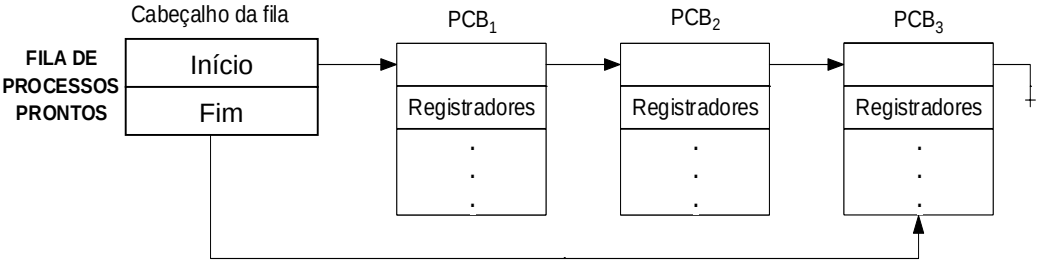


Figura 2.4 Fila de processos prontos, SILBERSCHATZ (2000).

Por meio de um diagrama de filas pode-se entender melhor os diversos escalonamentos que deve haver em função dos diferentes estados do processo.

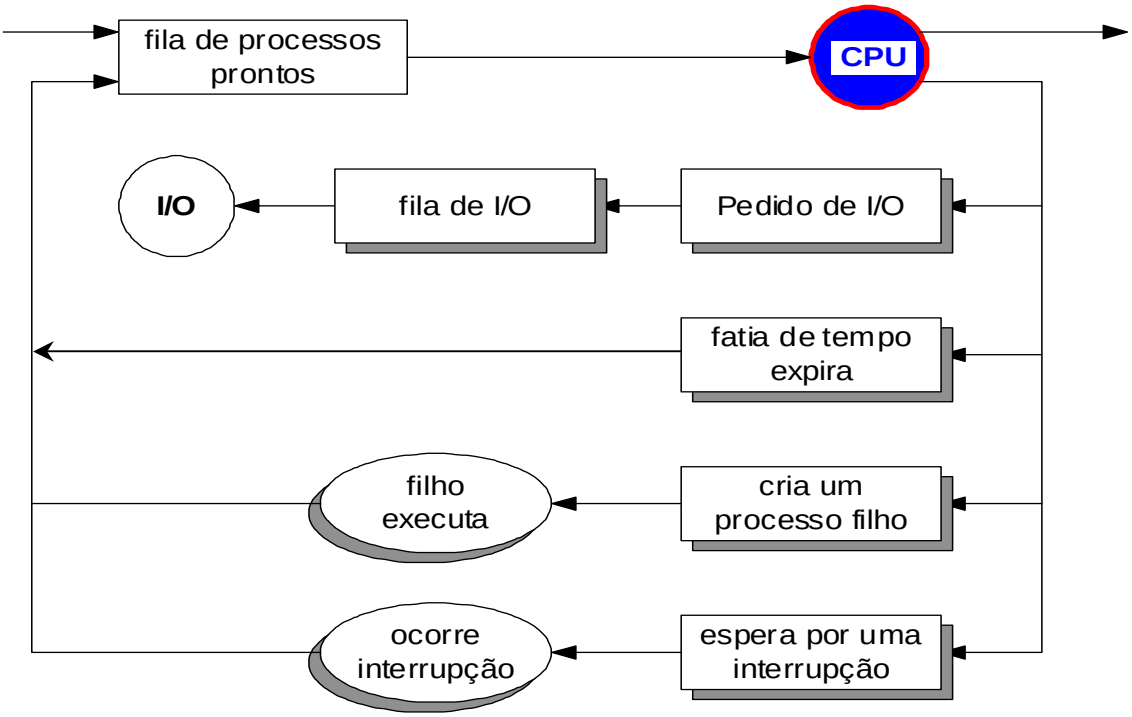


Figura 2.5 Diagrama de filas do escalonamento de processos, SILBERSCHATZ (2000).

Quando o processo é ativado, inserido é no final da lista encadeada (fila de processos), que é a fila de processos prontos. O processo permanece na fila até que seja sua vez, e quando o processo é selecionado para fazer uso do processador podem ocorrer várias situações:

- Solicitar um pedido de Entrada/Saída e ficar na fila de Entrada/Saída até que seja atendido;
- Criar um subprocesso (processo filho) e o processo pai esperam pelo término do processo filho, pois pode depender de algum resultado deste subprocesso;
- Acontecer uma interrupção, forçando o processo a sair da CPU e retornar à fila de processos prontos, mesmo que não tenha terminado a sua fatia de tempo.

Um processo nunca é executado sem que passe pela fila de processos prontos, ou seja, caso esteja bloqueado, em espera de algum evento ou interrompido por algum motivo, deve o processo voltar ao final da fila de processos prontos.

2.6.2 Escalonadores

O processo criado no sistema tem um ciclo bastante diversificado, pode ser executado rapidamente ou necessitar de maior tempo para o seu término. Devido aos diferentes estados que podem acorrer em um processo em toda sua existência, o sistema operacional deve preocupa-se em selecionar, com propósito de escalonamento, os processos que se encontram em alguma outra fila, por exemplo, fila de dispositivos de Entrada/Saída de dados.

Basicamente pode-se dizer que há escalonamento de curto e longo prazo, diferenciando-se na freqüência da sua execução. O escalonador de longo prazo pode ficar até minutos sem que se crie novo processo, enquanto no escalonador de curto prazo a freqüência é bastante curta, executando pelo menos uma vez a cada 100 milissegundos.

No escalonamento de curto prazo pode haver desperdício de processamento caso haja demora na decisão de qual processo irá para a CPU (Unidade Central de Processamento), por isso deve ser rápido.

O escalonador de longo prazo é responsável pelo grau de multiprogramação (o número de processos na memória). *"A taxa média de criação de processos deve ser igual à taxa média de partida de processos que saem do sistema",* SILBERSCHATZ (2000), caso o grau de multiprogramação for estável. Nesse caso, o escalonador de longo prazo só precisa ser chamado quando o processo sai do sistema. A decisão de escolha de qual processo deva ir para a CPU pode se tornar mais demorada em função de o intervalo entre processos ser maior.

2.6.3 Troca de Contexto

"Salvar o conteúdo dos registradores e carregá-los com os valores referentes ao do processo que esteja ganhando a utilização do processador", MACHADO (1997)

chama-se de troca de contexto. No PCB de cada processo armazena-se o contexto do processo, compreendendo o valor dos registradores de CPU, o estado do processo e as informações de gerência de memória. Esta troca se resume em substituir o contexto de hardware de um processo pelo outro. Descrita na figura 2.3 acima.

2.7 Escalonamento de CPU

Os sistemas operacionais multiprogramáveis têm como base o escalonamento de CPU, tornando o computador produtivo na alternância de CPU entre processos, SILBERSCHATZ (2000).

A CPU deve estar sempre com algum processo em execução, que é o ideal, ou seja, maximizar o uso de CPU. Em sistemas operacionais uniprocessados executará um processo por vez, caso tenha mais de um processo em fila de espera.

Em sistemas uniprocessados haverá uma perda de tempo relativamente grande, dependendo da quantidade de solicitação de Entrada/Saída de dados. Nestas solicitações ou em outros eventos, o processador ficará ocioso. Na multiprogramação isso é contornado, enquanto um processo solicita uma Entrada/Saída, um outro processo faz uso da CPU, dando maximização de uso de CPU.

"O escalonamento é uma função fundamental do sistema operacional. Quase todos os recursos do computador são escalonados antes do uso. A CPU, é claro, é um dos principais recursos do computador. Assim, o escalonamento é central ao projeto de sistemas operacionais", SILBERSCHATZ (2000).

Vários fatores influenciarão no bom escalonamento de CPU, como tempo de processamento e espera de Entrada/Saída. "*O escalonamento somente afeta o tempo de espera de processos na fila de pronto"*, MACHADO (1997).

2.7.1 Critério de Escalonamento

No compartilhamento de tempo de processos em sistema operacional necessita-se de critérios para um bom funcionamento no processamento. Segue abaixo alguns

critérios sugeridos por Abraham SILBERSCHATZ (2000), que será determinante na escolha da política de escalonamento:

- **Utilização da UCP**: é fundamental que o processador do sistema operacional fique quase 100% da sua capacidade de processamento em atividade;
- **Imparcialidade**: distribuir de forma "justa" o tempo de processamento para cada processo na memória do computador;
- **Throughput**: o número de processos executados em um determinado intervalo de tempo é chamado de throughput. Em tarefas longas pode ser usado como medida a hora, enquanto nas curtas, pode ser o throughput de 10 processos por segundo;
- **Turnaround (Tempo de retorno)**: denomina-se **turnaround** por MACHADO (1997) ou **Tempo de retorno** por SILBERSCHATZ (2000). É o tempo que leva o processo para terminar sua execução, desde o momento de estado de pronto ao estado de terminado. "*Esse tempo é a soma dos períodos gastos esperando para acessar a memória, aguardando na fila de processos prontos, executando na CPU e realizando operações de entrada/saída*", SILBERSCHATZ (2000);
- **Tempo de espera**: é a soma dos períodos gastos esperando na fila de processos prontos;
- **Tempo de resposta**: é a medida do tempo entre a submissão de um pedido até a primeira resposta produzida, ou seja, é o tempo que o processo leva para começar a responder, mas não é o tempo que leva para gerar a resposta. Este tempo é determinado pela velocidade do dispositivo de saída, indo de uma solicitação a uma resposta, SILBERSCHATZ (2000).

O ideal seria que fossem otimizados a utilização da CPU e o throughput e minimizar os tempos de retorno e resposta.

Há sugestão de pesquisadores que para sistemas interativos (como os de tempo compartilhado) enfatiza-se a maximização da variância no tempo de resposta do que a maximização da média de tempo de resposta. "*No entanto, pouco trabalho foi realizado*

em algoritmos de escalonamento de CPU que minimiza a variância" SILBERSCHATZ (2000).

Há duas classes principais de algoritmos de escalonamentos: **Não-preemptivo** e **Preemptivo**.

- **Escalonamentos não-preemptivos:** A vantagem de escalonamento não-preemptivo é a simplicidade, porém a desvantagem é quanto ao uso da CPU, ou seja, quando um processo assume o controle da CPU, manterá controle até o seu término, não permitindo que outro processo qualquer faça uso da CPU.
- **Escalonamentos preemptivos:** A característica de escalonamento preemptivo é o compartilhamento de tempo aos processos, permite prioridade aos processos, por exemplo, aos sistemas de tempo real. Outra grande vantagem desta política é o compartilhamento de CPU por vários processos alternadamente.

São quatro as circunstâncias para que haja decisão de escalonamento:

- Na mudança de estado do processo de **estado de executando** para o **estado de espera** (por exemplo, uma solicitação de entrada/saída e espera de término de um outro evento, como por exemplo, um processo filho);
- Na mudança do estado do processo de **estado de execução** para o **estado de pronto** (por exemplo, uma interrupção);
- Na mudança do estado do processo de **estado de espera** para o **estado de pronto** (por exemplo, conclusão de uma solicitação de entrada/saída);
- Término de um processo.

Dizemos que será **não-preemptivo** ou **cooperativo** no primeiro e último caso, caso contrário será **preemptivo**.

Um sistema operacional que usou o **não-preemptivo** ou **cooperativo** foi Microsoft Windows 3.x; o Windows 95 em diante passou-se a fazer uso de escalonamento preemptivo. O escalonamento preemptivo foi também introduzido no sistema operacional Apple Macintosh no MacOS 8 para arquitetura PowerPC.

2.7.2 Escalonamento First-In-First-Out (FIFO)

Alguns autores o chamam de **First-In-First-Out (FIFO):** MACHADO (1997), SHAY (1996) e OLIVEIRA (2000). Enquanto outros o chamam de **First-Come, First-Served (FCFS):** SILBERSCHATZ (2000) e STALLINGS (1998). No entanto, se referem à mesma política. Caracteriza-se FIFO pela sua implementação e modelo de uma lista (primeiro que entra, primeiro que sai), considerando que será alocado sempre o primeiro da fila, inserindo novos processos ao final da fila, demonstrados na figura 2.6.

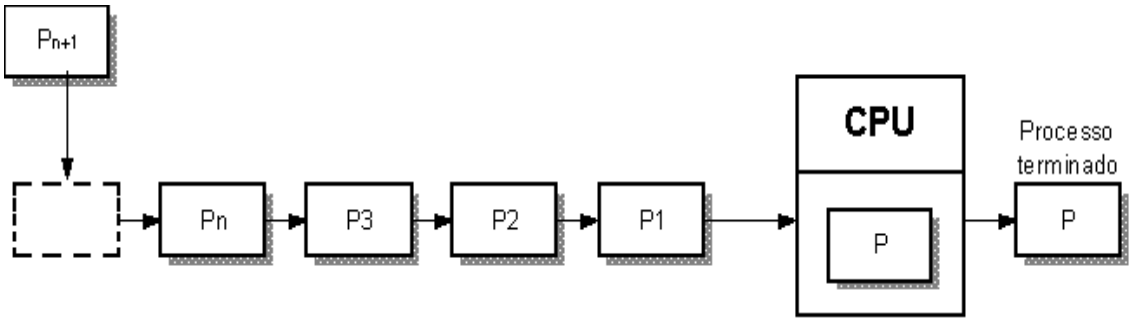


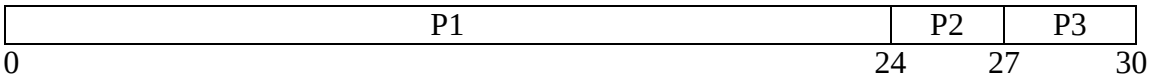
Figura 2.6 Escalonamento First-In-First-Out (FIFO).

As desvantagens são muitas, pequenos processos são prejudicados por processos maiores, tendo que esperar sua vez. Neste caso não há compartilhamento de tempo, e o FIFO em tempo compartilhado já é muito usado, porém com mais eficiência.

O tempo médio de espera nesta política costuma ser longo. Veja-se um exemplo abaixo com o tempo expresso em milissegundos, SILBERSCHATZ (2000):

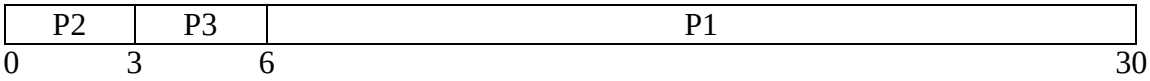
Processo N.º	Duração de surto
P1	24
P2	3
P3	3

Pelo **diagrama de gantt** podemos ter o seguinte resultado com a seqüência dos processos em P1, P2 e P3:



O tempo de espera é 0 milissegundos para o processo P1, 24 milissegundos para o processo P2 e 27 milissegundos para o processo P3. Assim, o tempo de espera médio é (0 + 24 + 27) / 3 = 17 milissegundos.

Veja-se outro exemplo, com ordem dos processos alterados, ou seja, P2, P3 e P1, SILBERSCHATZ (2000):



Neste exemplo o tempo de espera foi substancialmente menor que o anterior, ou seja, (6 + 0 + 3) / 3 = 3 milissegundos, isso demonstra que o tempo médio de espera será de acordo com surto do processo.

O algoritmo de escalonamento FCFS pertence à classe não-preemptivo. Uma vez que um processo tomasse a CPU, não mais sairia enquanto não realizasse uma solicitação de entrada/saída ou que o processo terminasse. Em função disso, não é interessante que um processo fique por muito tempo em poder da CPU.

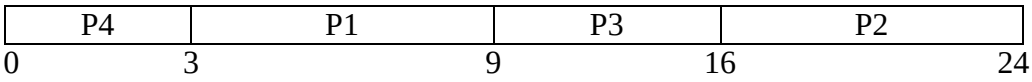
2.7.3 Escalonamento Shortest-Job-First (SJF)

Pertence à classe de algoritmos não-preemptivos, em inglês **Shortest-Job-First (SJF):** MACHADO (1997) e STALLINGS (1998). Em português **job mais curto primeiro:** SILBERSCHATZ (2000). Ou ainda, **menor job primeiro:** TANENBAUM (1995). Abraham SILBERSCHATZ (2000) diz que o termo mais apropriado seria **próximo surto de CPU mais curto,** por ser o escalonamento feito a partir do exame da duração do próximo surto da CPU de um processo, em vez de sua duração total. As referências são ao mesmo algoritmo, o SJF. Neste escalonamento, a política é dar prioridade aos pequenos processos, ou seja, sempre que um novo processo entra em estado de pronto é feita uma comparação dele com os já existentes na fila, caso ele seja menor que os demais, terá prioridade de execução.

Veja um exemplo abaixo, com o tempo expresso em milissegundos, SILBERSCHATZ (2000):

Processo N.º	Duração de surto
P1	6
P2	8
P3	7
P4	3

Com o algoritmo de escalonamento SJF, o escalonamento dos processos acima ficaria com o tempo médio de espera.

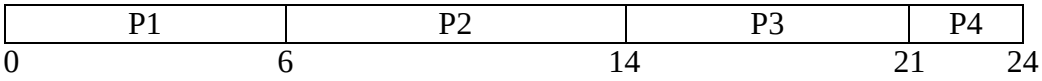


A média do tempo de espera desses processos escalonados seria de 7 milissegundos. Veja-se o tempo de espera em cada processo:

Processo N.º	Milissegundos
P4	0
P1	3
P3	9
P2	16

Assim, o tempo médio de espera será: (0 + 3 + 9 + 16) / 4 = 7 milissegundos.

Se esses mesmos processos fossem escalonados usando o algoritmo de escalonamento FCFS (FIFO) o tempo médio iria ser maior, (0 + 6 + 14 + 21) / 4 = 10,25, veja-se abaixo:



*"O algoritmo de escalonamento SJF é comprovadamente **ótimo**, pois fornece o tempo médio de espera para um determinado conjunto de processos"*, SILBERSCHATZ (2000). O tempo médio de espera diminui quando se colocam jobs menores antes dos maiores.

O escalonamento SJF é geralmente usado em escalonamento de longo prazo, pela dificuldade em saber a duração do próximo pedido, ou seja, não há como saber a duração do próximo surto de CPU, SILBERSCHATZ (2000), podendo-se prever qual será o próximo surto.

2.7.4 Escalonamento Circular (ROUND ROBIN - RR)

Algoritmo de **escalonamento circular (Round Robin):** MACHADO (1997). Algoritmo de **escalonamento por revezamento:** SHAY (1996). Ou ainda **Round Robin** por diversos autores: OLIVEIRA (2000), SHAY (1996) e MACHADO (1997).

A política de escalonamento é semelhante ao algoritmo de escalonamento FCFS, alterando esse processo com o incremento de preempção. Uma pequena unidade de tempo é dada a cada processo, denominada de **quantum de tempo**, **fatia de tempo** ou **time-slice**. Estas denominações de tempo são normalmente de 10 a 100 milissegundos, implementada em uma fila circular.

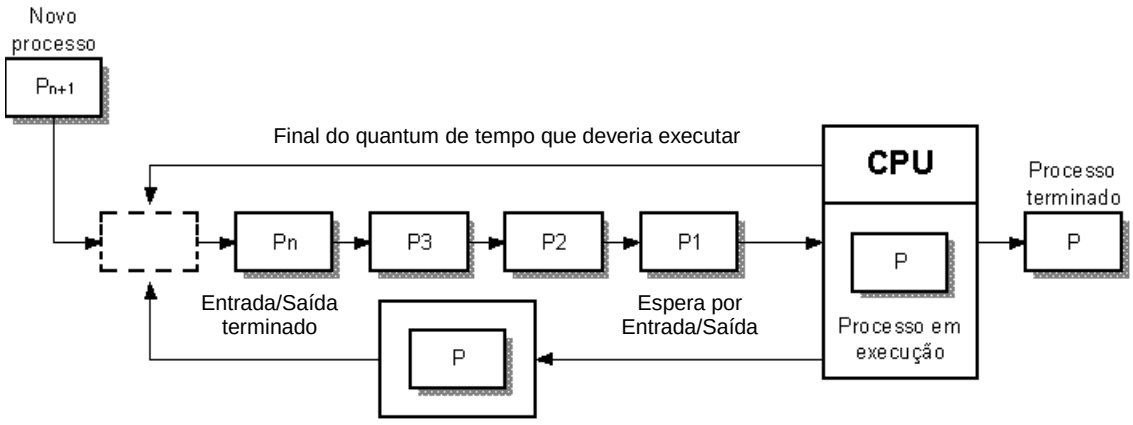


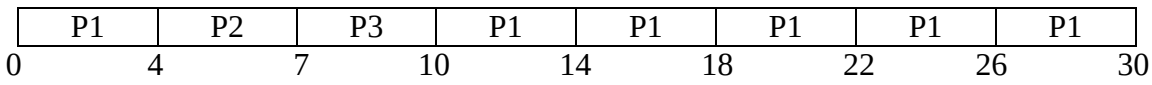
Figura 2.7 Escalonamento Round-Robin (RR).

O escalonador de CPU seleciona o primeiro processo da fila, define um temporizador para interromper depois de 1 (um) quantum e submete o processo", SILBERSCHATZ (2000).

Quando um processo toma a CPU pode ser interrompido por duas razões: o processo termina sua execução antes do término do seu quantum, o próprio processo se responsabiliza em sair da CPU, e o escalonador submete um outro processo da fila de prontos na CPU. Caso contrário, se o surto de tempo for maior que o quantum de tempo, ocorrerá uma interrupção para o sistema operacional. São feitas as trocas de contexto de hardware e do processo colocado na fila de prontos. Escalona-se outro processo da fila de prontos.

O tempo médio de espera no algoritmo de escalonamento RR é normalmente grande, em função da preemptição aplicada à tarefa, ou seja, alguma tarefa acaba voltando à fila de espera por razão do não término de seu surto total, uma vez terminado o tempo para uso do CPU. Veja um exemplo abaixo, SILBERSCHATZ (2000):

Processo N.º	Duração de surto
P1	24
P2	3
P3	3



O quantum de tempo é de 4 milissegundos, o P1 usará 4 milissegundos. Como o seu surto de tempo é de 20, irá retornar à fila de prontos para esperar sua vez. O segundo (P2) é encaminhado na CPU, não usando o quantum que lhe cabe, termina a execução. Como o terceiro processo (P3) também possui o surto de tempo de 3 milissegundos, não terminará o quantum de tempo.

2.7.5 Escalonamento por Prioridade

O algoritmo de escalonamento por prioridade é um algoritmo SJF com característica especial, a prioridade. Cada processo possui uma prioridade, sendo que a mais alta tem direito prioritário ao uso de CPU. São escalonados na ordem FCFS os processos com igual prioridade.

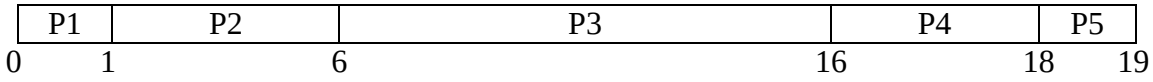
Normalmente as prioridades são definidas por um número inteiro, variando de sistema a sistema, por exemplo, o sistema operacional Windows de 32 bits utiliza prioridade de 0 (maior) a 31 (menor), o nível 0 (zero) recebe um nome especial "*escalonamento de página zero*", JAMSA (1999). Outro exemplo é o sistema operacional VMS, que também utiliza 32 prioridades, em que 16 a 31 são para processos em tempo real, enquanto os processos normais fazem uso 0 a 15, SHAY (1996).

Os processos no sistema operacional Unix também possuem essas prioridades, variando de 0 (maior) a 127 (menor), sendo: 0 a 49 para o kernel (processos do sistema), 50 a 127 para user-mode (processos do usuário).

Baseados em um conjunto de processos, no exemplo abaixo, com duração do surto expresso em milissegundos, é exemplificado o uso do algoritmo de escalonamento por prioridade, SILBERSCHATZ (2000).

Processo N.º	Prioridade	Duração de surto
P1	3	10
P2	1	1
P3	4	2
P4	5	1
P5	2	5

No diagrama de Gant, esses processos ficariam assim:



O tempo médio de espera será (0 + 1 + 6 + 16 + 18) / 5 = 8,2 milissegundos.

Um grande problema no algoritmo por prioridade é o bloqueio por tempo indefinido ou *starvation* (estagnação). Isso pode ocorrer em função de processos com baixa prioridade que nunca venham a fazer uso da CPU, ficando indefinidamente bloqueados.

Para bloqueio indefinido é aplicada a solução de processos envelhecidos (aging). Esta técnica aumenta a prioridade do processo gradualmente em um determinado tempo. Por exemplo, um processo pode receber um ponto a cada 15 minutos; em algum instante este processo atingirá prioridade mais alta e poderá ser escalonado para CPU, SILBERSCHATZ (2000).

2.7.6 Escalonamento por Múltiplas Filas

O Algoritmo de escalonamento por múltiplas filas tem a característica de possuir várias filas e cada fila com prioridade diferente de outra. Os processos da primeira fila possuem prioridade absoluta em relação a outras, ou seja, quanto mais baixa a fila, menor a prioridade aos processos.

Este algoritmo de escalonamento é bastante útil quando se sabe com clareza as diferentes classes de processos.

Cada fila deste algoritmo pode possuir características diferentes. Por exemplo, a fila um pode ser o algoritmo de escalonamento RR, enquanto outras filas podem ser o algoritmo de escalonamento FCFS.

Veja-se um exemplo de algoritmo de escalonamento por múltiplas filas, na figura 2.8, segundo SILBERSCHATZ (2000). Há neste modelo 5 filas, ou seja, fila para processos do sistema, fila para processos interativos, fila para processos de edição interativa, fila para processos batch e fila para processos secundários.

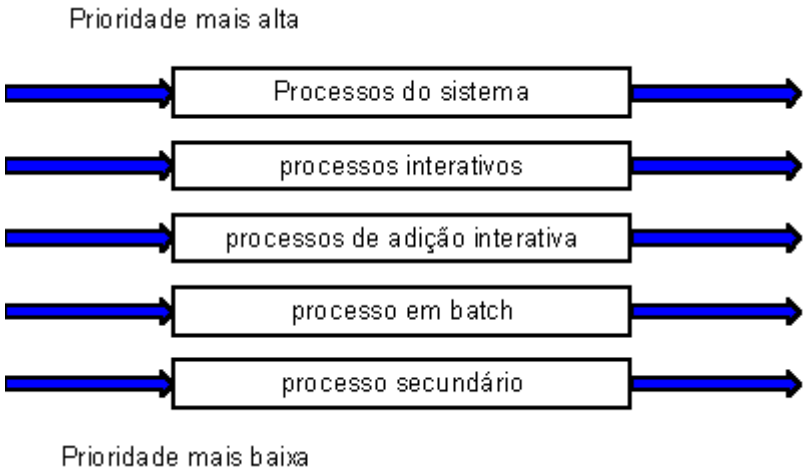


Figura 2.8 Escalonamento por Múltiplas Filas.

"*Cada fila tem prioridade absoluta sobre as filas de menor prioridade. Nenhum processo na fila batch, por exemplo, poderia executar, a menos que as filas para os processos do sistema, processos interativos e processos de edição interativa, estivessem todas vazias. Se um processo de edição interativa entrasse na fila de processos prontos enquanto um processo em bach estivesse executando, o processo em batch seria interrompido"*, SILBERSCHATZ (2000).

2.7.7 Escalonamento por Múltiplas Filas com Realimentação

Parecido com o algoritmo de escalonamento apresentado anteriormente, mas com característica peculiar de dinamismo, ou seja, os processos se alteram entre filas, de acordo com o comportamento na mudança de contexto. A idéia principal deste algoritmo é separar processos com diferentes características de surtos de CPU. Por exemplo, quando um processo irá usar excessivamente a CPU, executa um quantum na CPU e em seguida é reescalonado para uma fila de inferior prioridade. Pode ocorrer também de o processo ficar demasiado tempo na fila de espera. Uma das soluções de envelhecimento seria de encaminhar este processo para uma fila de alta prioridade.

Para cada fila é atribuída uma prioridade, a primeira é a mais alta, a segunda é de maior prioridade que as de baixo, assim por diante, caraterizando-se por possuir filas de alta prioridade, quantum de tempo menor que as de menores prioridade. A figura seguinte, figura 2.9, exemplifica o algoritmo de escalonamento com 3 filas, as duas

primeiras do tipo FIFO e a última do tipo Circular. Cada fila com quantum de tempo diferente, ou seja, 8, 16 e 32 respectivamente.

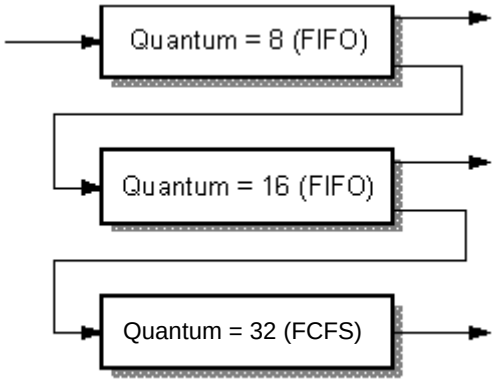


Figura 2.9 Escalonamento por Múltiplas Filas com Realimentação.

O algoritmo de escalonamento por múltiplas com realimentação possui as seguintes características:

- Todo processo criado entra na primeira fila, a fila de mais alta prioridade com um determinado quantum de execução;
- A segunda fila de processos prontos possui prioridade inferior à primeira fila de processos e maior prioridade em relação à segunda fila de processos prontos com quantum maior que a fila superior;
- Os processos da segunda fila de processos prontos somente serão escalonados para a CPU quando a fila superior estiver vazia, ou ainda, que seja beneficiada de política de resolução de problema de envelhecimento de processos na fila de prontos;
- Os processos somente voltarão para a mesma fila caso consumido na CPU o quantum de tempo a ele atribuído;
- Cada fila deve ter algoritmo de escalonamento diferente. Normalmente as primeiras são algoritmo de escalonamento FIFO e a última fila de processos prontos é o algoritmo de escalonamento RR (Circular).

Em geral, um escalonador por múltiplas com realimentação é definido pelos seguintes parâmetros:

- o número de filas;

- o algoritmo de escalonamento para cada fila;
- o método usado para determinar quando promover um processo para uma fila de maior prioridade;
- o método usado para determinar quando rebaixar um processo para uma fila de menor prioridade;
- o método usado para determinar em que fila um processo entrará quando esse processo precisar de serviço.

Este tipo de escalonamento atende a vários tipos de processos, mas vale salientar que seja um ótimo escalonador adaptável a qualquer tipo de sistema operacional, apresente algum tipo de deficiência, mas com uma boa técnica de envelhecimento de processos e uma boa administração de tempo aos processos, assim funcionará o sistema operacional em grande harmonia.

Podemos citar alguns dos sistemas operacionais que usam este algoritmo de escalonamento:

- os processos do sistema operacional VAX/VMS trabalham com 32 prioridades, em que 16 a 31 são processos de tempo real e os processos normais usam as primeiras 16 prioridades (0 a 15), SHAY (1996);
- os processos do sistema operacional UNIX também possuem prioridades (128 ao total), sendo 0 ao 49 processos do Kernel e o restante (50 a 127) são processos dos usuários.

Em ambos os casos, quanto menor o número maior é a prioridade.

2.7.8 Escalonamento por Tempo Real

Na atualidade, os sistemas operacionais de tempo real têm tomado corpo e grande importância em quase todos os meios em que a computação tem adentrado. Nos serviços de aeroportos, na robótica, na área médica, na indústria, etc. *"Um sistema de tempo real é usado quando existem requisitos rígidos de controle em uma aplicação dedicada"*, SILBERSCHATZ (2000).

Existem dois tipos de computação real:

- **Sistema de tempo real crítico**: a principal característica deste algoritmo escalonamento é a garantia na realização de uma tarefa crítica em um período de tempo limitado. A tarefa é submetida ao processador com tempo definido para concluir ou efetuar operação de entrada/saída. Exige que o escalonador saiba com precisão quanto tempo leva para conclusão de alguma tarefa, aceitando ou rejeitando o pedido.
- **Sistema de tempo real não-crítico**: requer que os processos críticos recebam prioridade em relação a outros menos favorecidos, pois pode haver alocações injustas e maiores atrasos para alguns processos.

2.7.9 Conclusão

Foram apresentados os principais algoritmos de escalonamento, alguns com tempo médio de espera calculados em pequenos exemplos e outros com destaque de alguns sistemas operacionais que fazem uso destes algoritmos. Há vários outros algoritmos:

- **Escalonamento em dois níveis:** O algoritmo de escalonamento em dois níveis move processos entre a memória principal e o disco magnético, TANENBAUM (1995).
- **Escalonamento garantido**: tenta controlar o total de tempo para cada usuário ao uso do processador, TANENBAUM (1995).
- **High Response Ratio Next (HRRN)**, OLIVEIRA (2000), STALLINGS (1998).
- **Shortest Remaining Time (SRT)**, OLIVEIRA (2000) e STALLINGS (1998),
- **Shortest Process Next (SPN)**, OLIVEIRA (2000) e STALLINGS (1998).
- Etc.

Como foi citado inicialmente neste capítulo, o algoritmo de escalonamento definido para o sistema operacional AURORA foi o algoritmo por múltiplas filas com

realimentação, aplicando técnica do problema de envelhecimento de processos na fila, a serem discutidos no capítulo 5.

A escolha por este algoritmo de escalonamento foi atribuída à sua característica dinâmica, pela portabilidade e distribuição "justa" de tempo a cada objeto que irá fazer uso do processador. Muitos autores têm comentado positivamente com relação a este algoritmo. Veja-se abaixo:

- Abraham SILBERSCHATZ (2000) afirma: "*A definição de escalonador por múltiplas filas com realimentação torna o algoritmo de escalonamento de CPU mais geral"*.
- William A. SHAY (1996): "*O sistema de filas múltiplas de realimentação é um método flexível que se adapta automaticamente à carga de trabalho e às alterações nos padrões de comportamento dos processos*".
- Francis B. MACHADO (1997) também afirma: "*O escalonamento por múltiplas filas com realimentação é um algoritmo de escalonamento generalista, podendo ser implementado em qualquer tipo de sistema operacional*".

3. Escalonamento em Sistemas Operacionais Tradicionais

Neste capítulo serão apresentados alguns sistemas operacionais com sua política de escalonamento. Os sistemas operacionais analisados demonstram que atualmente os principais sistemas utilizam muito o algoritmo de escalonamento de prioridade e de tempo real, mas no sistema operacional AURORA o Algoritmo de Escalonamento por Múltiplas Filas com Realimentação atinge as suas necessidades, como será demonstrado adiante.

3.1 Unix

Tecnologias importantes tiveram seu início por curiosidade ou simplesmente para suprir necessidades surgidas, mas que, ao final, funcionaram tão bem que acabaram sendo adotadas quase mundialmente. UNIX é um desses fenômenos, *"iniciado como um passatempo por um jovem pesquisador"*, TANENBAUM (2000), transformando-se em bilhões empreendidos em uso e desenvolvimento, envolvendo inúmeras entidades governamentais, educacionais e particulares.

"A primeira versão do UNIX foi desenvolvido em 1969 por Ken Thompson do Research Group da Bell Laboratories para usos em um PDP-7 que, de outro modo, ficaria ocioso. Dennis Richie logo se juntou a ele. Thompson, Richie e outros membros do Research Group produziram as primeiras versões do UNIX", SILBERSCHATZ (2000).

"Ao contrário do que você pode pensar, o UNIX não é um sistema operacional; ele é uma família de sistemas operacionais. Descrito pela primeira vez em 1974 por Thompson e Ritchie, tornou-se um dos sistemas mais difundidos", SHAY (1996).

Juntamente com as várias versões do sistema operacional UNIX, iam surgindo também linguagens de programação que serviram de berços para principais linguagens de programação atual. Inicialmente o UNIX foi escrito em Assembly, PL/I, posteriormente em linguagem de alto nível, o B, desenvolvido pelo próprio Thompson, oriunda da linguagem de programação BCPL, oriunda também de CPL. Finalmente, Richie desenvolveu a linguagem C, que teve uma grande importância no sistema operacional UNIX.

Evolução do UNIX:

- UNICS (UNIplexed Information and Computing Service) – reescreveu o MULTICS, em linguagem de montagem (Assembly), em um computador digital PDP-7 - (1969);
- Em 1972 Thompsom e Dennis Richie se unem e surge o UNIX, usando os computadores digitais PDP-11/20, PDP-11/45 e PDP-11/70, computadores estes de grande importância na época.
- Version 6 – padrão no mundo acadêmico – (1979);
- Version 7 – Uma versão comercial;
- System III e 4.2BSD – (1984);
- System V – (1981);

- SVR2 – (1984);
- 4.3BSD – (1985);
- SRV3, SunOS e DigitalUNIX – (1986);
- SRV4 – (1988);
- HP_UX, IBM_AIX, Solaris e SCO.

Baseados no UNIX, novos sistemas operacionais foram implementados, MINIX, pelo próprio Professor Andrew TANENBAUM (1980), e o LINUX, por um finlandês na década de 90, ambos com código fonte disponível.

3.1.1 Escalonamento em Unix (versão VAX do 4.3 BSD)

Com propósito de beneficiar processos interativos foi criado o escalonamento de CPU no UNIX.

Há três tipos de aplicações em sua política de escalonamento: **Interativa**, **batch** e **real-time**. A prioridade de processo varia dinamicamente, em que processos da mais alta prioridade tiram do processador, processo em execução, mesmo que não tenha terminado o seu tempo, colocando-o novamente na fila de espera.

Veja-se algumas características desta versão do UNIX:

- a cada processo é dada uma prioridade;
- as prioridades de escalonamento variam de 0 a 127, sendo: 0 a 49 para o kernel (processos do sistema), 50 a 127 para user-mode (processos do usuário).
- existem 32 filas de prioridade;
- quanto maior o número, menor a prioridade;
- processos com solicitações de ENTRADA/SAÍDA ou tarefas importantes possuem prioridades menores;
- processos do sistema têm maior prioridade que os do usuário;
- a quantidade de tempo na CPU está associada à prioridade, ou seja, quanto maior tempo para executar, menor prioridade;

3.1.2 Quantum de Execução

- As versões mais antigas do UNIX faziam quantum de 1 segundo para a política Round-Robin, enquanto o 4.3 BSD reescalona a cada 0,1 segundo, recalculando a prioridade para o processo.
- Dentro do kernel não há preempção de um processo por outro, isso quer dizer que o processo abandona a CPU pela solicitação de ENTRADA/SAÍDA ou por sua fatia de tempo ser estagnada, ou seja, não sai por motivo de algum processo com maior prioridade.

3.1.3 Problemas

Não escalona bem, tornando pesadas as mudanças de prioridades, não podendo também dar um quantum definitivo ao processo. E processos de alta prioridade, por vezes, têm que esperar muito tempo antes de executar.

3.1.4 Escalonamento em SVR4

O escalonador desta versão do sistema operacional UNIX objetiva suportar mais aplicações – incluindo o tempo real, separando as políticas de escalonamento dos mecanismos de escalonamento, adicionar novas políticas, etc.

Existem, neste algoritmo de escalonamento, 161 prioridades que vão de 0 a 160, em filas separadas, sendo:

- 0 a 59 para time-sharing;
- processos com prioridades menores adquirem mais tempo de processamento;
- usa event-driven sheduling: prioridade é alterada na resposta a eventos;
- tabela de prioridade inclui o quantum de execução;
- 60 a 99 para kernel;
- prioridade é determinada pela condição *sleep* (por exemplo, um processo que aguarda um evento de Entrada/Saída);
- 100 a 159 para tempo real;
- caracterizado por prioridade e quantuns fixos;

- processos exigem tempo de latência e tempo de resposta limitada.

Nesta versão, não há necessidade de recomputar todos os processos uma vez por segundo e configurar a tabela de prioridades.

3.2. Escalonamento no Windows

O sistema operacional Windows teve várias versões. Como na maioria de outros sistemas operacionais, a princípio usava-se apenas o DOS como sistema operacional. Nas evoluções surgiram várias versões para PC e redes de computadores.

3.2.1 Escalonamento no Windows 3.1 e Windows 3.11

Conhecido como multitarefa cooperativa, pertencente ainda ao escalonamento não-preemptivo, em que as aplicações verificam uma fila de mensagens periodicamente para detectar, se existem outras aplicações que necessitam fazer-se uso do CPU, enquanto isso, o processo fica na CPU. O grande problema é a possibilidade de um programa mal escrito tomar posse totalmente da CPU, impedindo que outros processos entrem em execução.

3.2.2 Escalonamento no Windows NT

O Microsoft Windows NT é um sistema Operacional de 32 bits preemptivos. Nesta classe de sistema para redes de computadores, houve várias alterações até as versões atuais (Windows 2000 e Windows XP), mas o grande início ocorreu com o Windows 3.11.

3.2.2.1 Histórico

Nos anos 80, a IBM e Microsoft tentaram trabalhar em conjunto, porém cada uma queria ter o seu próprio sistema operacional. Inicialmente, a cooperação gerou o sistema operacional OS/2, escrito em assembly. Desfeita a parceria, a Microsoft desenvolve, do ponto zero, a tecnologia NT.

3.2.2.2 Estrutura do Sistema

Boas partes do sistema operacional NT foram desenvolvidas em C, C++ e Assembly, com estrutura para cliente-servidor, dividindo processos (servidores), sendo serviço de memória, de arquivo, de escalonamento, etc.

O sistema operacional não é um sistema orientado a objetos, mas trata qualquer evento e recurso como tal.

3.2.2.3 Processo

Como mencionado no capítulo 1, Threads são tratados como processos, embora sejam parte de um processo, ocupando o mesmo endereçamento de memória que o processo de origem e fazendo-se uso de processamento assíncrono, como operações de entrada e saída. Os Processos multithread podem também ser utilizados para satisfazer concorrentemente solicitações de múltiplos clientes.

O objeto de escalonamento no Windows NT é o thread, que está associado a um processo, ou seja, threads são implementos como objetos, criados e eliminados pelo gerenciados de objetos. Desde a sua criação até a destruição, um thread passa por distintos estados de execução, como um processo normal.

3.2.2.4 Gerência do processador

A gerência do processador do Windows NT define a política de divisão do tempo da UCP entre os diversos threads, incluindo processos de usuários e do próprio sistema. O mecanismo de seleção do thread a ser executado é realizado por uma rotina do kernel denominado dispatcher.

Inicialmente, o thread recebe a prioridade do processo ao qual pertence, que pode ser alterada durante sua existência.

Normalmente no Windows de 32 bits há 32 prioridades, variando de 0 (o menor) a 31 (o maior), divididas em 2 faixas: Prioridade Variável (variable-priority) de 0 a 15 e Tempo Real (real-time) de 16 a 31. A prioridade 0 é um encadeamento especial do sistema denominado encadeamento de página zero. No win32 funciona a política de que se é ativado um processo de mais alta prioridade e, nesse momento, esteja um processo de menor prioridade sendo executado, o escalonador interrompe o atual processo permitindo que entre o de prioridade mais alta.

Os encadeamentos de processos são sempre feitos pela fila de número 31, ou seja, todos os processos entram com alta prioridade, sendo definida sua real importância ao chegar a sua vez de execução.

3.2.2.5 Escalonamento de Prioridade Variável

Um processo em execução, trabalhando na faixa de prioridade variável (entre 0 e 15) somente deixa a UCP caso ocorra uma destas situações:

- término de execução do thread;
- thread de maior prioridade entra em estado de pronto (preempção por prioridade);
- solicitação de um evento ou recursos do sistema;
- término da fatia de tempo (quantum end).

Para o escalonamento de prioridade variável, o NT trabalha com dois tipos de prioridades para o thread: base e dinâmica. Caso um thread seja grande consumidor de UCP (CPU-bound) ele tende a ter uma prioridade dinâmica menor que a de um outro que realize inúmeras operações de entrada e saída (I/O bound). Podemos entender, então, que a prioridade dinâmica é determinada através da soma da prioridade base mais o incremento recebido no momento em que o thread sai do estado de espera para o estado de pronto.

4.2.2.6 Escalonamento de Tempo Real

Um thread em execução, trabalhando na faixa de tempo real (entre 16 e 31), deixa a UCP apenas nos seguintes casos:

- término da execução do thread;
- thread de maior prioridade entra em estado de pronto (preempção por prioridade);
- solicitação de um evento ou recurso do sistema.

Os dois níveis de escalonamento apresentados permitem ao Windows NT oferecer características de sistemas de tempo compartilhado e de tempo real dentro do mesmo ambiente, tornando-o versátil e possibilitando seu emprego por diferentes tipos de aplicação.

O Windows 95, o Windows NT 3.51 e o NT 4.0, como também os mais recentes, têm todas diferentes variações do algoritmo de escalonamento.

3.3 Escalonamento no VMS

VMS escalona seguindo a lei de prioridade, ou seja, é um tipo de escalonamento parecido à fila múltipla de realimentação.

Cada processo é adicionado na fila de número 31, que é a fila-base e é a mais alta base do escalonamento. Após a primeira execução, o processo pode mudar de prioridade, por meio de um vetor de cabeçalho que definirá qual prioridade terá o processo.

As filas de 0 a 15, inclusive, são processos normais e os acima de 16 correspondem ao de tipo real, ou seja, se chegar na fila algum processo com prioridade mais alta do que se encontra no processador, o processo em execução é colocado em estado de espera para dar lugar ao recém-chegado.

3.4 Escalonamento em Sistemas Operacionais Orientados a Objetos

Como a arquitetura deste sistema é diferente dos sistemas existentes por se tratar de objetos e cada objeto possui seu meta-espaço, o escalonamento também pode ser diferenciado.

3.4.1 Escalonamento no Sistema Operacional Apertos

Visto que cada objeto possui sua meta-espaço, ***Apertos*** trata distintamente da política de escalonamento a objetos.

Cada meta-espaço possui um escalonador para os objetos que essa meta-espaço suporta. Esta estrutura cria uma hierarquia de escalonadores, uma vez que podem existir diversos escalonadores no sistema. Já que cada meta-espaço representa um ambiente computacional virtual, seu escalonador não consegue atender aos requisitos de “tempo-real”. Assim foi criado um objeto denominado “base policy”. Este objeto decide que algoritmo de escalonamento será usado, baseado em informações recebidas dos demais escalonadores distribuídos pelo meta-espaços, MIRANDA (2000).

3.4.2 Escalonamento no Sistema Operacional AURORA

Após análise de várias políticas de escalonamento e de alguns escalonadores de outros sistemas operacionais, foi possível chegar a uma política de escalonamento satisfatório que atenderá normalmente às necessidades do Aurora.

A grande maioria dos sistemas operacionais, principalmente os modernos, tem adotado algoritmo de escalonamento por tempo real, em que há uma tabela de regras para os processos que necessitam de urgência do CPU.

No AURORA, será utilizado o **Algoritmo por Múltiplas Filas com Realimentação**, ou seja, um algoritmo que atenderá às necessidades deste sistema operacional que está sendo implementado. Vale destacar que, além de fazer uso desse algoritmo existente, será tratado de uma forma especial o envelhecimento de objetos na fila, para que um objeto não fique em total estado de obsolescência, ou seja, nunca tendo oportunidade de fazer uso da CPU.

4. Sistemas Operacionais Orientados a Objetos

Uma rápida descrição sobre Reflexão Computacional, estrutura de um sistema operacional reflexivo, alguns sistemas operacionais orientados a objetos reflexivos e o sistema operacional AURORA.

Pressmam comenta sobre o impacto que a Programação Orientada a Objetos (POO) exercerá sobre futuras arquiteturas computacionais baseada nessa tecnologia que tem tomado força a partir de meados da década de 80. Alguns autores prevêem sistemas operacionais específicos a esta finalidade e é por causa disso que tem despertado muito interesse na comunidade acadêmica a Computação Orientada a Objetos.

O paradigma de orientação a objetos da engenharia de software tem demonstrado a grande importância de sistemas orientados a objetos. A década de 90 foi marcada com pesquisas e muitos projetos em andamento a busca de soluções a esta nova tecnologia, alguns em fase bem adiantada, pode-se citar como exemplo, sistemas operacionais (Apertos, 1991, CHOICES, 1992, Deubbler, Koestler, 1994, AURORA, 1997)

4.1 Reflexão Computacional

Reflexão Computacional é toda a atividade de um sistema computacional realizada sobre si mesma e de forma separada das computações em curso, com o objetivo de resolver seus próprios problemas e obter informações sobre suas computações, MAES (1987).

4.2 Arquitetura Reflexiva

A **reflexão computacional** (RC) define sua arquitetura em dois níveis: meta-nível (Meta-Core) e nível base (Meta-Objeto). Na meta-nível é onde se encontram as estruturas de dados, as ações e comportamentos dos objetos, localizados no nível base. Podemos afirmar que em RC existem dois subsistemas que interagem para a formação de um todo: o primeiro subsistema atua sobre um domínio externo ao sistema, enquanto o segundo atua sobre o próprio objeto do sistema. A RC é feita no meta-nível que pode interferir no nível base.

A figura 4.1 apresenta a modelagem do ambiente em termos do relacionamento entre o *meta-objeto* terminal, *Meta-Core*, e o modelo *objeto*/*meta-objeto*. Na figura 4.1, podemos visualizar como a interação entre objetos é realizada por inteermédio das primitivas de meta-computação implementadas. Sempre que um objeto (a) deseja enviar uma mensagem para outro objeto (b), o objeto (a) executa uma chamada ao *Meta-Core* por meio da primitiva M, provocando a transferência do controle da execução para um *meta-espaço* no meta-nível.

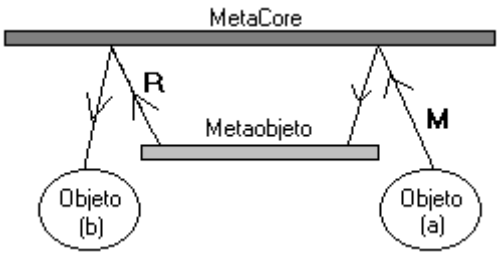


Figura 4.1 Interação entre objetos via *Meta-Core.*

Este modelo proporciona uma implementação por meio de um código compacto e otimizado, o que o torna de fácil entendimento e portabilidade. A interface da classe C++ modelada para implementação da classe *Meta-Core* é apresentada a seguir:

```
class MetaCore
{
     public:
            // primitivas para meta-computação
            void  M  ( MetaActivity* mObject,  MessageM* pMsg );
            void  R   ( MessageR* pMsg );
};
```

A função da primitiva **M** é transferir o controle da execução de um objeto para o meta-objeto. Significado da definição dos parâmetros da primitiva são mostrados a seguir:

Meta-Activity contém informações que identificam o meta-objeto no meta-nível.

MessageM contém informações sobre a mensagem a ser transferida e a identificação dos objetos origem e destino da mensagem.

```
struct structMessageM
{
     Activity*          source;     // Activity do objeto fonte
     Activity*          target;     // Activity do objeto destino
     void*              message;    // mensagem a enviar
}: MessageM;

struct structMetaActivity
{
      Activity*         self; // Activity do objeto em execução
}:MetaActivity;
```

A primitiva **R** tem a função de realizar o processamento inverso ao da primitiva **M,** isto é, retornar o controle da execução do meta-nível para o nível do objeto. Significado e definição do parâmetro da primitiva são mostrados abaixo:

MessageR, contém informações a respeito do objeto que enviou a mensagem e informações a respeito da mensagem enviada.

```
struct structMessageR
{
      Activity*         source;     // Activity do objeto fonte
      void*             message;    // mensagem a receber
}:MessageR;
```

O benefício imediato de se implementar unicamente suporte ao modelo conceitual no nível da *Meta-Core* é torná-lo independente das abstrações básicas do sistema, tais como objetos, *threads*, etc. A responsabilidade do conhecimento a respeito das abstrações conceituais é transferida para o nível da *meta-espaço* na meta-hierarquia, de modo que a *Meta-Core* nada conhece a respeito dos objetos que existem no sistema.

4.3 O Modelo de Estrutura Reflexiva de Aurora

O modelo de plataforma reflexivo orientado a objeto, multiprocessado, paralelo e utilizável em ambiente distribuído, foi proposto juntamente com AURORA por ZANCANELLA (1997).

Este modelo de estrutura reflexiva vem ao encontro da necessidade de criar um ambiente computacional, direcionando a orientação a objetos e é de se observar que os sistemas operacionais existentes não foram planejados para suportar as linguagens modernas.

O modelo computacional reflexivo é caracterizado por *objetos*, *meta-objetos* e *meta-espaços*, em que objetos são instância de uma classe, os quais contém todas as informações e características dos dados; enquanto Meta-objetos refletem o comportamento desses objetos. Cada objeto possui seu Meta-Espaço, que é a composição de mais meta-objetos do objeto, ZANCANELLA (1997).

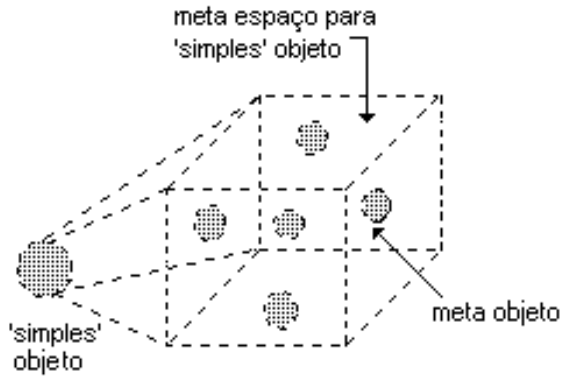


Figura 4.2 Um “simples objeto”

5. Algoritmo de Escalonamento de Objetos - AEO

A proposta para o escalonamento de objetos do sistema operacional AURORA é discutida e apresentada neste capítulo, bem como soluções a eventuais problemas de envelhecimento de objetos nas filas inferiores, podendo ocorrer que o objeto nunca venha a sair da fila de espera ou sendo prejudicado, demasiadamente, por esperar muito tempo na fila de prontos, ou seja, estes objetos estão nas filas de objetos com prioridades inferiores, causando retardamento à activity(objeto do AURORA) em espera da CPU.

5.1 Introdução

O Escalonador de Objetos por Múltiplas Filas com Realimentação foi adotado após estudos e análise de outros algoritmos, mediante informações citadas por renomados autores de livros de sistemas operacionais tradicionais e de sistemas operacionais existentes, bem como por artigos de sistemas operacionais reflexivos orientados a objetos. Muitos dos atuais sistemas operacionais, principalmente os dois sistemas operacionais reflexivos existentes (Apertos e Choice), usam a política de escalonamento de tempo real, que com certeza atinge o objetivo desses sistemas operacionais. Para a atual necessidade do sistema operacional Aurora, que ainda não há nenhum escalonador, o "**Algoritmo de Escalonamento por Múltiplas Filas com Realimentação**" alcançará suficientemente as necessidades para escalonamento dos objetos no sistema.

5.2 Linguagem de Programação e Modelagem de Dados

Como o sistema operacional é orientado a objetos e reflexivo haveria a necessidade de fazer uso de uma linguagem de programação orientada a objetos. A princípio se pleiteava o uso de Java, mas pelo motivo de todos os componentes do sistema operacional Aurora existentes estarem em C++, foi, então, adotada esta mesma linguagem de programação.

O ambiente e compilador utilizados foram um compilador compatível com ANSIC o Borland C++ Buider 6.0, por ser a mais moderna das versões dos compiladores e pelos ótimos recursos oferecidos por esta ferramenta. Levou-se em conta também a grande quantidade de literaturas acadêmicas e comerciais que tratam e trazem soluções a grandes variedades de problemas, considerando que sistemas operacionais importantes existentes no mercado foram implementados parcialmente e, até, totalmente nesta linguagem. A portabilidade e recurso oferecidos pela linguagem de programação C++ foram os estímulos para escolhê-lo como ferramenta para implementação deste escalonador.

A validação do modelo AEO foi implementado usando um processador Pentium III, 1 Gigahetz de velocidade e 512 MgBytes de Memória RAM, em sistema operacional Windows 2000.

A modelagem das classes do escalonador (AEO) foi realizada com o uso de UML.

5.3 Escalonamento de Objetos

Sempre que um novo objeto é criado ou um objeto inativo é ativado ele deve ser atribuído a um processador. Um novo objeto é usualmente atribuído ao processador no qual ele é criado. O sistema pode permitir que o objeto seja criado em um processador remoto. Um objeto que é reativado pode ser retribuído para qualquer processador do mesmo tipo do qual ele foi originalmente criado. A exceção é encontrada nos objetos imóveis, cuja localização é codificada dentro da identificação do objeto, isto é, sempre retribuído ao mesmo processador do qual se originou.

Um objeto durante toda sua existência pode possuir distintos comportamentos, desde o momento de sua ativação até a sua desativação do sistema. Alguns algoritmos de escalonamento tratam esses comportamentos indistintamente, ou seja, tempo igual para todos, ou uso total do processador por toda a existência do processo, porém nenhum usuário deve ficar em estado de total desfavorecimento, dando oportunidade justa à execução de cada objeto. Neste ponto torna-se fundamental um gerenciamento dinâmico dos processos, ou seja, no momento do surgimento destes comportamentos, aplicando soluções a cada estado de comportamento. Por exemplo: há objetos que farão somente cálculos matemáticos e testes lógicos, ficando pouco tempo com a UCP em poder, como também há objetos que irão solicitar do sistema Entrada/Saída constante ou momentaneamente.

Veja abaixo novamente as políticas de escalonamento que farão parte do escalonador para sistemas operacionais reflexivos orientados a objetos, ou seja, será a junção de escalonamento First-In-First-Out (FIFO) e escalonamento Round-Robin (Circular) que, juntos, farão parte do escalonamento com múltiplas filas com realimentação, que é objeto de estudo e implementação deste trabalho.

No escalonamento FIFO, algum objeto será prejudicado no uso da UCP, enquanto outros terão tempo excessivo ao processador. Na figura 5.1, há exemplo de três tarefas na fila. A tarefa C só terá o processador para execução após o término da tarefa B e, conseqüentemente, a tarefa B somente terá o processador após o término da

tarefa A. Na prática, três pequenas tarefas seriam imperceptíveis ao usuário, porém, em números maiores de tarefas e duração de surtos também maiores o usuário sentirá drasticamente um resultado negativo do desempenho do sistema operacional.

- **Escalonamento FIFO:** algum objeto será prejudicado no uso da UCP, enquanto outros terão tempo excessivo no processador.

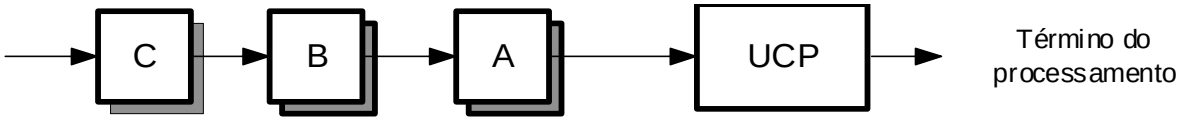


Figura 5.1 Escalonamento FIFO.

- **Escalonamento Circular** resolverá parcialmente o problema, com o uso de tempo compartilhado (quantum predefinido), mas, mesmo assim, ainda terá objetos em grandes desvantagens. É uma boa política de escalonamento e fácil de implementar, o único problema acarretaria em não diferenciar uma tarefa do outra, possibilitando privilégios ou prioridades. Na figura 5.2 segue o mesmo exemplo, três tarefas concorrendo à CPU com quantum de tempo igual para todas as tarefas, ou seja, suponha-se que as tarefas terão quantum de tempo igual a 10 milissegundos para cada vez que for à CPU; não terminando a tarefa nesta fatia de tempo, retornará à fila e esperará a sua vez. Para que a tarefa B tenha oportunidade de fazer uso da CPU deve a tarefa A consumir seu quantum de tempo no processador, valendo para a tarefa C em relação à tarefa B, e assim por diante.

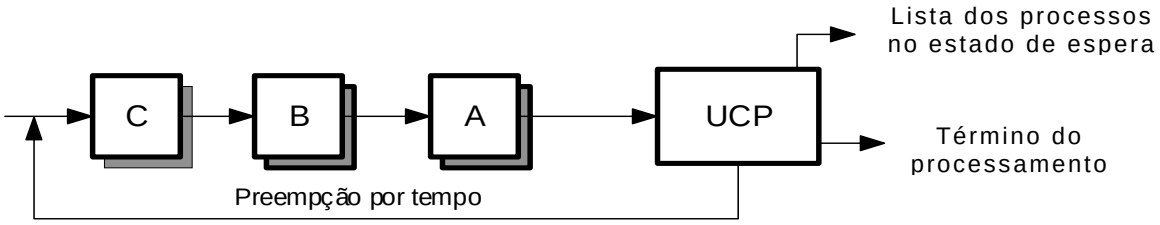


Figura 5.2 Escalonamento Circular

- **Escalonamento por múltiplas filas** apresenta-se como melhor para o controle destes objetos, em relação aos anteriores, já citados. Tal como foi comentada, esta política trabalha com várias filas, sendo a primeira de maior prioridade, tendo prioridades menores à medida que as filas ficam mais abaixo, caracterizando-se cada fila a processos com comportamento específico, por exemplo: podemos dividir os processos em três grupos: sistema, interativo e batch, MACHADO (1997). Como mostra a figura abaixo, o tipo de tarefa é associado ao tipo de fila, ou seja, cada tarefa é analisada antecipadamente e atribuída à fila correspondente. No exemplo abaixo, as tarefas do sistema sempre estarão na primeira fila, enquanto as tarefas intermediárias (interativas) e Bach nas inferiores, diferenciando uma fila da outra pela prioridade que as tarefas possuem, menor prioridade, mais abaixo é a fila de tarefas. SILBERSCHATZ (2000) exemplifica essas filas como sendo cinco:

  - Processos do sistema
  - Processos interativos
  - Processos de edição interativa
  - Processos em Bach
  - Processos secundários

  Cada fila possui prioridade alta em relação às inferiores, que são de menores prioridades. Por exemplo, as tarefas da fila de tarefas secundárias somente poderiam ser executadas se as filas de tarefas de cima estivessem vazias, SILBERSCHATZ (2000).

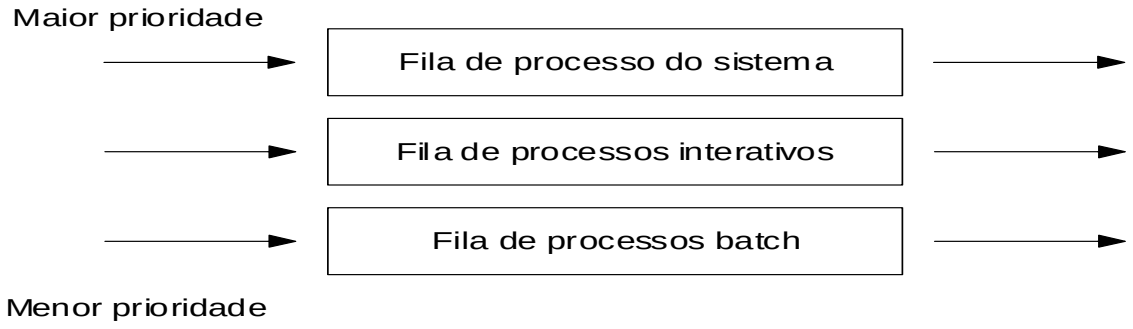


Figura 5.3 Escalonamento por Múltiplas Filas.

5.4 Algoritmo de Escalonamento por Múltiplas Filas com Realimentação

O união dessas três políticas citadas acima First-come, First-served (FIFO), Round-Robim (RR) e Múltiplas filas substancializam eficiência ao sistema operacional na implementação do escalonador. Baseado neste modelo de escalonamento, que será implementado o escalonador para o sistema operacional AURORA, o AEO.

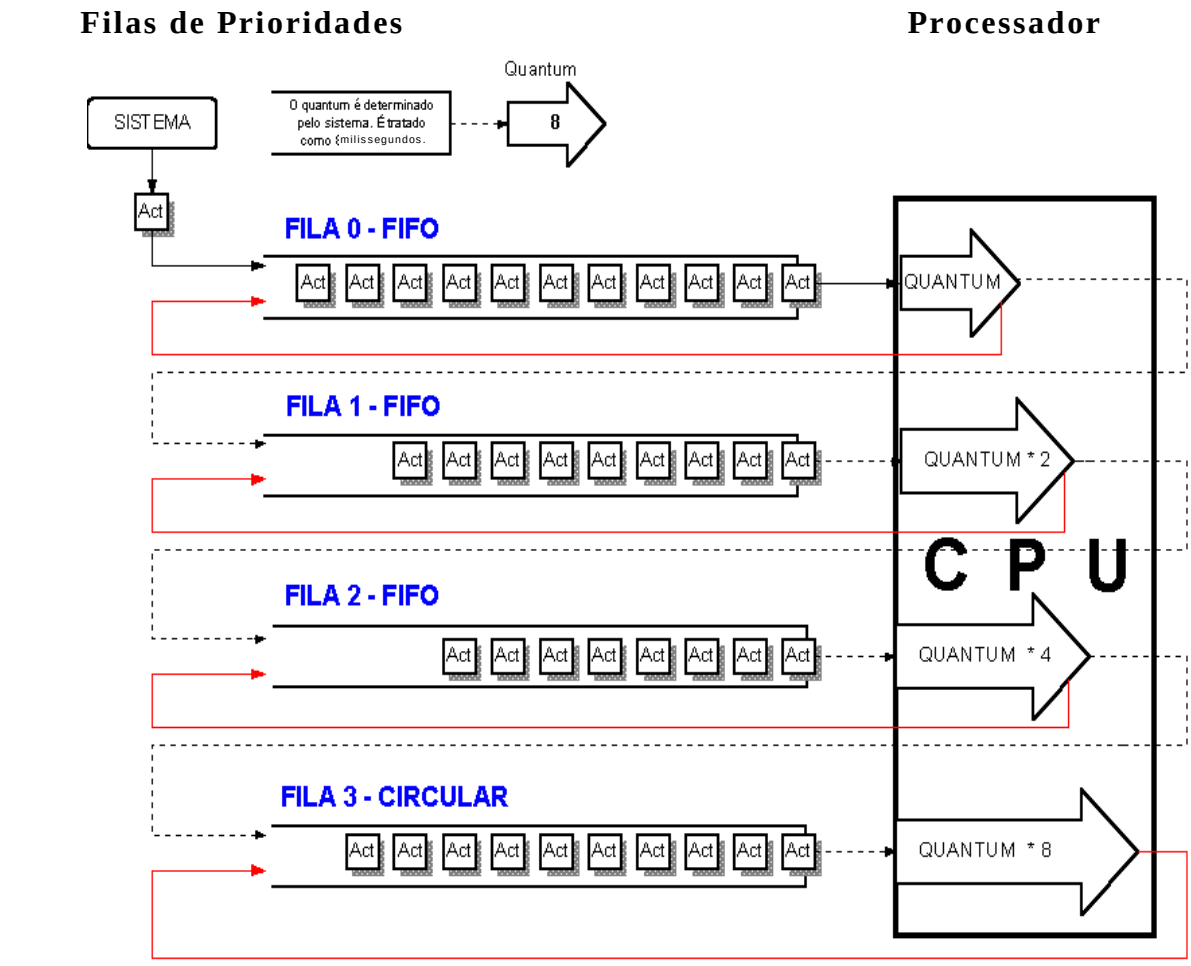
Esse algoritmo de escalonamento apresenta muitas características positivas, que neste projeto terão muita utilidade. Vejamos algumas vantagens:

- Todos os objetos ao entrar no escalonador terão alta prioridade;
- Há uma harmonia em uso de escalonamento FIFO e Circular;
- Pode ser implementado em qualquer tipo de sistema operacional;
- Os objetos terão tempo justo no uso da CPU.

As prioridades para os objetos neste tipo de escalonamento são dinâmicas, alternam-se de acordo com o comportamento do objeto em momento de execução, ou seja, tenta identificar estes comportamentos aplicando soluções a estes comportamentos. Porém o comportamento principal que se levará em conta, nesta implementação, será atribuído ao tempo de execução na CPU e o tempo executado. Com essas informações será possível inseri-lo na sua respectiva fila de objetos em estado de pronto.

Baseado na figura 5.4, pode-se entender como será o AEO. No Multi-level Feedback Queues (MFQ) há várias filas em que a primeira é a de maior prioridade com pequena fatia de tempo (quantum). Após o uso deste quantum de tempo, o objeto é redirecionado para a segunda fila (caso o objeto não tenha terminado de executar e tenha feito o uso total de tempo atribuído a ele para uso do processador) com quantum um pouco maior, porém, com menor prioridade. Outras filas abaixo terão menores prioridades que as superiores e maior fatia de tempo de processamento. E há nesta política de escalonamento o critério que a “fila 1” só será ativada caso não haja processo nenhum na fila anterior, que no caso é a fila 0 (zero), mas com este critério surge a possibilidade de processos em filas inferiores nunca terem oportunidade, ou tardar-se

demais a fazer uso do processador. Esses objetos envelhecidos na fila terão um tratamento especial que será descrito mais abaixo.



Caso o quantum (fatia de tempo) atribuído ao processo seja totalmente consumido no processador, irá para a fila de baixo.

Retornará à mesma fila, em caso de não-consumo do tempo recebido.

Figura 5.4 Escalonamento por Múltiplas Filas com Realimentação

As quantidades de filas poderão variar de 2 a 10 filas, mas por padrão terá um número fixo de filas, definido pelo administrador do sistema operacional AURORA. Neste modelo proposto haverá quatro filas, em que, as primeiras terão escalonamento FIFO e a quarta, escalonamento Circular. Objetos dessas filas possuirá prioridade diferente, aumentando o quantum de tempo a medida que à prioridade diminui.

Na validação do modelo cria-se possibilidade de escolher:

- A quantidade de filas de estado de pronto (2 a 10);
- Quantum de tempo para uso da CPU;
- Quantidade de objetos (1 a 50.000 Objetos);
- Objetos sendo criados dinamicamente;
- Pausa para visualização dos objetos sendo executados.

Mais detalhes serão apresentados adiante, que é a validação do modelo propriamente dito.

Todos os objetos iniciam-se com alta prioridade e entram na primeira fila. Haverá uso do processador por um quantum definido como 8 milissegundos, (definido pelo sistema, podendo ser maior ou menor, de acordo com a necessidade do sistema – veja-se o código: `typedef int quantum = 8;  //milissegundos`); usada esta fatia de tempo é reescalonado para a fila de baixo. Poderá também ser encaminhado para a mesma fila caso solicite Entrada/saída do sistema. Quando estiver na fila de baixo somente irá para o processador novamente se a fila de cima estiver vazia, ou seja, se não houver objeto algum, mas, para que o objeto não fique obsoleto na fila, será tratado o problema de objetos envelhecidos na fila.

A segunda fila terá prioridade inferior à primeira, porém com tempo maior para uso do processador (16 milissegundos), respeitando o mesmo critério da primeira fila, que uma vez consumido o tempo será encaminhado à fila de baixo, voltando à mesma fila caso não tenha consumido o quantum de tempo ou em outras situações.

A terceira com prioridade inferior a segunda fila, com quantum de 32 milissegundos; e a quarta, com quantum de 64.

A última fila é diferenciada das demais, posto que ficará o objeto até que se termine por completo sua execução, voltando sempre ao final da fila para ter direito novamente ao processador.

5.5. Armazenamento das Activitys

Como foi explanado anteriormente, todo objeto é inserido na primeira fila, cada fila (zero a três) é uma lista simplesmente encadeada, em que cada nó da lista é o endereço de uma activity.

Cada fila tem armazenado o seu primeiro endereço na posição de um vetor, depreendendo-se que poderá ter, até, 10 filas, pois o vetor possui 10 posições.

Em cada lista há dois ponteiros auxiliares: **Primeiro** e **Último**. Esses ponteiros auxiliares determinam o começo e o fim da lista, que é de grande utilidade para inserir e excluir activity, sendo que neste contexto, a inserção da activity será feita no final da lista, e a remoção, no início da lista. Veja a figura 5.5.

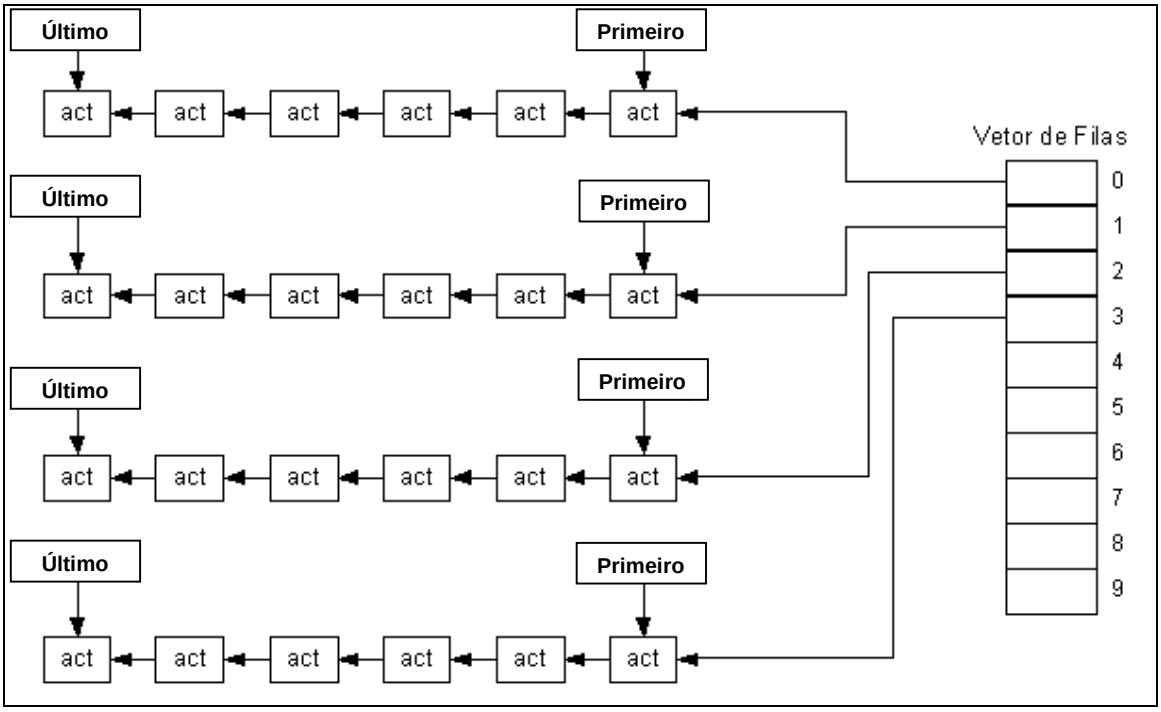


Figura 5.5 Estrutura de dados (Listas)

Em cada posição do vetor mencionado, não se armazenam apenas a lista e as variáveis auxiliares, primeiro e último, bem como também o número da fila, quantidade de objetos armazenados e o valor referente a objetos envelhecidos, para controle da segunda fila em diante.

Em resumo, o vetor armazena uma classe de nome **FilaLista**, como demonstra a figura 5.6.

```
class FilaLista{
     private:
          struct TipoCelula{
               Activity Item;
               struct TipoCelula *Prox;
          };
          TipoCelula *Primeiro;
          TipoCelula *Ultimo;
          int numero;
          int Quantidade;
          int Envelhecidos;

     public:
          FilaLista ( ); //construtor padrão
          int Vazia ( );
          int Cheia ( );
          void Excluir( );
          Activity Primeiro_No ( );
          void Inserir(Activity Elemento); //inserir um elemento na lista

          void NumeroDaFila (int);
          int  RetornaNumeroDaFila();

          void QuantidadeObjetosNaFila(int);
          int  RetornaQuantidadeObjetosNaFila( );

          void ObjetosEnvelhecidos(int);
          int  RetornaObjetosEnvelhecidos( );
};//fim classe TipoLista
```

Figura 5.6 Classe FilaLista

5.6 Activity

Uma *Activity* é similar ao modelo de *thread*, e inclui a abstração da CPU através de uma estrutura identificada como *context*.

"*Todo objeto instanciado é associado a uma activity. Tais objetos incluem objetos descritores de meta-espaços, meta-objetos do sistema e/ou da aplicação e objetos da aplicação. Activity é usada para representar os objetos em execução e contém a seguinte representação básica:*" ZANCANELLA (1997)

```
class Activity
{
      protected:
```

```
        CPUContext  Context;    // Estado dos registradores da CPU
        AID*        Identify;   // Identificação da Activity
        EntryTable* ExecQueue;  // Lista de Entradas (mensagens)
        pActivity   Meta;       // pointer para o meta-objeto

        Entry       Next;
        plongword   Address;
        MetaCore*   AuroraMetaCore;
        mcState     state;      // Estado da Activity

        Activity_AEO* AEO;
};
```

As informações associadas à *Activity* têm o seguinte significado e atribuições:

Context:	representa a abstração da CPU.
Identify:	identificação do objeto associado à activity.
Meta:	identificação da meta-espaço na meta-hierarquia.
Execqueue:	lista de ativações a serem executadas.
AEO:	**Algoritmo de Escalonamento de Objetos**

5.7 O objeto que será escalonado

Com finalidade de gerenciar tempo aos objetos que serão escalonados, foi criada a classe Activity_AEO que objetiva, no momento da instância do objeto:

- valor inteiro referente a quantum de tempo para fazer uso da CPU, sendo este valor: QT_serEXE = 8, ou seja, 8 milissegundos.
- valor inteiro referente a quantum de tempo que fez uso da CPU, sendo este valor: QT_foi_EXE = 0, ou seja, nada executou ainda.

Esses dois membros servem para controlar o tempo de uso da CPU, e também, o escalonamento e reescalonamento, conforme o caso. Os valores iniciais são atribuídos no momento da instância em função de que em cada activity instanciada é também instanciado uma **`Activity_AEO`**, como mostra o código abaixo:

```
Activity_AEO::Activity_AEO( )
{
     QT_serexe = quantum;     //Quantidade de Quantum para ser execução
     QT_foiEXE = 0;           //Quantidade de Quantum que foi executado
}
```

Uma vez ativada, os valores dos membros serão mudados de acordo com a execução da activity, ou seja:

- **QT_serEXE** será alterado ou mantido no momento em que estiver prestes a entrar em uma das filas;
- **QT_foiEXE** será alterado pelo sistema no momento em que estiver saindo da CPU, seja por término da fatia de tempo ou por solicitação de algum evento, e zerado quando estiver na fila.

Esses dois membros possuem grande importância no escalonador, pois as informações contidas neles serão determinantes para o destino da fila de espera em que o objeto será inserido. Para o controle de estado desses membros foram criados dois métodos:

- **void atribuir_QT_serEXE (int Q):** Quantum de tempo, em milissegundos, para o objeto;
- **int retornar_QT_foiEXE ( ):** Retorna quantum de tempo executado na CPU.

Ou seja, a **class Activity_AEO** foi implementada especialmente para o controle da activity. Veja o código abaixo:

```
class Activity_AEO
{
     protected:
            int   QT_serEXE;  //Quantidade de Quantum para ser execução
            int   QT_foiEXE;  //Quantidade de Quantum que foi executado

     public:
            Activity_AEO ( );
            ~Activity_AEO ( );
```

```
            void atribuir_QT_serEXE ( int Q);
            int  retornar_QT_foiEXE (      );
};
```

5.8 Controle das Activitys

Como demonstra a figura 5.5 (estrutura de dados das filas), as activitys devem ser inseridas em uma das filas, por escalonamento ou reescalonamento, e os objetos nas filas devem ser gerenciados para que nenhum fique estagnado, como também saber qual objeto deve ir ao processador. Veja como funciona este controle:

- Na **classe FilaLista** há como membro a fila, que armazenas os objetos, e outro membro do tipo inteiro, com propósito determinante de qual fila será extraída uma activity, lembrando que objetos da fila 1 (segunda fila) só serão escalonados caso o membro da fila 0 (zero) for igual a zero, ou seja, que esteja a fila 0 (primeira fila) vazia, ou que entre na característica de que há objetos envelhecidos na fila 1 (segunda fila);
- Há também um membro para cada uma das filas, a partir da segunda, para impedir que um objeto fique muito tempo na fila sem ter oportunidade de ir ao processador, levando em conta que sempre haverá objetos entrando nas filas acima. Cada um desses membros será testado para verificação de objetos envelhecidos e este cálculo será feito a partir de objetos inseridos nas filas, para que a cada objeto, ao entrar no escalonador, seja incrementada uma quantidade referente ao quantum de tempo para execução na CPU (o padrão será 8), ou seja, quando um dos membros possuir o valor $\boldsymbol{Quantum^2}$ deverá ser extraída da fila um objeto e enviado ao processador, retornando à mesma fila quando este objeto for reescalonado, caso termine o seu quantum de tempo, zerando também o valor do membro de controle da fila.

Observa-se a estrutura de controle das filas na figura 5.6 anteriormente.

5.9 Modelagem do Escalonamento

Para melhor compreensão do modelo do sistema serão vistos, nos próximos itens, alguns dos diagramas para representação das classes do sistema e com o auxílio da ferramenta de modelagem UML (Unified Modeling Language - Linguagem de Modelagem Unificada) foi possível descrever o modelo do escalonador. Os diagramas aqui utilizados foram **diagrama de casos de uso**, de **seqüência** e de **classe**.

5.9.1 Diagrama de Caso de Uso

Representa um conjunto de atores, caso de uso e os relacionamentos. Nesse modelo de sistema orientado a objeto, o único ator é o sistema operacional, que terá grande responsabilidade em:

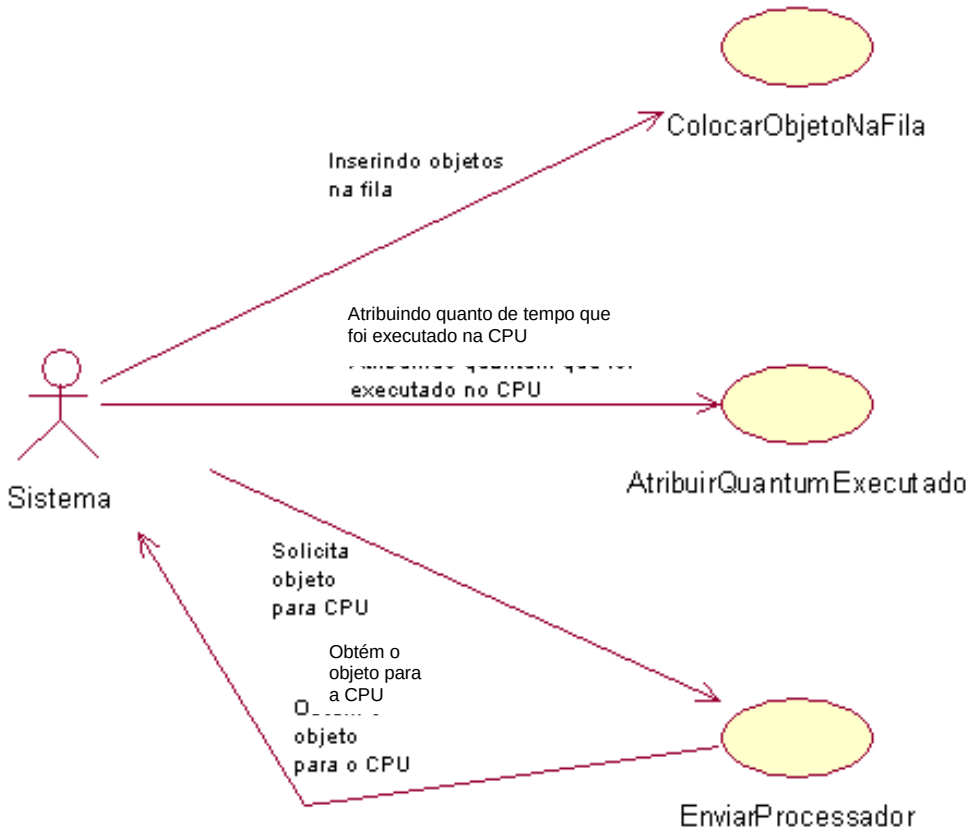


Figura 5.7 Diagrama de casos de uso

- **Encaminhar** o objeto ao escalonador, ou seja, chamar um método do escalonador que faça esse trabalho de inserir o objeto na fila **`void`**

`Carrega_Objetos_na_fila(Activity Act)`; esse encaminhamento pelo sistema operacional realiza-se após a ativação de uma activity ou quando uma activity tiver que ser reescalonada;

- **Atribuir** quantum de tempo à activity que será reescalonada, que acabou de sair do processador. Esta atribuição será feita pelo uso do método **`void atribuir_QQ_foiEXE ( int Q);`**
- **Solicitar** do escalonador uma activity para que seja levada à CPU, contido na activity fatia de tempo máximo (quantum) que poderá ficar sobre o poder da CPU.

5.9.2 Diagrama de Seqüência 1

O diagrama de seqüência dá ênfase às mensagens trocadas entre os objetos de um sistema. Essas mensagens são os serviços solicitados de um objeto ao outro. Veja a figura na página seguinte:

5.9.2.1 A Activity_AEO ( )

```
Activity_AEO::Activity_AEO( )
{
     QT_serEXE = quantum;   //Quantidade de Quantum para ser executado
     QT_foiEXE = 0;         //Quantidade de Quantum que foi executado
}
```

Quando uma nova activity é ativada recebe a fatia de tempo para executar que, por padrão, será de oito milissegundos, como também o tempo que foi executado, determinante para a fila que irá ser adicionada. Após a ativação terá outro tratamento, determinado quando entra no escalonador e quando sai da CPU.

5.9.2.2. Carrega_Objetos_na_Fila(Activity)

Método utilizado pelo sistema operacional para escalonar ou reescalonar uma activity, ou seja, é passado como argumento o endereço do objeto que será escalonado, com o quantum de tempo alterado, caso seja um reescalonamento.

1 Activity_AEO ()

```
Activity_AEO::Activity_AEO ( )
{
      QT_serEXE = QUANTUM;  //Quantum de tempo para ser executado
      QT_foiEXE = 0;        //Quantum de tempo que foi executado
}
```

Na ativação de uma nova activity, que é a instância de um novo objeto, receberá uma fatia de tempo para que seja executado no primeiro momento em que for encaminhado à CPU, o padrão será de oito milissegundos. Receberá também o tempo que foi executado, ou seja, zero. Estas atribuições são feitas pelo *construtor da* ***Activity_AEO()***, *dos alterados dinamicamente esses valores em momentos de execução do objeto.*

2 Carrega_Objetos_na_Fila (Activity)

Método utilizado pelo sistema operacional para encaminhar objetos para a fila de objetos em estado de pronto. Esse método é invocado após uma nova instância ou quando uma activity terminar sua fatia de tempo na CPU e necessite de ser reescalonado.

3 retornar_QT_foiEXE ( )

4 retornar_QT_serEXE ( )

Uma vez que o escalonador recebe o argumento (a activity), verifica-se a quantidade de quantum executado e a quantidade de quantum que deveria executar, independentemente de escalonamento ou reescalonamento. Com essas duas informações será possível definir qual das quatro filas deverá ficar.

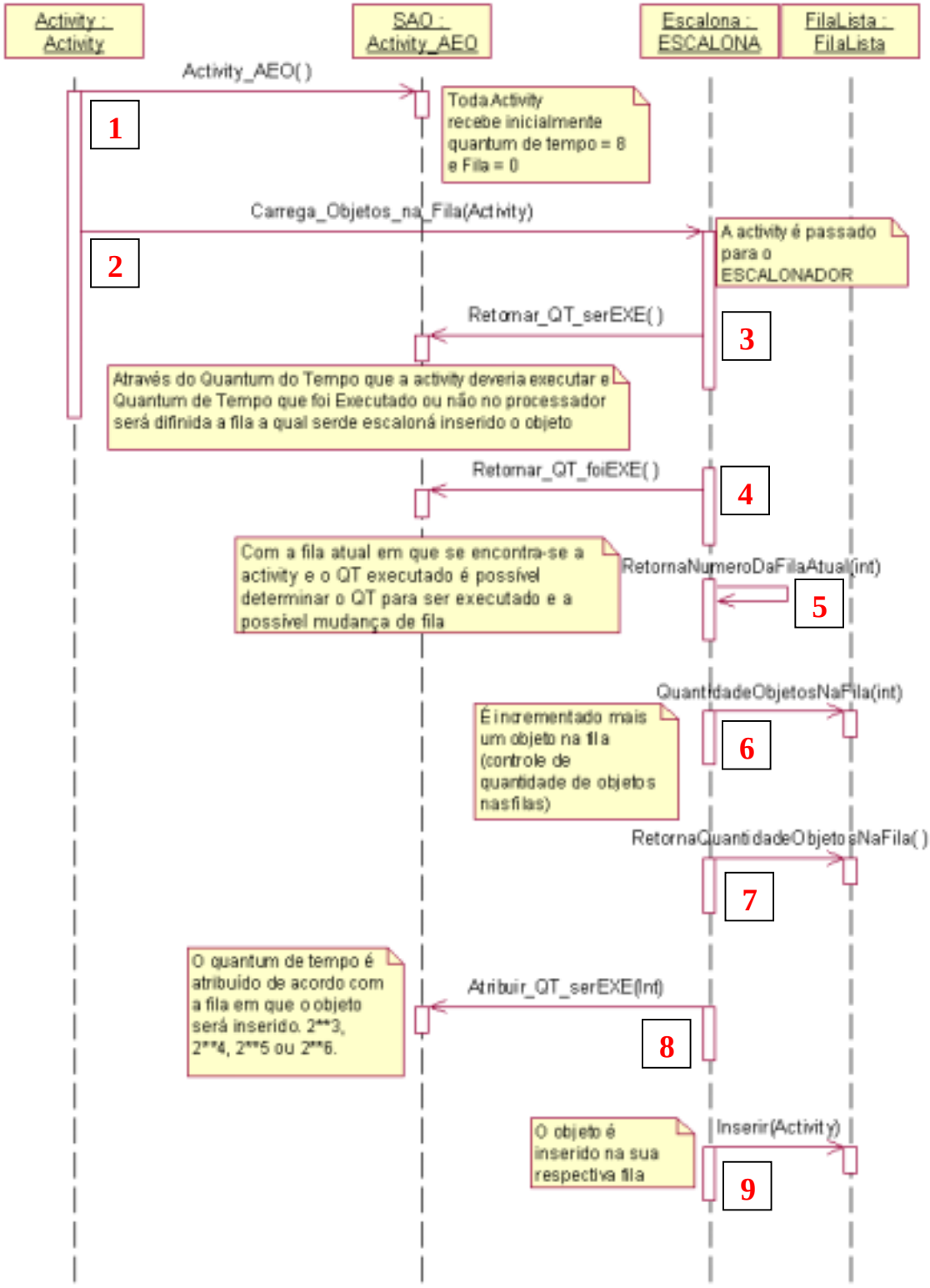


Figura 5.8 Diagrama de seqüência - Inserindo Activity no AEO

5 RetornaNumeroDaFilaAtual (int QsE)

Quando um objeto é inserido na fila, ou seja, quando está sendo inserido, é verificado em qual das filas este objeto se encontrava, mesmo que não tenha feito ainda uso da CPU possui uma fila atual, que é a fila 0 (zero). É passado como argumento por valor para o método o quantum de tempo a ser executado na CPU (QsE), mediante o retorno deste método é possível saber em que fila será inserida o objeto.

6 QuantidadeObjetosNaFila(int N)
7 RetornaQuantidadeObjetosNaFila ( )

É de fundamental importância saber a quantidade de objetos que há na fila na qual será inserida o novo objeto, a este dado será incrementado mais um, servindo sempre de controle para inserção ou exclusão de objetos. Por meio dessa quantidade de dados é que definirá se da fila abaixo poderá ser abstraída um objeto para a CPU, quando solicitado pelo sistema operacional.

8 atribuir_QT_serEXE (int QT)

Chegando a este ponto já se sabe em que fila o objeto será inserido, sendo possível definir o quantum de tempo (QT) em que o objeto terá que executar quando fizer uso da CPU. Esse quantum de tempo é passado como argumento e atribuído a um dos membros da classe activity, a qual servirá de controle para o sistema operacional AURORA.

9 Inserir(Activity Elemento)

Finalmente, o objeto, com a fila definida e o quantum de tempo para ser executado atualizado, poderá ir, então, para a respectiva fila, que é uma lista simplesmente encadeada circular.

5.9.3 Diagrama de Seqüência 2

A figura a seguir descreve activity solicitada pelo sistema operacional do escalonador, para ser enviado ao processador. Veja a figura descrita abaixo e alguns rápidos comentários sobre os itens numerados:

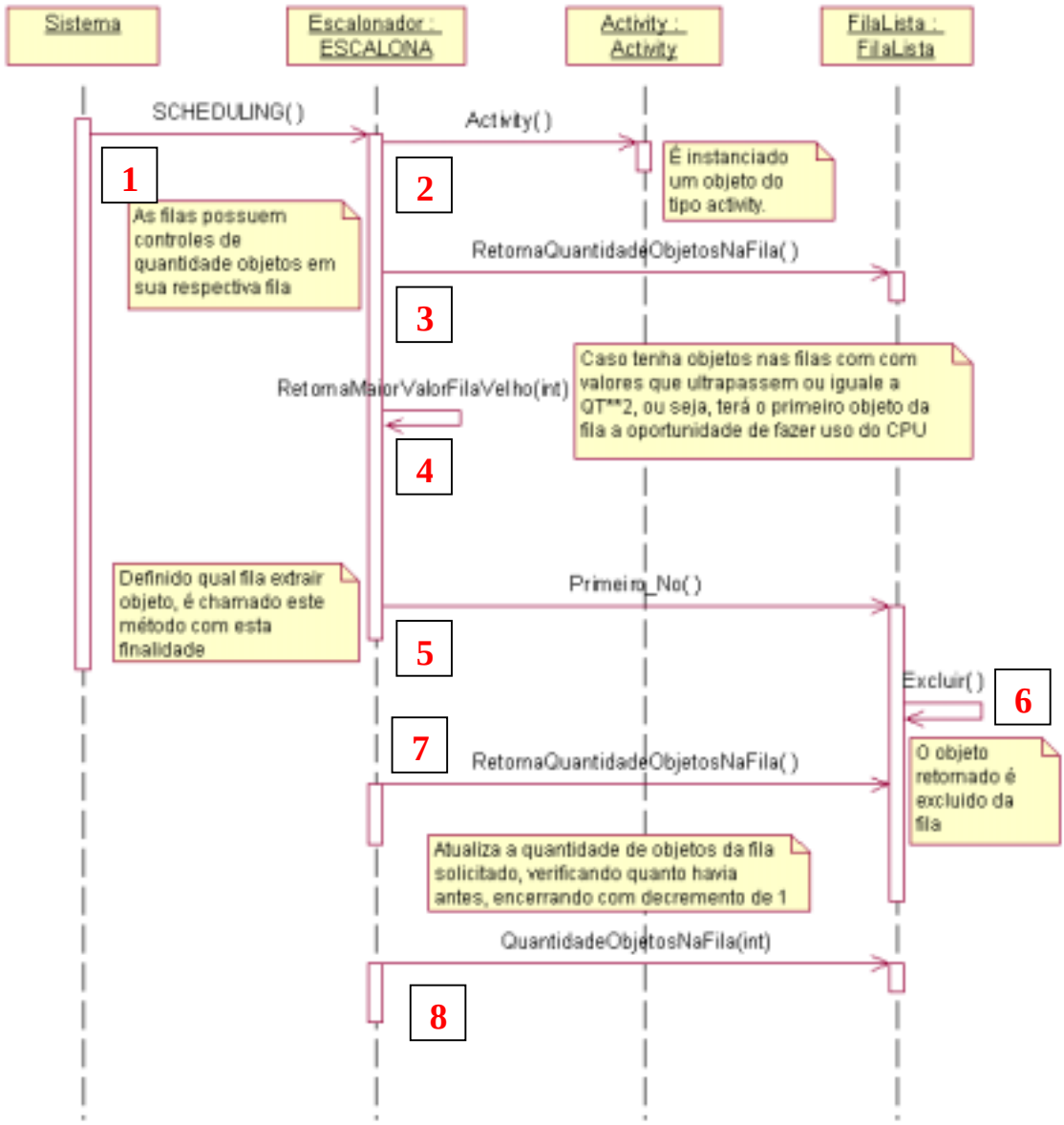


Figura 5.9 Diagrama de seqüência - Obtendo Activity do AEO para o CPU

1 SCHEDULING( )

Como foi comentado no início deste capítulo, o ideal é que a CPU não fique ociosa por instante algum. Deste ponto em diante é descrito como o sistema seleciona o objeto para fazer uso da CPU. As seleções deste objeto são realizadas pelo ESCALONADOR, ou seja, o sistema operacional faz a invocação do método **SCHEDULING (),** que é um dos métodos do escalonador, tornando-se um dos mais importantes do sistema operacional Aurora.

A invocação deste método pelo sistema operacional tem como retorno um objeto que se encontra em uma das filas de estado de pronto.

2 Activity( )

Uma activity é instanciada no momento da invocação do método **SCHEDULING(),** com propósito único de retorná-lo ao sistema.

3 RetornaQuantidadeObjetosNaFila ( )

Este método tem por finalidade única saber a quantidade de objetos que possui a fila que está sendo manipulada, pois com esta informação é possível incrementar ou decrementar um, como também buscar por objetos em filas inferiores.

4 RetornaMaiorValorFilaVelho (int IndiceMatriz)

Antes de testar se há objetos envelhecidos em filas abaixo é necessário verificar se na primeira fila há algum objeto para ser processado. Caso a fila zero esteja vazia, o teste é feito na fila um, testando-a da mesma forma que a anterior, ou seja, os testes são feitos até que cheguem à última fila. Com estas duas funções-membros pode o escalonador obter o tempo de objetos ociosos nas filas, para com estes dados tomar decisão para solução destes problemas.

5 **`Primeiro_No (Activity)`**

Uma vez definida a fila da qual será extraído o objeto, o sistema operacional terá como retorno a activity que fará uso da CPU.

6 **`Excluir( )`**

Com o objeto atribuído ao sistema operacional não mais há necessidade de que ele fique na fila, pois é excluído.

7 **`RetornaQuantidadeObjetosNaFila ( )`**

Pela fila da qual será extraído o objeto envelhecido, há uma variável de controle de tempo do objeto na fila, citado acima, independentemente do tempo que contém na variável-membro, é inicializada com o valor 0 (zero).

8 **`QuantidadeObjetosNaFila (int IndiceMatriz)`**

Estes dois métodos juntos servem para levar o conhecimento da quantidade de objetos na fila e decrementar 1 (um) do controle de número total de objetos na fila.

5.9.4 Diagrama de Classes do Algoritmo de Escalonamento de Objetos

Diagrama de classes representa a estrutura de um sistema e aqui se apresenta a estrutura do escalonamento de objeto.

Há 3 (três) classes e 2 (duas) estruturas neste sistema, veja abaixo:

- **Activity_AEO ( )**: Classe com propósito específico para controle comportamental de toda activity ativada e reescalonada. Na própria activity será instanciado um objeto do tipo **Activity_AEO** e, do começo ao final, haverá mudança na activity para o escalonamento e reescalonamento. Quando

instanciada a activity é instanciada também uma **Activity_AEO**, que no construtor inicializa o tempo para executar e o tempo executado da activity.

Os atributos desta classe caracterizarão atualização e a pesquisa de quantum de tempo a ser executado e quantum de tempo que foi executado pelo objeto.

- **Fila_Lista ( ):** Junto da classe há a estrutura de cada nó da lista (**TipoCelula ( )**), sendo o tipo de nó (activity) e um campo que apontará para o próximo nó.

As principais atividades desta classe serão: incluir, excluir e obter objeto do primeiro nó da fila, quando solicitado pelo sistema operacional.

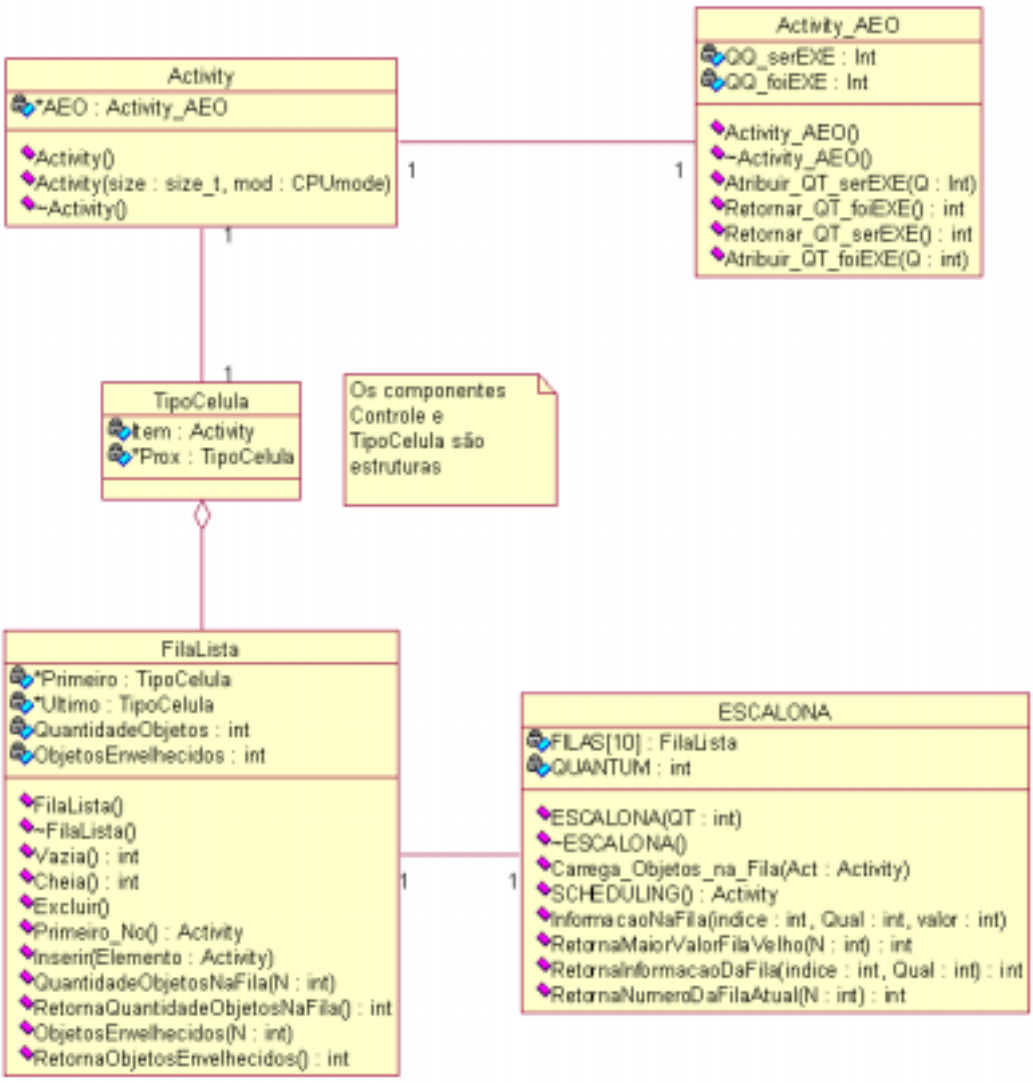


Figura 5.10 Diagrama de classes

- **ESCALONA ( ):** É a mais importante de todas as classes, ou seja, é desta classe que o sistema fará uso do método **Carrega_Objetos_na_Fila(Activity),** quando desejar inserir uma nova activity para ser escalonada, e também o método **&SCHEDULING( ),** para solicitar objetos da fila para fazer uso do processador. E, junto da classe, há também uma estrutura **Controle ( )**, de objetos na fila e de objetos envelhecidos.

5.10 Validação do Algoritmo de Escalonamento de Objetos

Para validação do modelo foi implementado um simulador para demonstrar uma das características mais importantes do ***"Algoritmo de Escalonamento por Múltiplas Filas com Realimentação"***, a qual é a distribuição "justa" de tempo e uso da CPU às tarefas, ou seja, uma tarefa que solicitará ENTRADA/SAÍDA fará pouco uso da sua fatia de tempo, necessitando ir à CPU com mais freqüência.

5.10.1 Tipos de tarefas

As tarefas podem ser classificadas de acordo com o comportamento e processamento, inclusive na mudança de contexto da tarefa MACHADO (1997). Seguindo este princípio, podemos definir as tarefas em dois tipos:

- **CPU-bound**: Diz-se de CPU-bound a uma tarefa que passa a maior parte do tempo em estado de execução, ou seja, permanece a maior parte de seu tempo na CPU. Esta tarefa possui características de quase não solicitar entrada/saída de dados, fazendo muitas operações matemáticas, científicas e lógicas e pouca escrita e leitura de dados (solicitação de I/O).
- **I/O-bound**: Diz-se de I/O-bound a uma tarefa que passa a maior parte do tempo no estado de espera, pois constantemente solicita leitura e escrita de dados.

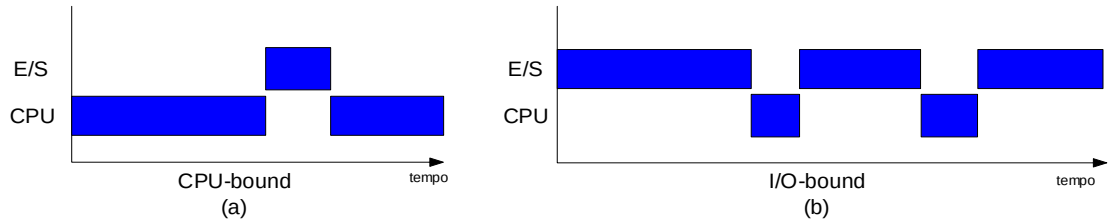


Figura 5.11 PU-bound X I/O-bound, MACHADO (1997)

5.10.2 Validação

AEO, um algoritmo de escalonamento para o sistema operacional AURORA, é uma implementação para suportar o modelo de objetos no sistema operacional AURORA, sendo que este modelo de objetos é o mesmo que o conceito de thead, ou seja, segue o mesmo princípio dos dois tipos de tarefas.

A validação foi feita com o escalonador proposto neste trabalho, acrescentando um ambiente para simular alguns objetos no sistema, em que a quantidade de objetos e outros atributos são definidos pelo usuário.

Veja as características da validação:

- Uma classe de nome activity foi definida, pois serão activitys que estarão compondo as filas do escalonador. Nesta classe é definido um vetor de oito posições, com propósito de armazenar os contextos da activity, claro, isto é apenas para a validação do modelo, pois os controles desses contextos serão feitos pelo sistema operacional. Também foi criado um outro vetor para armazenamento de valores inteiros, neste exemplo serão armazenados valores de 0 a 90. Quando a activity é instanciada, é atribuídos a todas as posições do vetor valores numérico inteiro 91, isso com finalidade única de determinar final de dados, que é, no máximo, 300 valores permitidos armazenados no vetor, correspondendo a surto máximo de execução. Essa quantidade de valores é definida aleatoriamente, tanto a quantidade de valores diferentes de 91 como também o valor contido em cada posição, que é abaixo de 91.
- Quando encontrados valores menores ou iguais a qual o usuário determinou, iniciando de zero, dependendo da quantidade de Entrada/Saída de dados, isso será mais bem explicado adiante.

- Cada posição do vetor de informação representa um quantum de um milissegundos, que é o tempo que terá a activity de fazer uso da CPU;
- Na função principal serão instanciados 5 objetos do tipo activity, com todas as informações já definidas, podendo alterar de acordo com o comportamento do objeto;
- Os tamanhos dos objetos são criados aleatoriamente, para que haja diferenciação de tamanho, e o tamanho máximo é de 300, como comentado acima;
- As filas de objetos são criadas segundo a necessidade definida pela criação de objetos e destruídos automaticamente ao término dos objetos, ou seja, ao destruir o último nó da lista.

5.10.3 VALIDAÇÃO DO MODELO

A validação do modelo foi dividida em três partes distintas. O formulário do próprio C++ Builder 6.0, AEO (classe ESCALONADOR) e Activity (classe ACTIVITY).

- O formulário principal objetiva simular as activitys sendo inseridas nas filas, solicitadas da fila e executadas na CPU. O formulário gerencia o escalonador e as activitys, que são os objetos, como se fosse o próprio sistema operacional AURORA.
- A classe de nome ESCALONADOR, que é o modelo para escalonar os objetos no sistema operacional AURORA, é a segunda camada desta validação. Esta classe é exatamente como funcionará no sistema operacional AURORA, não usando qualquer recurso do ambiente da programação, e sim, as instruções da linguagem C++.
- A classe de nome ACTIVITY gerará as informações e atributos necessários a cada objeto instanciado no sistema.

Veja a figura abaixo:

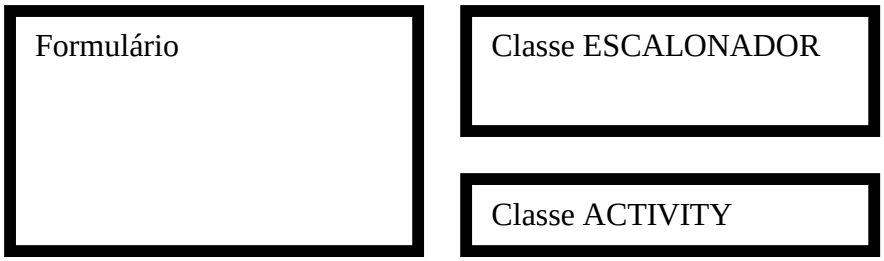


Figura 5.12 Camadas de validação do modelo AEO

5.10.3.1 O simulador

A partir deste ponto será explicado o modelo em seu real funcionamento. A quantidade de objetos instanciados dependerá da capacidade de processamento do computador em que estiver sendo executado o simulador, porém, quanto maior a quantidade de objetos instanciados, maior será a demora de processamento. O simulador está apto a executar 50.000 objetos, com criação de objetos dinâmicos da mesma quantidade, ou seja, terminando ao final com 100.000 objetos executados, porém, não faz sentido tanto objeto instanciado simultaneamente em uma aplicação comum.

Na figura a seguir, é mostrado o formulário citado, cujas páginas explicarão a funcionalidade de cada componente deste formulário, isto é, da quantidade de objetos instanciados ao relatório final destes objetos executados.

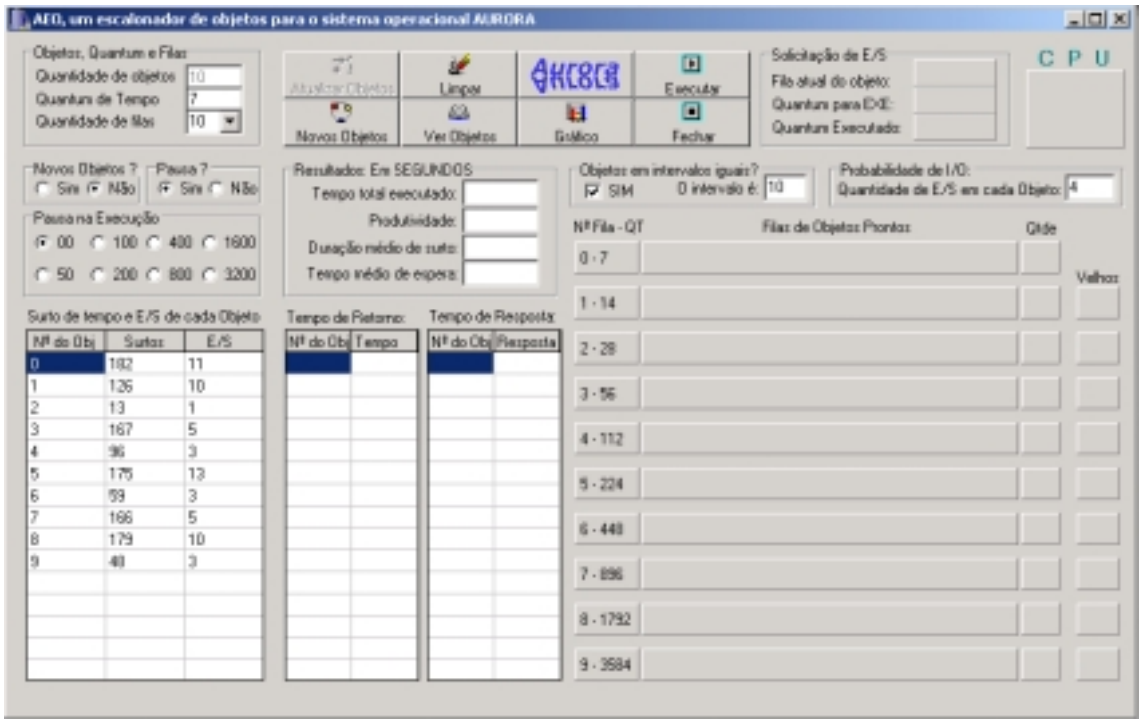


Figura 5.13 Formulário do simulador.

Para o bom funcionamento do escalonador, é de fundamental importância que seja preenchido corretamente cada campo disponível a ser inserido valores, valores estes que têm sua própria finalidade. Veja abaixo como funciona o simulador.

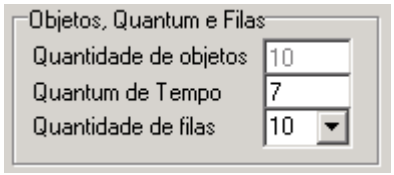


Figura 5.14 Quantidade de objetos, Quantum de tempo e Quantidade de filas.

Quantidade de Objetos: Por default a quantidade de objetos vem já definida como 10, porém se pode alterar esta quantidade a escolha do usuário, o limite de objetos está relacionado ao tipo de configuração do computador em que está sendo executado o simulador, segundo comentário acima. Caso o usuário opte por criar novos objetos em tempo de execução, será possível que seja criado até o dobro da quantidade escolhida inicialmente. A figura seguinte mostra como se pode optar para que isso ocorra.

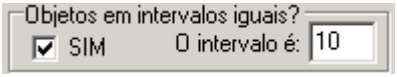


Figura 5.15 Objetos dinamicamente.

Caso esteja selecionada a opção “SIM”, e um valor determinado ao lado, na figura acima é 10, ou seja, será criado um novo objeto a cada 10 objetos solicitados ao escalonador, pelo sistema operacional, para que use a CPU. No final da execução do simulador, a quantidade de objetos escolhida pelo usuário inicialmente estará alterada, veja a figura abaixo:

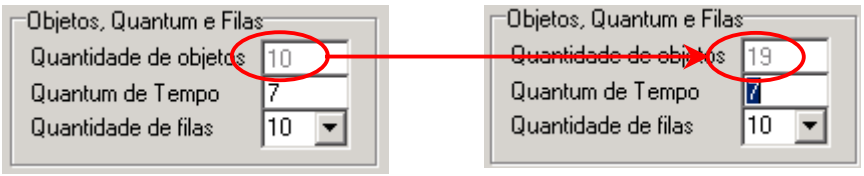


Figura 5.16 Novos Objetos dinamicamente.

Quantum de Tempo: É o tempo máximo que cada objeto terá para fazer uso da CPU quando tiver sua vez, isso é válido caso o objeto tenha saído da fila 0 (a primeira fila). As filas inferiores sempre serão o dobro da anterior.

Quantidade de Filas: Como o algoritmo de escalonamento é por múltiplas filas com realimentação, haverá, no mínimo, 2 filas. O máximo permitido é no total de 10 filas, de 0 a 9.

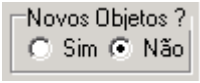


Figura 5.17 Instância de novos objetos.

Novos Objetos: Ao optar por “Sim”, são destruídos todos os objetos instanciados anteriormente, caso haja esses objetos. São destruídos também vários outros componentes que apresentam as informações desses objetos, é claro que normalmente não estará habilitado para que o usuário escolha uma dessas opções, já que somente será habilitado quando for executado o aplicativo ou quando se atualizar objetos.

O aplicativo é executado ao clicar sobre o botão “Executar”. Veja a próxima figura:

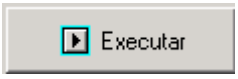


Figura 5.18 Botão executar.

Após executados todos os objetos, o sistema fica apto a novos processamentos, habilitando o botão “Atualizar objetos” e desabilitando o botão “Executar”.

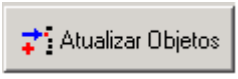


Figura 5.19 Botão Atualizar objetos

A atualização dos objetos é feita de acordo com a escolha do usuário, corresponde à instância dos objetos, à criação das filas, a um componente **StringGrid** para armazenamento do número do objeto, a seu tamanho e à quantidade de solicitação Entrada/Saída.

Os componentes StringGrid para “Tempo de Retorno” e “Tempo de Resposta” são criados em branco e com limite para o dobro de quantidade de objetos escolhidos pelo usuário. As linhas estarão em branco antes do aplicativo ser executado, sendo preenchidos com os resultados logo após o término do último objeto contido no escalonador a ser executado.

Figura 5.20 Relatório de cada objeto de ainda sem informação.

Na figura 5.21 ilustra o número da fila juntamente com o quantum de tempo que objetos da respectiva fila terão para fazer uso da CPU, a fila que terá o número dos objetos os quais se encontram na fila, a quantidade de objetos existentes na fila e o controle de objetos envelhecidos na fila, que é Quantum^2.

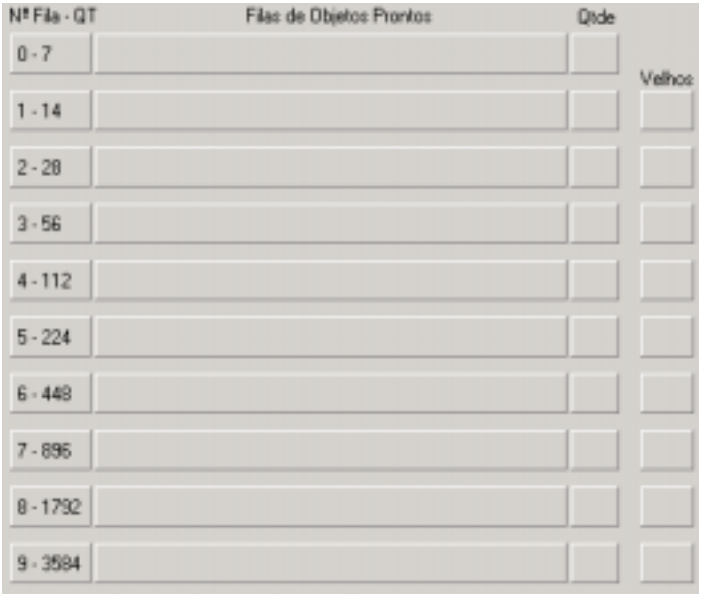


Figura 5.21 Número da fila, Quantum de tempo, Filas, Quantidade de Objetos na fila e Controle de Objetos envelhecidos.

O simulador permite que o usuário opte por visualizar os objetos em movimento entre as filas e processador. Para isso o usuário deverá marcar “sim”, com esta opção não sofrerá nenhum retardamento do surto atribuído ao objeto no momento da sua instância, considerando que este surto não é medido com tempo do sistema, e sim, a quantidade de informação contida no vetor de cada objeto.

Além de optar por “sim”, deve também o usuário escolher o tempo em milissegundos, que variam de 100 a 3200. Esta pausa acontecerá cada vez que o objeto sair do processador, quanto maior o número, maior será o tempo de execução ao total de todos os objetos. Dependendo da quantidade de objetos, podem ficar o sistema mais de 3 horas em execução, perdendo o sentido de avaliar e ver os objetos sendo executados.

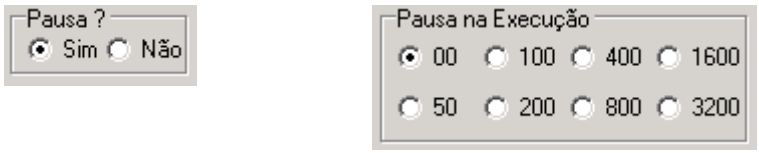


Figura 5.22 Pausa de execução.

A política padrão do “algoritmo de escalonamento por múltiplas filas com realimentação” é a de que o objeto somente poderá mudar de fila caso utilize por inteiro o tempo que lhe cabe para fazer uso da CPU. Caso o objeto solicite uma Entrada/Saída antes do término do seu quantum de tempo, deverá voltar para a mesma fila. Assim o usuário poderá optar em maior ou menor número de Entrada/Saída para os objetos, neste exemplo, o usuário solicitou 4, e isso quer dizer que os valores de 0 a 3 do vetor de informação do objeto será considerado como Entrada/saída de dados.

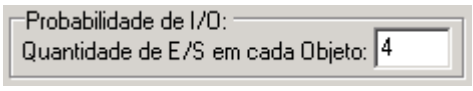


Figura 5.23 Pausa de execução.

Quando é solicitado pelo sistema operacional ao escalonador um objeto para fazer uso da CPU, as informações contidas no objeto terão uma grande importância tanto ao sistema operacional como ao escalonador. O sistema operacional deverá saber o quantum de tempo que o objeto terá para executar, após a saída do objeto da CPU. Com atualização do quantum de tempo no objeto, atualizado pelo sistema operacional, é possível saber em qual das filas este mesmo objeto será escalonado, isto se voltar a concorrer pelo uso da CPU, pois poderá entrar no estado de término.

O simulador apresentará em momento de execução o número do objeto, descrito acima do painel CPU, em cor vermelha, a fila na qual se encontrava, o quantum de tempo permitido para fazer uso da CPU e o quantum de tempo executado. Veja a figura 5.24.

Figura 5.24 Objeto em execução.

Depois de todos os campos preenchidos e clicados no botão “Novos Objetos”, são criados dinamicamente as filas, os objetos com seus respectivos tamanhos, a quantidade de Entrada/Saída de dados de cada objeto, bem como a criação de novos objetos, também criados dinamicamente. No final, após a execução do último objeto, será gerado um relatório com as informações de tempo total executado, produtividade, duração média de surto e tempo médio de espera dos objetos na fila. Veja a figura 5.25.

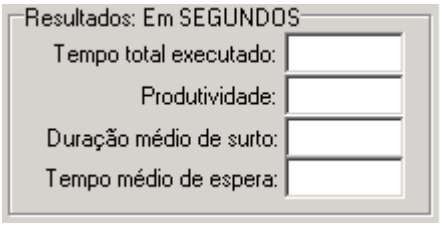


Figura 5.25 Relatório dos objetos processados.

5.10.3.2 Primeiro exemplo com poucos objetos

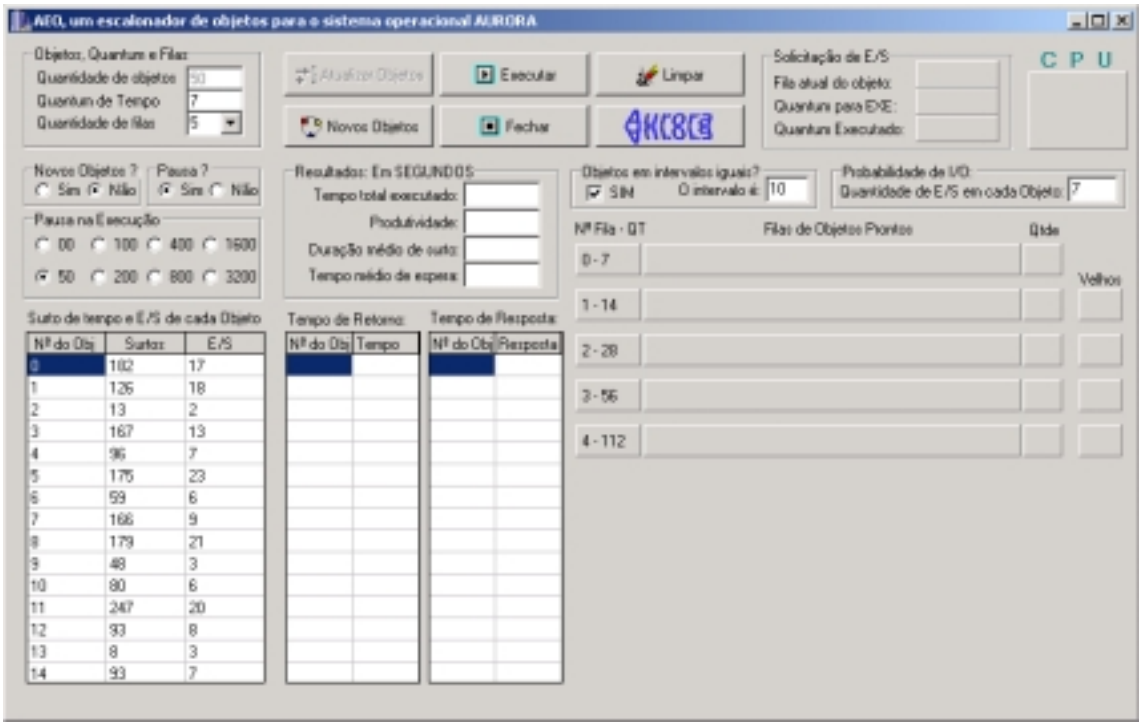


Figura 5.26 Primeiro exemplo da validação do modelo. Com 50 objetos inicialmente.

Neste primeiro exemplo, a validação do modelo foi configurada com 50 objetos para serem instanciados logo que clicar no botão “Atualizar Objetos”. O quantum de

tempo para objetos que entrarem na primeira fila será de sete milissegundos, sempre dobrando esse valor nas filas inferiores, como se vê na figura 5.27, a segunda fila, a de número 1, os objetos terão 14 milissegundos, a terceira é de 28 e assim por diante. Foi configurado também para criação de cinco filas e que a cada dez objetos, que fizerem uso do processador seja criado um novo objeto. A possibilidade de Entrada/Saída de dados é na ordem de sete, ou seja, todo valor contido no vetor da activity entre zero e 6, inclusive, é considerado como Entrada/Saída de dados, lembrando que esses valores são preenchidos aleatoriamente.

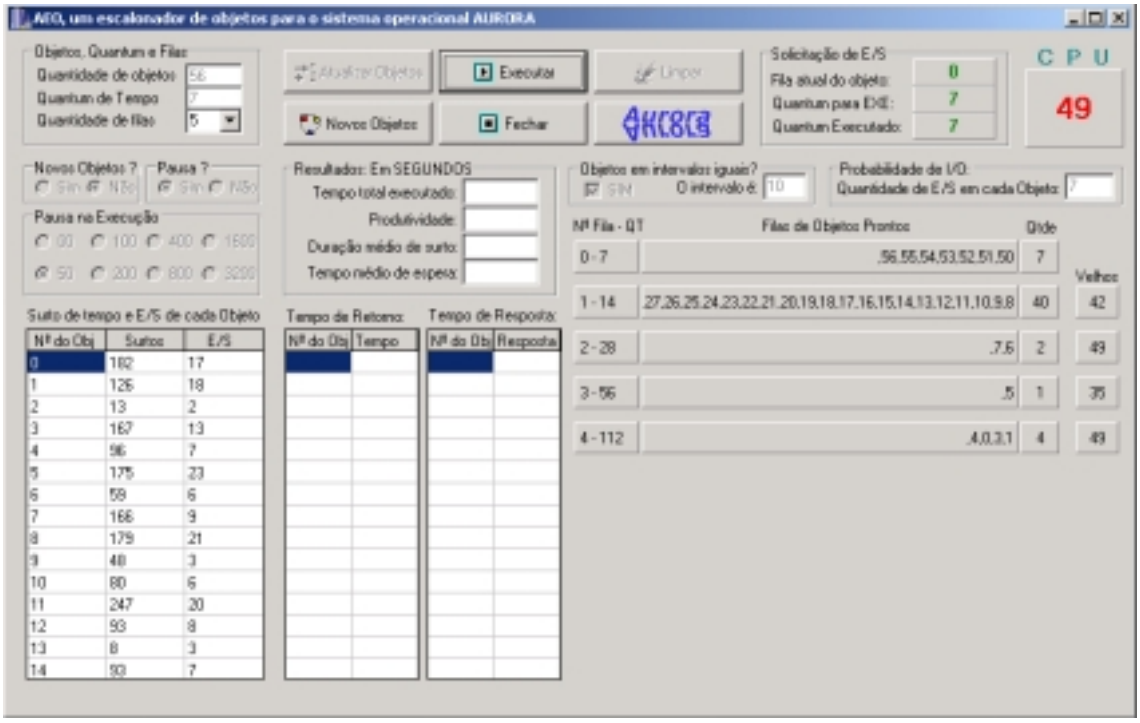


Figura 5.27 Primeiro exemplo da validação do modelo. Já com 56 objetos.

Na figura 5.28 já é demonstrado o modelo em funcionamento, ou seja, em execução. Percebe-se que, no momento da captura da imagem, o objeto que se encontrava fazendo uso da CPU era o objeto de número 49, que se apresenta em formato negrito vermelho, e que sua origem foi da fila zero, tendo sido utilizado todo o tempo que lhe cabia, o qual era de sete milissegundos. Observa-se também que todas as filas contêm objetos e ao lado de cada fila, a partir da segunda, há um controle para

objetos envelhecidos, por exemplo, o próximo elemento a ir para o processador será o objeto de número 5, que se encontra na fila de número 3, pois o seu tempo de espera já permite que faça uso do processador, ou seja, 49 milissegundos (7^2).

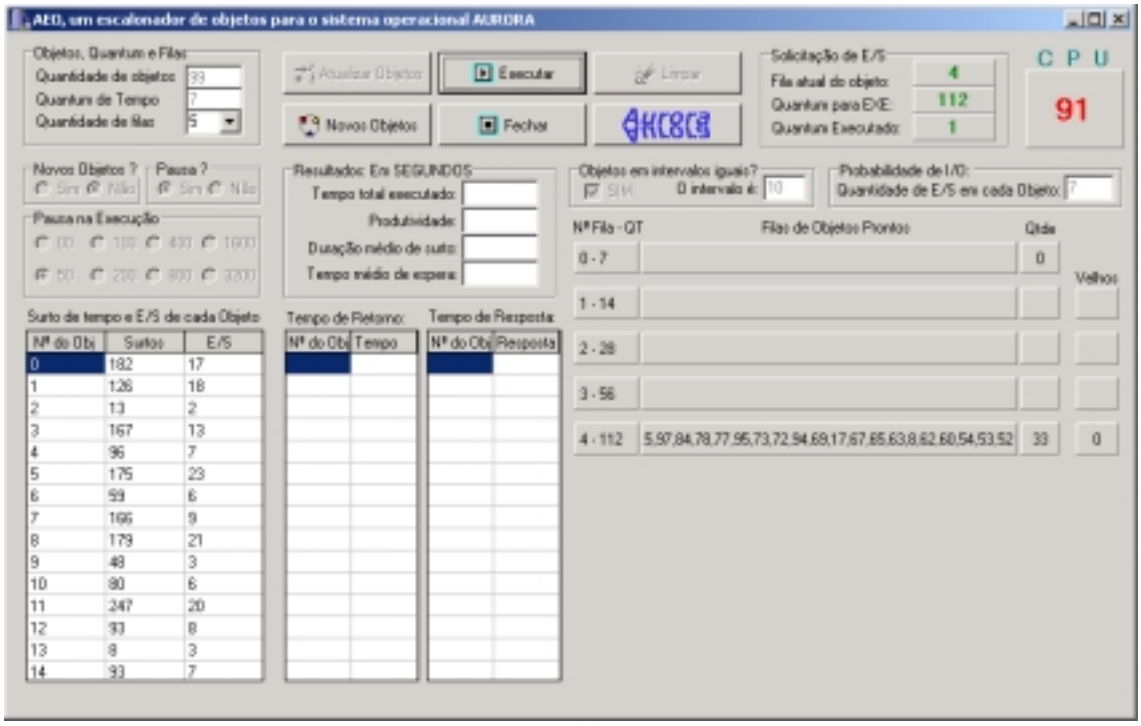


Figura 5.28 Primeiro exemplo da validação do modelo. Já com 99 objetos.

Na figura 5.29, já quase no final da execução de todos os objetos, com sua capacidade máxima de objetos permitidos, a qual é o dobro da que se iniciou, agora com 99 objetos, sendo que começou com 50, por esta razão as primeiras filas encontram-se vazias, ficando todos os objetos na última fila. Se por ventura, entrasse um novo objeto neste instante, iria ser inserido na fila de número zero.

Depois de todos os objetos executados, foi emitido um relatório referente a cada objeto e a todos os objetos gerados do início ao fim da execução do sistema. Por exemplo, o objeto de número zero tinha um surto de 182 milissegundos, com um total de 18 solicitações de Entrada/Saída de dados. Não houve tempo resposta, pois era o primeiro objeto da fila não teve, portanto, que esperar instante algum para obter sua primeira resposta, enquanto outros já tiveram um tempo razoável de espera. O objeto zero teve tempo de retorno, ou seja, até que finalizasse a sua execução um total de

10.920 milissegundos, como mostra a figura abaixo, o objeto de número dois já teve o tempo de resposta reduzido, isso em consideração ao seu pequeno surto de tempo para execução, ou seja, 13 milissegundos, indo ao processador no máximo 4 vezes.

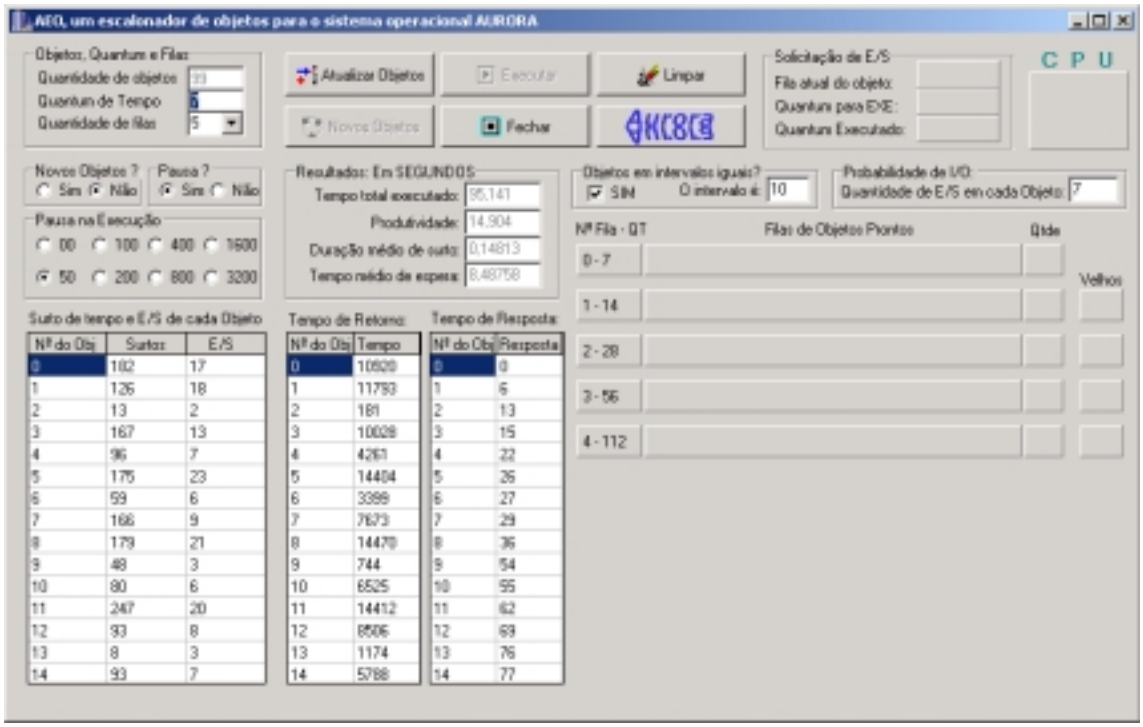


Figura 5.29 Primeiro exemplo da validação do modelo. Final de execução.

5.10.4 Conclusão

Diante do exemplo acima, percebe-se a flexibilidade dos objetos nas filas, dando ênfase a objetos envelhecidos em filas inferiores. Vale salientar que o objetivo e vantagem principal do **"Algoritmo por Múltiplas Filas com Realimentação"**, que é o modelo para o AEO, visa dar maior prioridade às tarefas que mais fazem solicitações de ENTRADA/SAÍDA. Com esse critério de escalonamento o AEO terá o tempo de retorno reduzido, levando em consideração que as tarefas com maior número de solicitação de ENTRADA/SAÍDA estarão mais vezes na CPU, com menos quantum de tempo, porém em maior prioridade. Enquanto outros tipos de tarefas irão menos à CPU, terão menos prioridade, mas em compensação ficarão mais tempo executando, podendo terminar rapidamente toda a tarefa. Com esse critério a CPU estará sempre ocupada, o que é o ideal.

Outro fator importante é que o modelo de algoritmo implementado, o AEO, estará sendo executado em um sistema operacional orientado a objeto, com reflexão computacional. Como o modelo do sistema operacional AURORA é exclusivamente para objetos, as filas de objetos em estados de pronto do escalonador, que são também orientadas a objetos, poderão ter objeto em espera, e ao mesmo tempo, ser ativado por diversas vezes, pois serão escalonados ponteiros de objetos, e isto possibilitando que o mesmo objeto seja ativado por diversas vezes ao mesmo tempo.

"Pesquisadores sugeriram que, para sistemas interativos (como os de tempo compartilhado), é mais importante minimizar a variância no tempo de resposta do que minimizar o tempo de resposta médio. Um sistema com um tempo de resposta razoável e previsível pode ser considerado mais desejável do que um sistema que é mais rápido em média, mas é altamente variável. No entanto, pouco trabalho foi realizado em algoritmos de escalonamento de CPU que minimizem a variância", SILBERSCHATZ (2000).

6. Conclusão

Conclui-se este trabalho de pesquisa, apresentando os resultados obtidos, a contribuição ao meio científico, bem como, abertura para trabalhos futuros.

Os resultados obtidos na implementação do “AEO, um escalonador para o sistema operacional AURORA”, foram satisfatoriamente alcançados, tendo em vista que o objetivo deste projeto é a implementação de um algoritmo de escalonamento e que suporte o modelo de um sistema operacional orientado a objetos: o AURORA. No capítulo 5 foram descritas a modelagem, a implementação e a validação do modelo, como também uma simulação para o modelo implementado. Partiu-se do princípio de utilizar algum dos algoritmos já existentes, ou seja, o “Algoritmo de Escalonamento por Múltiplas filas com Realimentação”, acrescentando algumas outras características ao modelo final, que é o tratamento a objetos envelhecidos na fila.

A primeira grande contribuição ao meio científico e acadêmico com este trabalho foi a implementação do escalonador ao sistema operacional AURORA, que até agora não tinha nenhum escalonador, que é o resultado da tese de doutorado do Prof. Dr. Sc. Luiz Carlos Zancanella. Destacando que o ambiente sobre o qual o Algoritmo de Escalonamento de Objetos será executado é um ambiente reflexivo e que suporta exclusivamente o modelo de objetos, visto que o modelo escolhido para a implementação (Algoritmo de Escalonamento por Múltiplas Filas com Realimentação), do qual se originou o AEO, permite que várias ativações possam ser feitas para o mesmo objeto que se encontra em uma das filas de objetos em estado de pronto.

A segunda contribuição foi o acréscimo às características nessa política de escalonamento escolhida, ou seja, a aplicabilidade à solução de problemas de envelhecimento de tarefas na fila (aging), caso contrário poderia ocorrer de tarefas ficarem em estados obsoletos e nunca irem ao processador, que é problema característico em múltiplas filas e prioridades. Um fato curioso: *“dizem, inclusive, que quando o IBM 7094 do MIT foi desligado em 1973, encontraram um processo de baixa prioridade que tinha sido submetido em 1967 e que ainda não tinha sido executado”*, SILBERSCHATZ (2000).

Outra contribuição será a abertura de novos temas para futuros trabalhos, por exemplo, desenvolver um software para cálculo da média de tempo de espera em diferentes políticas de escalonamento.

O AEO foi implementado em C++ com uso do ambiente e compilador compatível com ANSIC, o Borland C++ Builder 6.0, por ser a mais moderna das versões dos

compiladores e pelos ótimos recursos oferecidos por esta ferramenta. Levou-se em conta também a grande quantidade de literatura acadêmica e comercial que trata de trazer soluções a grandes variedades de problemas, considerando que sistemas operacionais importantes existentes no mercado foram implementados parcialmente e, até, totalmente nesta linguagem. A portabilidade e recurso oferecidos pela linguagem de programação C++ foram os estímulos para escolhê-lo como ferramenta para implementação deste escalonador.

A modulagem das classes do Algoritmo de Escalonamento de Objetos (AEO) foi realizada com o uso de UML (Unified Modeling Language - Linguagem de Modelagem Unificada), ambiente gráfico.

Bibliografias

BALZAN, José Rodrigo. **Reflexão Computacional para Gerenciamento de Objetos Distribuídos**. Dissertação (Mestrado em Informática) - Universidade Federal de Santa Catarina, Florianópolis, 39 p, 2001.

BARRETO, Jorge Muniz. **Inteligência Artificial:** No Limiar do Século XXI. Florianópolis: Editora Duplic, 2000. 2. ed. 324 p.

BROOKSHER, J. Glenn. **Ciência da Computação: Uma Visão Abrangente.** trad. Cheng Mei Lee. Porto Alegre: Editora Bookman, 2000. 5. ed. 499 p.

FERBER, J. **Computational Reflection in Class Based Object-Oriented Languages**. Sigplan **Notices**, OOPSLA New York, 1989, v.24, n.10. p.317-326.

FERREIRA, José Ricardo da Silva. **Escalonador Inteligente de Tarefas para Aplicações Robóticas**. Dissertação (Mestrado em Informática) – Centro de Ciências e Tecnologia, Universidade Federal da Paraíba, Campina Grande, 101 p, 1999.

JAMSA, Kris; KLANDER, Lars. **Programando em C/C++: A Bíblia.** trad. Jeremias René D. Pereira dos Santos. São Paulo: Makron Books, 1999. 1012 p.

LISBÔA, M. L. B. **Arquiteturas de meta-nível**. In: Simpósio Brasileiro de Engenharia de Software, XI, Fortaleza, CE, 1997.

MACHADO, Francis Berenger, MAIA, Luiz Paulo. **Arquitetura de Sistemas Operacionais.** Rio de Janeiro: Editora LTC, 19972. ed. 232 p.

MAES, Pattie. **Issues in Computacional reflection. Meta-Level architectures and reflection**. Bélgica: D. Nardi Editors

MIRANDA, Fernando C. **Sistemas Operacionais Orientadas a Objetos**. Monografia, Universidade Federal de São Paulo, Instituto de Ciências Matemáticas e de Computação, USP, São Carlos, SP, 2000.

OLIVEIRA, Rômulo Silva; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas Operacionais**. Porto Alegre, Instituto de Informática da UFRGS: Editora Sagra Luzzatto, 2000. 233 p.

ROGES, S. Pressman. **Engenharia de Software**. Trad. José Carlos Barbosa dos Santos. São Paulo: Editora Makron, 1995. 1056 p.

SILBERSCHATZ, Abraham; PETER, Galvin; GAGNE, Greg. **Sistemas Operacionais:** Conceitos e aplicações, Rio de Janeiro: Editora Campus, 2000. 585 p.

SHAY, William A. **Sistemas Operacionais**. trad. Mário Moro Fechio. São Paulo: Makron, 1996. 758 p.

STALLINGS, **William. Operating Systems:** Internals and Design Principles. 5. ed. New Jersey: Prentice, 1998. 781 p.

TANENBAUM, Andrew S.; WOODHULL, Albert S. **Sistemas Operacionais:** Projeto e Implementação, trad. Edson Furmankiewicz. Porto Alegre: RS, 2000. 760 p.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**, trad. Nery Machado Filho. Porto Alegre: Editora LTC, 2000. 493 p.

YOKOTE, Yasuhiko. **The Apertos Reflective Operating System: The Concept and Its Implementation. In Proceedings of the 19**

92 Conference on Object Oriented Programming System, Languages and Applications, October, 1992.

ZANCANELLA, Luiz C. **Estrutura Reflexiva para Sistemas Operacionais multiprocessados**. Tese (Doutorado em Informática), Instituto de Informática Universidade Federal do Rio Grande do Sul, Porto Alegre, 123p, 1997.

Anexo 1

*Após apresentar algumas políticas de escalonamento no capítulo 2, a seguir, será explicitada a implementação da proposta para o "**Algoritmo de Escalonamento para o Sistema Operacional AURORA**".*

1 Type.h

```
//C:\aurora\hardware\h\Types.h

//---------------------------------------------------------------------
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// Colaborador: M. Sc. Deniz Pedrozo de Almeida - Cuiabá/MT maio/2002
//---------------------------------------------------------------------

#ifndef     _Types_h_DEFINED
#define     _Types_h_DEFINED
typedef     unsigned char     byte;       //  8-bits sem sinal
typedef     unsigned short    word;       // 16-bits sem sinal
typedef     unsigned int      longword;   // 32-bits sem sinal
typedef     char        sbyte;            //  8-bits com sinal
typedef     short       sword;            // 16-bits com sinal
typedef     int         slongword;  // 32-bits com sinal
typedef     byte*       pbyte;            // enderecamento de  8-bits
typedef     word*       pword;            // enderecamento de 16-bits
typedef     int*        plongword;  // enderecamento de 32-bits

typedef     unsigned short  boolean;

typedef     unsigned long     magicword;  // identificacao de objetos
typedef     unsigned char     u_char;
typedef     unsigned short    u_short;
typedef     unsigned int      u_int;
typedef     unsigned long     u_long;

typedef     long        off_t;
typedef     unsigned int      size_t;

class   Activity;

typedef     Activity*       pActivity;

typedef int quantum = 8;    //Quantum de tempo: Em milissegundos

#define     NULL  0
#define     TRUE  (!NULL)
#define     FALSE NULL

//---------------------------------------------------------------------
#endif      // _Types_h_DEFINED
```

2 Activity.H

```
//C:\aurora\common\h\activity.h

//-----------------------------------------------------------------------
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// Colaborador: M. Sc. Deniz Pedrozo de Almeida - Cuiabá/MT maio/2002
//-----------------------------------------------------------------------

#ifndef Activity_h_DEFINED
#define Activity_h_DEFINED

#include  <\aurora\hardware\h\CPU.h>
#include  <\aurora\hardware\h\Types.h>
#include  <\aurora\common\h\System_ID.h>
#include  <\aurora\common\h\EntryTable.h>
#include  <\aurora\common\h\Structures.h>
#include  <\aurora\metacore\h\metacore.h>
#include  <\aurora\scheduler\escalonar.h>



//-----------------------------------------------------------------------
// A classe Activity  é utilizada  para  representar  o aspecto
// dinâmico dos objetos. Seus atributos incluem: identificação do
// objeto, link para meta-nível, lista de mensagens disponíveis.
// e o estado da execução (dependentes da CPU)

// Diagrama de estados
//
//                new Activity
//             M/R     |
//       +--------+ |
//     |         | |   Exception
//     +---+     v v   --------->
//           sDORMINDO          sSUSPENSO
//     +---+     ^ |   <---------
//     |         | |    ExecuteR
//     +--------+ |
//      ExecuteM/ |
//      ExecuteR  |
//                v
//              delete Activity
//


enum mcState { sSUSPENSO, sDORMINDO };


class Activity
{

   protected:

      CPUContext  Context;    // Estado dos registradores da CPU
      AID*        Identify;   // Identificao da Activity
      EntryTable* ExecQueue;  // Lista de Entradas (mensagens)
```

```
        pActivity   Meta;       // pointer para o metaobjeto

        Entry       Next;
        plongword   Address;
        MetaCore*   AuroraMetaCore;
        mcState       state;      // Estado da Activity

        int QT_serEXE;  //Quantum de tempo para ser executado
        int QT_foiEXE;  //Quantum de tempo que foi executado

    public:

        Activity ( );
        Activity ( size_t size, CPUMode mod );
        ~Activity ( );

        // métodos usados por metacore
        void      SetMeta ( pActivity ThisMeta );
        void      SetMode ( CPUMode mod );
        pActivity GetMeta ( );
        CPUMode   GetMode ( );
        void      SetEntry ( Entry n );
        void      SetAddress ( plongword address );
        plongword GetAddress ( );

        // dependentes de hardware
        //      void SetContextoCPU( CPUContext* context );
        //      CPUContext GetContextoCPU( );
        //      void SetInterruptEnable (boolean in);

        // métodos usados por objetos da aplicacao e do sistema
        void   ExecuteM ( Message* pMsg );
        void   ExecuteR ( MessageR* pMsg );

        void atribuir_QT_serEXE ( int Q);
        void atribuir_QT_foiEXE ( int Q);
        int  retornar_QT_foiEXE (      );
        int  retornar_QT_serEXE (      );
        void Inicializa(int QT);

};


//---------------------------------------------------------------------
#endif   // Activity_h_DEFINED
```

3 Activity.CPP

```
//C:\aurora\common\h\activity.cpp

//---------------------------------------------------------------------
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// Mo. Deniz Pedrozo de Almeida     -     Cuiabá/MT maio/2002
//---------------------------------------------------------------------
```

```
#include <\aurora\common\h\activity.h>

//---------------------------------------------------------------------

Activity::Activity( int QT )
{
      AuroraMetaCore = LocalMetaCore;

      Identify = new AID();
      ExecQueue = new EntryTable();

      // incluir inicializacoes dependentes de hardware

      QQ_serEXE = QT;  //Quantum de tempo inicial para ser executado
      QQ_foiEXE = 0;   //Quantum de tempo que foi executado
}

//---------------------------------------------------------------------

Activity::Activity( size_t size, CPUMode mod , int QT)
{
      AuroraMetaCore = LocalMetaCore;

      Identify = new AID();
      ExecQueue = new EntryTable( size );

      Context.mode = mod;
      //incluir inicializacoes dependentes de hardware

      QQ_serEXE = QT;  //Quantum de tempo inicial para ser executado
      QQ_foiEXE = 0;   //Quantum de tempo que foi executado

}

//---------------------------------------------------------------------

Activity::~Activity()
{
      delete Identify;
      delete ExecQueue;
}

//---------------------------------------------------------------------

void Activity::SetMeta( pActivity ThisMeta )
{
      Meta = ThisMeta;
}

//---------------------------------------------------------------------

pActivity Activity::GetMeta( )
{
      return ( Meta );
}

//---------------------------------------------------------------------

void Activity::SetMode (CPUMode mod)
{
```

```
    Context.mode = mod;
}

//---------------------------------------------------------------------

CPUMode Activity::GetMode ()
{
      return ( Context.mode );
}

//---------------------------------------------------------------------
void Activity::SetEntry ( Entry n )
{
      Next = n;
}

//---------------------------------------------------------------------

void Activity::SetAddress ( plongword address )
{
      Address = address;
}

//---------------------------------------------------------------------

plongword Activity::GetAddress ( )
{
      return ( Address );
}

//---------------------------------------------------------------------

void Activity::atribuir_QQ_serEXE (int Q)
{
        QQ_serEXE = Q;
}

//---------------------------------------------------------------------

void Activity::atribuir_QQ_foiEXE (int Q)
{
        QQ_foiEXE = Q;
}

//---------------------------------------------------------------------

int  Activity::retornar_QQ_foiEXE (  )
{
        return QQ_foiEXE;
}

//---------------------------------------------------------------------

int  Activity::retornar_QQ_serEXE ( )
{
        return QQ_serEXE;
}

//---------------------------------------------------------------------

void Activity::Inicializa(int QT)
```

```
{
     QT_ser_EXE =  QT;
     QT_foi_EXE = 0;
}
```

6 Escalonador.H

```
//C:\aurora\Scheduler\ESCALONADOR.h

//---------------------------------------------------------------------
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
// Scheduler of Processes  - UFSC/UNIRONDON
// Author  M. Sc. Deniz Pedrozo de Almeida  -  Cuiabá 10/04/2002
// Advisor Dr. Luiz Carlos Zancanella
//---------------------------------------------------------------------

#include <\aurora\common\h\activity.h>
#include  <\aurora\hardware\h\Types.h>

//---------------------------------------------------------------------

#ifndef ESCALONADORH
#define ESCALONADORH

#include "Activity.cpp"

class FilaLista
{
     private:
          struct TipoCelula
          {
               Activity Item;
               struct TipoCelula *Prox;
          };

          TipoCelula *Primeiro;
          TipoCelula *Ultimo;
          int numero;
          int Quantidade;
          int Envelhecidos;


     public:

          FilaLista ( ); //construtor padrão
          int Vazia ( );
          int Cheia ( );
          void Excluir( );

          Activity Primeiro_No ( );
          void Inserir(Activity Elemento); //inserir um elemento na lista

          void NumeroDaFila (int);
          int  RetornaNumeroDaFila();

          void QuantidadeObjetosNaFila(int);
          int  RetornaQuantidadeObjetosNaFila( );
```

```
          void ObjetosEnvelhecidos(int);
          int  RetornaObjetosEnvelhecidos( );

};//fim classe TipoLista


class ESCALONA
{
     private:
          FilaLista FILAS[10];
          int QUANTUM;

     public:
          ESCALONA(int QT);
          ~ESCALONA();
          void Carrega_Objetos_na_fila(Activity Act);
          void Adiciona_QUANTUM(int QT);
          Activity SCHEDULING ();

          int RetornaMaiorValorFilaVelho(int);
          int RetornaNumeroDaFilaAtual(int);

          void InformacaoNaFila(int indice, int Qual, int valor);
          int  RetornaInformacaoNaFila(int indice, int Qual);

          int RetornaFilaParaPrincipal( );

};

//---------------------------------------------------------------------
#endif
```

7 Escalonador.CPP

```
//C:\aurora\Scheduler\ESCALONADOR.CPP


//---------------------------------------------------------------------
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
// Algoritmo de Escalonamento de Objeto - AEO  - UFSC/UNIRONDON
// Author  M. Sc. Deniz Pedrozo de Almeida  -  Cuiabá 10/04/2002
// Advisor Dr. Luiz Carlos Zancanella
//---------------------------------------------------------------------

#include "ESCALONADOR.h"
#include <stdio.h>
#include <math.h>


//---------------------------------------------------------------------

FilaLista::FilaLista( ) //construtor padrão
{
     Quantidade = 0;
     Envelhecidos = 0;
     Primeiro = Ultimo = new(TipoCelula);
     if (Primeiro == NULL && Ultimo == NULL)
          printf("Memoria insuficiente !!!");
```

```
     else
          Primeiro->Prox = NULL;

}

//---------------------------------------------------------------------

void FilaLista::NumeroDaFila (int N)
{
     numero = N;
}

//---------------------------------------------------------------------

int  FilaLista::RetornaNumeroDaFila( )
{
     return numero;
}

//---------------------------------------------------------------------

void FilaLista::QuantidadeObjetosNaFila(int N)
{
     Quantidade = N;
}

//---------------------------------------------------------------------

int  FilaLista::RetornaQuantidadeObjetosNaFila( )
{
     return Quantidade;
}

//---------------------------------------------------------------------

void FilaLista::ObjetosEnvelhecidos(int N)
{
     Envelhecidos = N;
}

//---------------------------------------------------------------------

int  FilaLista::RetornaObjetosEnvelhecidos( )
{
     return Envelhecidos;
}

//---------------------------------------------------------------------

int FilaLista::Vazia(void)
{
     return(Primeiro == Ultimo);
}

//---------------------------------------------------------------------

Activity FilaLista::Primeiro_No ( )
{
     TipoCelula *Aux = Primeiro->Prox;
     Activity Act = Aux->Item;
     Excluir();
```

```
     return Act;
}

//---------------------------------------------------------------------

void FilaLista::Inserir(Activity Elemento)
{
     Ultimo->Prox = new(TipoCelula);
     if(Ultimo->Prox == NULL)
          printf("Memoria insuficiente !!!");
     else
     {
          Ultimo = Ultimo->Prox;
          Ultimo -> Item = Elemento;
          Ultimo -> Prox = NULL;
     }
}

//---------------------------------------------------------------------

void FilaLista::Excluir( )
{
     Primeiro = Primeiro->Prox;
     Ultimo->Prox = NULL;
}

//---------------------------------------------------------------------

//construtor padrão de ESCALONA
ESCALONA::ESCALONA(int QT)
{
     QUANTUM = QT;

     for (int i = 0; i < 10; i++)
     {
          FILAS[i].QuantidadeObjetosNaFila(0);
          FILAS[i].ObjetosEnvelhecidos(0);
     }

}

//destrutor
ESCALONA::~ESCALONA() { }

//---------------------------------------------------------------------

void ESCALONA::Adiciona_QUANTUM(int QT)
{
     QUANTUM = QT;
}

//---------------------------------------------------------------------

void ESCALONA::InformacaoNaFila(int indice, int Qual, int valor)
{
     switch (Qual)
     {
            case 1: FILAS[indice].NumeroDaFila(valor);
            break;
            case 2: FILAS[indice].QuantidadeObjetosNaFila(valor);
            break;
```

```
            case 3: FILAS[indice].ObjetosEnvelhecidos(valor);
            break;
     }
}

//---------------------------------------------------------------------

int  ESCALONA::RetornaInformacaoNaFila(int indice, int Qual)
{
     int r;
     switch (Qual)
     {
            case 1: r = FILAS[indice].RetornaNumeroDaFila( );
            break;
            case 2: r = FILAS[indice].RetornaQuantidadeObjetosNaFila( );
            break;
            case  3:  r  =  FILAS[indice].RetornaObjetosEnvelhecidos(  );
            break;
     }
     return r;
}

//---------------------------------------------------------------------

int ESCALONA::RetornaNumeroDaFilaAtual(int N)
{
     int i = -1;
     do
     {
          i++;

     }while (QUANTUM * pow(2,i) < N );
     return i;
}

//---------------------------------------------------------------------

//Insere objetos na fila de execucao: 10 filas
void ESCALONA::Carrega_Objetos_na_fila(Activity Act)
{
     int QsE = Act.Retorna_QT_ser_Exe (); //8
     int QfE = Act.Retorna_QT_foi_EXE (); //0

     int Fila_Ser_Inserida = RetornaNumeroDaFilaAtual(QsE);

     if (QfE >= QsE && Fila_Ser_Inserida < QdeF)
         Fila_Ser_Inserida++;


      FILAS[Fila_Ser_Inserida].QuantidadeObjetosNaFila(FILAS[Fila_Ser_
      Inserida].RetornaQuantidadeObjetosNaFila() + 1);

     Act.Atribuir_QT_ser_EXE(QUANTUM * pow(2,Fila_Ser_Inserida));

     ////Esta linha é apenas para uso do simulador
     Act.Atribui_Valor_para_INFORMACAO(5,Fila_Ser_Inserida);

     //Em que fila ira inserir
     FILAS[Fila_Ser_Inserida].Inserir(Act);
}
```

```
//---------------------------------------------------------------------

int ESCALONA::RetornaMaiorValorFilaVelho(int N)
{
     int R = 99;
     int sair = 0;
     for (int i = 1; (i < N && sair == 0); i++)
     {
          if (FILAS[i].RetornaObjetosEnvelhecidos() >= pow(QUANTUM,2) &&
              FILAS[i].RetornaQuantidadeObjetosNaFila() > 0)
          {
               FILAS[i].ObjetosEnvelhecidos(-QUANTUM);
               sair = 1;
               R = i;
          }
          if (FILAS[i].RetornaQuantidadeObjetosNaFila() > 0)
               FILAS[i].ObjetosEnvelhecidos(QUANTUM +
                  FILAS[i].RetornaObjetosEnvelhecidos());
     }
     return R;
}

//---------------------------------------------------------------------

//escalona objetos
Activity ESCALONA::SCHEDULING ( )
{
     Activity a;
     int  QF  =  StrToInt(Form_AEO->ComboBox1->Items->Strings[Form_AEO-
>ComboBox1->ItemIndex]);

     int rr = -1, saindo = 0;
     while (rr < QF && saindo == 0)
     {
          rr++;
          if (FILAS[rr].RetornaQuantidadeObjetosNaFila()!= 0)
               saindo = 1;
     }

     int  Nova = RetornaMaiorValorFilaVelho(QF);

     if (Nova != 99)
         rr = Nova;

     if (rr < QF)
     {
          a = FILAS[rr].Primeiro_No( );
          int NN = (FILAS[rr].RetornaQuantidadeObjetosNaFila() - 1);
          FILAS[rr].QuantidadeObjetosNaFila(NN);
     }
     else
          a.Atribui_Valor_para_INFORMACAO(0,99);

     return a;
}
//---------------------------------------------------------------------
```