UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



# Estrutura Reflexiva para Sistemas Operacionais Multiprocessados

por

LUIZ CARLOS ZANCANELLA

Tese submetida à avaliação como requisito parcial para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador



Porto Alegre, Dezembro de 1997.

# CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Zancanella, Luiz Carlos
Estrutura Reflexiva para Sistemas Operacionais Multiprocessados / por Luiz Carlos Zancanella. - Porto Alegre: CPGCC da UGRGS, 1997.

117 f.;il.

Tese(doutorado) - Universidade Federal do Rio Grande do Sul.
Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS 1997.
Orientador: Navaux, Philippe Olivier Alexandre

1.Sistemas Operacionais. 2. Multiprocessadores. 3. Orientação a Objetos.
4. Reflexão Computacioanl. I. Navaux, Philippe Olivier Alexandre. II. Título



Universidade Federal do Rio Grande do Sul

À minha vida: **Leonita, Shelen, Junior.**

Ao meu orientador: **Dr. Philippe Olivier Alexandre Navaux**

À memória do professor: **Dr. Hermann Adolf Harry Lucke**

# Sumário

# Lista de figuras

# Lista de tabelas

# Resumo

É crescente, nos últimos anos, a utilização da tecnologia de orientação a objetos para a construção de sistemas complexos. A aceitação de que tal tecnologia, além de facilitar a modularização e proporcionar maior reusabilidade, permitindo uma visão unificada dos sistemas, tem encorajado sua utilização na construção de sistemas operacionais, onde recursos do sistema e aplicações do usuário passam a ser modelados em termos da mesma abstração.

Na realidade, esta nova tecnologia de desenvolvimento de *software*, aliada a evolução do *hardware*, da tecnologia de comunicações e a necessidade de um incremento qualitativo, principalmente no que diz respeito a ambientes de programação e interfaces, está provocando o surgimento de uma nova geração de sistemas operacionais, mais dinâmicos, mais flexíveis e capazes de suportar de forma transparente a presença de processamento cooperativo, distribuído ou não, heterogêneo ou não.

Todavia, ainda que a literatura científica demonstre a aceitação do paradigma de orientação a objetos como um enfoque promissor a ser adotado na nova geração de sistemas operacionais, o estado atual da tecnologia de implementação e gerenciamento de objetos está aquém da consolidação. Este trabalho surgiu neste contexto com o objetivo de contribuir na busca de um modelo apropriado ao gerenciamento de objetos e capaz de proporcionar a existência de um modelo uniforme, tanto para o nível do sistema operacional como para o nível da aplicação.

O resultado da pesquisa desenvolvida foi o surgimento de um modelo estrutural orientado a objetos e baseado nas idéias da reflexão computacional, não somente como disciplina de implementação, mas como modelo conceitual para a implementação de sistemas operacionais multiprocessados.

**Palavras-Chave**: Sistemas Operacionais, Multiprocessadores, Orientação a Objetos, Reflexão Computacional.

**TITLE: "Reflective Structure for Multiprocessor Operating System"**

# Abstract

The use of object-oriented technology for the construction of complex systems has been increasing in recent years. The assumption that such technology, besides facilitating modularization, increases reusability and maintainability, providing a unified view of the systems, has encouraged its use in building operating systems, where the system resources and the user's applications come to be modeled in terms of the same abstraction.

Actually this new technology for development of software, associated with the evolution of hardware, as well as communication technology and the need for qualitative enhancement, mainly concerning programming and interface environments, is giving rise to a new generation of operating systems, more dynamic, more flexible and capable of maintaining, in a transparent way, the presence of cooperative processing, distributed or non-distributed, heterogeneous or homogeneous.

However, although the scientific literature shows an acceptance of the object-oriented paradigm as a promising focus(insight) to be adopted in the new generation of operating systems, the current state of technology for object management is still far from being one of consolidation. This project has emerged in this context, with the aim of contributing to the search for an appropriate model for office management that is able to provide a uniform model, not only on the operating system level, but also on the application level.

The result of the research is a new object-oriented structural model, based on the concepts of computational reflection, both as an implementation discipline and as a conceptual model for the utilization of multiprocessor operating systems.

These features are very useful in developing operating systems which contain components, including resources and applications that are modeled in the same way, using object-oriented abstraction.

**Keywords:** Operating System, Multiprocessor, Computational Reflection, object-oriented,.

**Capítulo 1**

# Introdução

*Situa o trabalho aqui apresentado em termos da motivação para desenvolvê-lo, dos problemas existentes e dos resultados buscados ao longo de seu desenvolvimento.*

## 1.1 Motivação.

Observa-se, ao longo da história da ciência da computação, uma demanda crescente por equipamentos mais poderosos. Esta necessidade induziu a uma constante busca por computadores com poder de processamento cada vez maior. Processadores mais poderosos, periféricos mais rápidos, execução simultânea de diversas atividades de hardware, multiprogramação, são alguns exemplos de soluções largamente utilizadas e que visam obter ganhos na velocidade de processamento [FER88].

Todavia, devido a limitações de natureza física, é crescente nos últimos anos a aceitação de que, somente a utilização de processamento paralelo, será capaz de suprir toda a demanda computacional exigida pelas complexas aplicações científicas e da engenharia moderna. Neste particular, multicomputadores e multiprocessadores são vistos como uma alternativa capaz de proporcionar, não somente maior poder computacional às atuais arquiteturas, mas talvez, mais importante que isto, oferecer significativos incrementos na confiabilidade dos computadores, além de uma melhor relação custo/desempenho.

A complexidade destes ambientes, sua exploração eficiente e a necessidade de transferir seus benefícios funcionais ao usuário, introduziu novos desafios ao projeto e implementação de sistemas operacionais. Como conseqüência, os anos recentes foram marcados pelo surgimento de uma nova geração de sistemas operacionais [BLA86, CAM87, NIC87, HER88, SHA89, MUL90, YOK92, SCH94, BUT94], mais dinâmicos, mais flexíveis e capazes de suportar de forma transparente a presença de processamento cooperativo, distribuído ou não, heterogêneo ou não.

Evidentemente, existem outros fatores responsáveis pelo atual estímulo da pesquisa em sistemas operacionais, e entre estes salienta-se o aspecto qualitativo [NIC89, OS/2-WARP, WIN95], principalmente no que diz respeito a ambientes de programação e interfaces. Parece claro que, apesar de desenvolvidos de forma independente, sistemas operacionais e linguagens de programação estão fortemente relacionados e que a crescente introdução de recursos às linguagens, antes somente providos pelos sistemas operacionais e vice-versa, demonstra a necessidade de se prover modelos capazes de suportar abstrações conceituais independentes de nível, recursos ou ambiente.

Entretanto, dentre os fatores de influência para o surgimento da nova geração de sistemas operacionais, talvez o mais importante seja a nova

tecnologia de desenvolvimento de software, em particular o modelo de objetos, responsável pela introdução de conceitos e abstrações que encapsulam naturalmente dentro do próprio modelo muitos dos problemas envolvidos no projeto de sistemas operacionais, tais como: identificação, proteção, atomicidade e sincronização.

Outro aspecto animador é a evidência de que a tecnologia de objetos, além de encorajar a modularização e proporcionar maior reusabilidade, permite que recursos do sistema e aplicações do usuário sejam modelados em termos da mesma abstração. Tal característica introduz aos sistemas operacionais uma habilidade adicional, que é a de poder manipular o comportamento transparente e dinâmico dos sistemas, visto que recursos, serviços e o próprio sistema podem ser modelados de forma abstrata.

A premissa do modelo orientado a objetos é que o mesmo se propõe a modelar objetos que no mundo real são naturalmente concorrentes (e distribuídos) e cujo processamento pode ser simplificadamente representado como um conjunto de mensagens fluindo entre objetos, executando de forma paralela. Se partirmos da observação de que programas concorrentes, tais como sistemas operacionais, ao serem construídos podem ser modelados como uma coleção de objetos concorrentes, nos induz a pensar que: o paradigma de objetos e a exploração simultânea do paralelismo representam uma combinação poderosa tanto para o desenvolvimento como para a execução de programas concorrentes.

Na literatura científica são encontradas inúmeras referências de projetos de pesquisa, demonstrando a aceitação do paradigma de orientação a objetos como um enfoque promissor a ser adotado na nova geração de sistemas operacionais. CHAOS [SCH89], SOS [SHA89], Elmwood [LEB89], Cool [HAB90] e [SCH94] são alguns exemplos de projetos que visam combinar os benefícios do modelo orientado a objetos com as arquiteturas multiprocessadoras.

É notório, hoje, o crescimento da importância do modelo orientado a objetos na construção de sistemas operacionais e, como conseqüência, a aceitação de que tal tecnologia direcionará o desenvolvimento dos sistemas operacionais nos próximos anos [KIC93]. Alguns sinais bastante claros desta importância podem ser apontados:

1. O nível de complexidade requerido para a nova geração de sistemas operacionais devido a diversificação de ambientes paralelos, sendo a tecnologia de orientação a objetos um dos mecanismo mais promissores para o domínio da complexidade.
2. Popularização da metodologia de desenvolvimento de software orientado a objetos e o conseqüente aumento da importância que as linguagens orientadas a objetos vem adquirindo dia a dia.

Todavia, analisando-se as atuais implementações de modelos orientados a objetos, constata-se que algumas características do modelo de objetos, quando implementadas, não têm apresentado resultados satisfatórios, executando de forma muito mais lenta que suas implementações nas linguagens tradicionais. Por exemplo, a implementação mais eficiente para troca de mensagens em Smalltalk-80 executa em uma razão de 1/10 da velocidade de uma implementação C.

Esta perda de eficiência, segundo [CHA92], é ocasionada em grande parte pela carência de tecnologia de implementação adequada ao modelo de objetos. Tal fato tem forçado muitas das linguagens orientadas a objetos serem projetadas de forma híbrida, ou mesmo traduzidas em alto nível para as tradicionais linguagens não orientadas a objetos na busca de implementações eficientes.

Para Chin [CHI91], o problema não está localizado somente no nível de implementação das linguagens. Segundo ele, sistemas cuja proposta é suportar a abstração de objetos, devem gerenciar objetos eficientemente a nível de sistemas operacionais. Note que o processo de gerenciamento de objetos concorrentes não é trivial, visto existirem vários aspectos a serem considerados, tais como: *controle das ativações, sincronização, segurança* e *confiabilidade.*

Existem ainda características relevantes que, embora não digam respeito ao gerenciamento de objetos propriamente, são igualmente fundamentais para a questão da eficiência, tais quais: *a estrutura dos objetos e o gerenciamento da interação entre objetos.*

Relativamente a estas características, existem questões não resolvidas no escopo dos sistemas operacionais. Por exemplo, no que diz respeito à estrutura dos objetos, um problema existente em sistemas que suportam objetos de granulosidade fina é o excessivo *overhead*[1], obtido em

[1] *Custo de comunicação e processamento*

função da necessidade de tornar quase toda operação uma ativação a um objeto [CHI91].

Outra questão ainda não resolvida, diz respeito à representação eficiente da ativação a um objeto. Conceitualmente, objetos são ativados de maneira uniforme através de mensagens, no entanto, aparentemente, "nem todas as ativações são iguais", variando desde ativações rápidas e não confiáveis, porém úteis para aplicações de tempo real, até ativações lentas, porém confiáveis, requeridas para certas aplicações distribuídas [BOD93].

Para Yokote [YOK92], existe um problema de conceituação. Segundo ele, atualmente os conceitos estruturais sobre os quais os sistemas operacionais estão construídos, estão voltados para o suporte a processos e conseqüentemente não possuem habilidade para manter e gerenciar objetos eficientemente. Deve-se notar que, sob o ponto de vista do sistema operacional, objetos são entidades de granulosidade mais leve que processos. Por exemplo, um objeto quando ativado não exige necessariamente a presença de um contexto e/ou de uma área de trabalho.

Propostas de modelos estruturais ditos "apropriados" ao gerenciamento de objetos podem ser encontrados em Apertos [YOK92], onde o modelo é baseado no conceito de computação reflexiva e em Cosmos [NIC89] que implementa um modelo estrutural baseado no conceito de árvores semânticas.

As afirmações anteriores estão contidas na colocação de Lea [LEA93], segundo o qual a maioria dos sistemas operacionais existentes não foram projetados para suportar as linguagens de programação modernas, particularmente as linguagens orientadas a objetos.

O projeto Aurora [ZAN93a] surgiu neste contexto com o objetivo de contribuir na busca de um modelo adequado ao paradigma de objetos e visando proporcionar através da exploração eficiente das arquiteturas multiprocessadoras, um suporte de software para a modelagem de ambientes apropriados à execução de modelos orientados a objetos. O modelo desta proposta, ainda que independente da implementação em particular, está inserido neste contexto e constitui-se numa das principais características que distingue Aurora de outros sistemas operacionais[2].

---

[2] *Esta análise não considera ambientes monolinguagens tais como SmallTalk*

## 1.2 Desafios

O contexto anterior situa o estágio da pesquisa em sistemas operacionais, evidenciando a necessidade de evolução da atual tecnologia de gerenciamento de objetos em diferentes níveis. Entretanto, partidário das colocações de Yokote[YOK92] e Lea[LEA93], nossa opinião é que:

> "o *problema principal da ineficiência no suporte a linguagens orientadas a objetos é causada pela diferença entre as abstrações providas pelo sistema operacional e aquelas oferecidas pelas linguagens.*"

Percebe-se no entanto, que a solução do problema de ineficiência no suporte ao modelo de objetos, deve ser abordado sob dois enfoques distintos: sob o enfoque *qualitativo* e sob o enfoque da *eficiência.*

Na opinião de Nicol [NIC87], quando um novo ambiente computacional é desenvolvido, a linguagem, o sistema operacional e possivelmente o hardware deveriam ser projetados simultaneamente, de modo que os diversos níveis do ambiente estejam fortemente relacionados. Existem alguns inconvenientes neste processo que Nicol chamou de *total system design.* Primeiro é que o desenvolvimento de *hardware* normalmente envolve recursos nem sempre alcançáveis, segundo é que o produto resultante é um ambiente monolinguagem.

A opinião de Nicol é apropriada sob o ponto de vista qualitativo, onde é evidente a necessidade de se prover para as arquiteturas atuais, sistemas multilinguagens capazes de suportar abstrações conceituais independentes de níveis, recursos ou ambientes. A dificuldade que se apresenta reside exatamente na modelagem de tais ambientes, que devem ser capazes de prover suporte eficiente em baixo nível e ao mesmo tempo manter as abstrações de alto nível.

Sob a ótica da eficiência é consenso a necessidade de uma tecnologia mais adequada ao modelo de objetos e a percepção de que a busca deste objetivo não pode desconsiderar a natureza concorrente (e distribuída) do modelo de objetos, sob pena de, ao descaracterizar o modelo, incrementar a complexidade de seu gerenciamento.

Evidentemente, temos plena consciência de que os parâmetros associados a objetos não são únicos, e alguns de significado relativo, especialmente quando o objetivo é alcançar desempenho na exploração de arquiteturas multiprocessadoras. Por exemplo, os principais parâmetros de

medição são: a razão de velocidade (*speed-up ratio* ou simplesmente *speed-up*) e a taxa de utilização dos processadores.

Neste aspecto, constata-se que os sistemas multiprocessados não apresentam um incremento linear na sua capacidade de processamento com o número de processadores, reduzindo a taxa de utilização dos $N$ processadores para uma razão entre ($\log_2 N$) e ($N$ / ln $N$) [HWA84]. Existem restrições de natureza física e teóricas para esta ociosidade nos processadores, entretanto um fator não único, mas diretamente relacionado é o grau de paralelismo extraído da aplicação.

São conhecidas as dificuldades existentes para a construção de aplicações paralelas, particularmente para as arquiteturas *MIMD*[3] onde o grau de paralelismo, muitas vezes, deve ser expresso pelo programador. Além da complexidade natural, a paralelização manual de uma aplicação pode produzir resultados não satisfatórios mesmo quando o número de processadores é pequeno. Por conseguinte, caso o número de processadores seja da ordem de centenas, ou mesmo dezenas, alcançar um grau de paralelização aceitável para a aplicação, além de ser uma tarefa árdua pode ser impossível.

Esta dificuldade pode ser justificada pela existência de inúmeras formas de paralelismo embutido. Mesmo no modelo orientado a objetos, onde a tarefa de desenvolvimento do software é facilitada pela sua característica de modularidade, extrair todo o paralelismo existente em um sistema não é uma tarefa trivial. Por exemplo, diferentes objetos podem ser ativados em paralelo, um único objeto pode ter diversos de seus métodos ativados simultaneamente, além disso cada método pode ele próprio conter operações em paralelo.

Uma conseqüência visível desta complexidade é, que apesar do avanço das arquiteturas multiprocessadoras conquistando espaço, não somente nas aplicações científicas, como também tornando-se disponíveis comercialmente, esses computadores continuam a margem da comunidade computacional, ainda que potencialmente mais promissores que os modelos seqüenciais.

A observação destes fatos leva a percepção de que:

1. o grau de paralelismo tem importância determinante no desempenho;

---

[3] *Multiple Instruction Flow- Multiple Data Flow*

2. a exploração efetiva do grau de paralelismo máximo, somente será alcançada às custas do esforço adicional de manipulação do código;
3. a extração e exploração transparente do paralelismo, pode ser um aliado importante à popularização dos sistemas multiprocessadores.

Tais constatações levam a apologia de que para sistema multiprocessados orientados a objetos, *a solução do problema de exploração do grau de paralelismo, é concomitante ao problema de gerenciamento dos objetos.*

Foi mencionada a existência de outros parâmetros igualmente responsáveis pela ociosidade dos processadores, todavia pelo raciocínio anterior e pelo fato de ser impossível obter um resultado ótimo para todas as medidas de desempenho [KOT84], parece plausível concentrar os esforços de otimização em relação ao grau de paralelismo extraído da aplicação.

Nestes enfoques e inserido no contexto do paradigma de orientação a objetos, são claros alguns, senão os principais, problemas e desafios que hoje se colocam:

- Primeiro, quanto ao modelo de ambiente capaz de viabilizar um suporte eficiente de baixo nível e ao mesmo tempo manter as abstrações de alto nível?.
- Segundo, quanto à extração do grau de paralelismo máximo no modelo orientado a objetos, visto ser ele um fator determinante na exploração das arquiteturas multiprocessadoras.

## 1.3 Escopo e Delimitações

Conforme explicitado anteriormente, existem diversas questões não resolvidas no atual estágio da tecnologia de gerenciamento de objetos. Todavia, nos parece claro que a consolidação da aparente adequabilidade do modelo orientado a objetos à construção de sistemas operacionais multiprocessados, somente será alcançada através da obtenção de um modelo de ambiente capaz de, ao mesmo tempo em que reflete o paradigma de orientação a objetos, ser capaz de gerenciar objetos eficientemente através da exploração do paralelismo potencial das arquiteturas multiprocessadoras.

Aceita-se que a obtenção deste ambiente somente pode ser alcançado através de uma arquitetura adequada, visto que tal ambiente, pela sua natureza, deve incorporar em tempo de execução características inerentes ao gerenciamento de objetos, tais como: criação dinâmica de objetos, acoplamento dinâmico[4], gerenciamento de mensagens, pesquisa a métodos e coleta de lixo.

Deve-se notar no entanto, que mesmo para estas arquiteturas, existe o aspecto conceitual do ambiente que deve ser adequadamente definido. Nossa convicção de que o tratamento uniforme das abstrações conceituais tem um papel importante na consolidação deste processo, tanto pelo aspecto qualitativo, quanto pelo aspecto da eficiência, nos levou a busca deste ambiente no sentido de contribuir principalmente em dois aspectos:

- *minimizar a diferença de abstrações conceituais existentes entre sistemas operacionais e linguagens de programação, visto ser nossa convicção de que o problema da ineficiência no gerenciamento de objetos está fortemente relacionado a diferença de abstrações conceituais existentes entre ambos.*
- *na adequabilidade do modelo orientado a objetos na exploração eficiente das arquiteturas multiprocessadas.*

Este trabalho é apresentado em duas etapas distintas. A primeira etapa, compreendida pelos capítulos 2, 3 e 4, apresentam os fundamentos teóricos sobre os quais este trabalho foi realizado. Na segunda etapa, compreendida pelos capítulos 5 e 6, são apresentados os resultados da pesquisa realizada sob a forma de proposta e validação de resultados. O capítulo 7 apresenta considerações finais e as conclusões do trabalho realizado.

---

[4] *do termo dynamic binding*

Capítulo 2

# Modelos Estruturais para Sistemas Operacionais

*Sintetiza e discute conceitos e características encontradas nos principais modelos estruturais já propostos e utilizados na implementação de sistemas operacionais.*

A maioria dos primeiros sistemas operacionais consistia simplesmente de um único programa escrito como uma coleção de procedimentos, cada um dos quais podendo chamar outro procedimento quando necessário. À medida que os sistemas operacionais evoluiram, tornou-se impossível manusear esta estrutura monolítica e o surgimento de novas estruturas mostrou-se uma necessidade.

Uma análise sobre os conceitos estruturais[1] surgidos é realizada neste capítulo através de um enfoque sobre suas caraterísticas principais.

## 2.1 Estruturas em Camadas

No modelo de camadas, que pode ser considerado como uma generalização do modelo monolítico, o sistema operacional é projetado como um conjunto de camadas sobrepostas (figura 2.1). Cada camada do sistema provê um conjunto de funções para a camada superior e é implementada em termos de funções providas pela camada inferior. Cada camada é constituida de processos concorrentes que implementam as funções da camada.

Esta divisão em camadas facilita a construção (e manutenção) do sistema, visto que processos pertencentes a uma dada camada não podem usar recursos dos processos pertencentes às camadas mais altas. Especificamente, as funções de um determinada camada podem ser totalmente compreendidas em termos da camada imediatamente abaixo, não sendo necessário nenhum conhecimento sobre os recursos disponíveis nas outras camadas, ou de como uma dada camada é utilizada pelas camadas acima. Os exemplos mais conhecidos deste tipo de sistema são o sistema THE [DIJ68] e o sistema Multics [ORG72].

Ainda que atraentes, sistemas estruturados em camadas tornam-se inerentemente difíceis para construção de sistemas complexos, porque nem sempre é possível garantir a existência de um processo restrito a uma camada. Por exemplo, muitos sistemas operacionais contam com a habilidade das funções de baixo nível (tais como sinais em Unix) para invocar processos a nível do usuário e isto é impossível de realizar usando estritamente técnicas de estruturação em camadas.

---

[1] *Este levantamento não considera ambientes monolinguagens como Emerald*

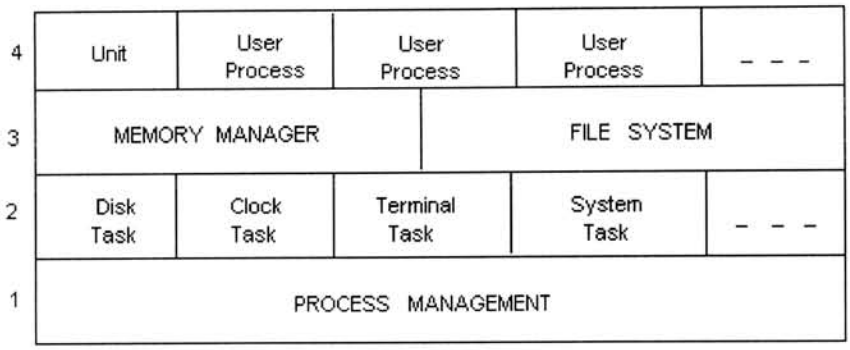


Figura 2.1: Estrutura interna do MINIX, quatro camadas

Nicol [NIC87] por sua vez, afirma que projetos baseados em camadas resultam em sistemas insatisfatórios por duas razões:

- A primeira é que a funcionalidade de tais sistemas é severamente limitada já que a base, normalmente de propósito geral, provê serviços que são raramente apropriados para as necessidades específicas de todos os usuários.
- A segunda razão é a redução no desempenho devido a existência de serviços não modelados para o suporte eficiente de qualquer aplicação de alto nível e deste modo o desempenho degrada, porque as aplicações devem ser implementadas no topo das camadas de um sistema de propósito geral.

## 2.2 Estrutura Hierárquica

A estrutura hierárquica é similar a estrutura em camadas, no sentido de que o sistema é construído nível a nível. Os níveis $L_0$, $L_1$,..., $L_n$ são ordenados tal que funções definidas no nível $L_i$ são também conhecidos para $L_{i+1}$ (e a descrição de $L_{i+1}$ para $L_{i+2}$, etc). Isto significa que, diferentemente do sistema em camadas, é permitido transpor níveis da hierarquia funcional. $L_0$ corresponde as instruções do *hardware* da máquina hospedeira.

Outro aspecto que difere da estrutura em camadas é que a estrutura hierárquica é baseada em funções e não em processos. Deste modo, cada nível é caracterizado por um conjunto de funções estaticamente identificadas e conhecidas.

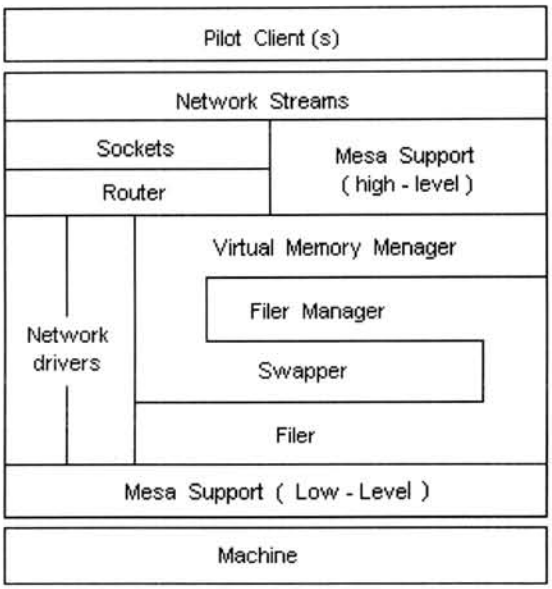


Figura 2.2 Estrutura hierárquica utilizada em Pilot

Também em contraste com a estrutura de camadas, onde o conceito de módulo[2] é usado para divisão de um nível em vários módulos distintos, na estrutura hierárquica o conceito de módulo é usado para expandir vários níveis por meio de um único módulo .

A figura 2.2 ilustra a estrutura hierárquica usada na implementação de Pilot [DAL80]. Uma característica da estrutura hierárquica e que pode ser vista na ilustração, é o entrelaçamento de módulos, como ocorre com os módulos do sistema de arquivos e memória virtual. Choices [CAM87] elabora o conceito de hierarquia através do uso da noção de classes[3], para implementar funções do sistema operacional, tais como gerência de memória e processos.

Como benefícios deste esquema pode-se ressaltar: modularização, reusabilidade, facilidade para manutenção, que pode ser facilmente extendida e assim por diante. Entretanto, tais benefícios são exclusivamente internos ao sistema operacional, o que torna-se uma limitação, visto que, seria desejável uma visão uniforme dos objetos a serem aplicados tanto fora como internamente ao sistema.

---

[2] *Deve-se notar que a noção de nível e módulo não necessariamente coincidem, já que módulos são usados para ocultar detalhes de projetos sobre estruturas de dados, tais como tabelas, enquanto um nível é um conjunto de funções que são implementadas via funções de níveis mais baixos*

[3] *Classes representam funções e compõem a estrutura hierárquica: subclasses implementam os detalhes de suas superclasses; subclasses podem usar as funções de suas superclasses*

## 2.3 Estrutura de Máquinas Virtuais

Uma estrutura de máquinas virtuais provê um conjunto de máquinas abstratas, que atuam separadamente e simulam uma máquina real. Por exemplo, uma simulação de uma leitora de cartões (impressoras, discos) é usada para produzir (múltiplas) leitoras de cartões virtuais (impressoras, discos). Desta forma, para cada usuário do sistema, tudo parece como se ele tivesse sua própria cópia idêntica ao *hardware* real.

Esta estrutura de máquinas virtuais é baseada na observação de que um sistema *time-sharing* provê (1) multiprogramação e (2) uma extensão de uma máquina com uma interface mais conveniente que o *hardware* puro. Entretanto, a essência do sistemas operacional VM/370 [CRE81], exemplo mais conhecido deste modelo, é completamente separada destas duas funções. O coração do sistema, conhecido como *virtual machine monitor*, executa sobre o *hardware* básico e provê o compartilhamento do mesmo entre as várias (não uma) máquinas virtuais do nível acima, como mostrado na figura 2.3.

Outra característica que difere de outros sistemas operacionais, é que as máquinas virtuais não são extensões de outras máquinas, com arquivos e outras características, mas sim cópias exatas do *hardware* básico, incluíndo modos (kernel/usuário), interrupções, enfim, todo o conjunto de recursos que a máquina real dispõe.

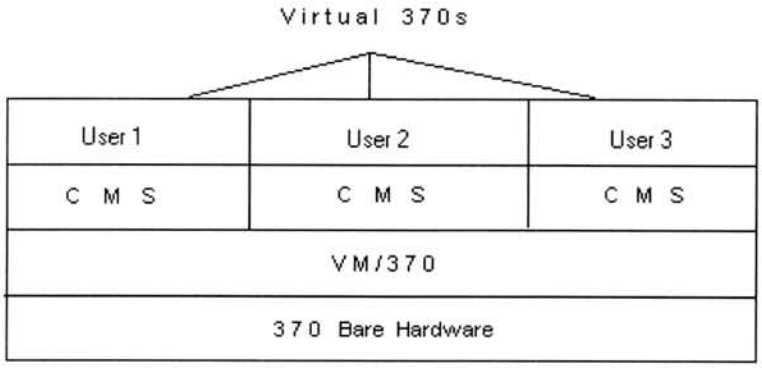


Figura 2.3 Estrutura de Máquinas Virtuais IBM

O fato de cada máquina ser idêntica ao *hardware* verdadeiro, permite que cada uma possa executar qualquer sistema operacional projetado para funcionar diretamente sobre o *hardware* real. Na verdade diferentes máquinas virtuais podem, e usualmente o fazem, executar diferentes sistemas operacionais. Por exemplo, uma delas pode estar executando o descendente OS/360 para processamento *batch*, enquanto

outra estar executando o simples, monousuário, sistema interativo chamado CMS (*Conversational Monitor System*). Note que o compartilhamento do *hardware* verdadeiro entre os usuários é provido pelo monitor.

Um dos problemas que este modelo apresenta é a tendência a degradação no desempenho do sistema operacional, visto que simulações podem ter custos consideráveis. Tecnologias mais recentes para implementação de máquinas virtuais reduzem esta degradação de desempenho através de facilidades especiais tais como VMA (*virtual memory access*).

Uma das vantagens das estruturas de máquinas virtuais é a possibilidade de implementar vários tipos de sistema operacionais; entretanto, os sistemas são disjuntos e mesmo com um suporte para comunicação através dos diferentes sistemas operacionais, cada um deles é uma entidade distinta, o que dificulta uma forma rica de interação, compartilhamento e comunicação.

## 2.4 Independência entre Políticas e Mecanismos

A separação entre políticas e mecanismos, o que Levin [LEV75] chama de princípio da independência entre políticas e mecanismos[4], é na verdade uma variante da estrutura hierárquica. Políticas são definidas em níveis externos ao kernel, enquanto mecanismos internos ao kernel provêem um conjunto de primitivas que visa permitir a definição e implementação destas políticas, significativamente diferentes, no nível mais externo (nível do usuário). Esta separação acrescenta ao sistema a flexibilidade de permitir a troca da política sem a necessidade de modificação dos mecanismos básicos.

Hidra [WUL74] é um exemplo clássico de construção onde o kernel é composto inteiramente (quase) de mecanismos que incluem capacidades básicas para proteção, criação e representação de novos tipos de objetos e, gerenciamento de tipos primitivos de objetos (procedimentos, áreas locais de dados e processos).

Deve-se notar que um importante objetivo deste conceito é permitir a construção de facilidades do sistema operacional como o programa normal do usuário, o que implica em permitir ao usuário controlar

[4] *do termo Policy/Mechanism Separation*

políticas que determinam a utilização de recursos básicos do sistema, escalonamento, proteção, etc. Hidra demonstra que várias (mas nem todas) políticas úteis e confiáveis podem ser construídas sobre cuidadosos mecanismos desenvolvidos no nível do kernel.

Apesar desta separação aumentar a flexibilidade do sistema para o usuário, uma das dificuldades é a determinação de quais mecanismos deverão ser fornecidos, de modo a torná-lo abrangente o suficiente para permitir a implementação de qualquer política do usuário. Todavia, este modelo contém conceitos importantes, conceitos estes encontrados repetidamente em vários contextos.

## 2.5 Estrutura de Kernel Coletivos

Muitos sistemas operacionais modernos são projetados como uma coleção de processos altamente independentes. Neste estilo estrutural, uma das responsabilidades chaves do núcleo (ou micro kernel) é suportar interações entre processos que implementam serviços do sistema operacional. Exemplos deste modelo incluem Chorus [HER88], V-Kernel [CHE84], Mach [BLA86] e Amoeba [MUL90].

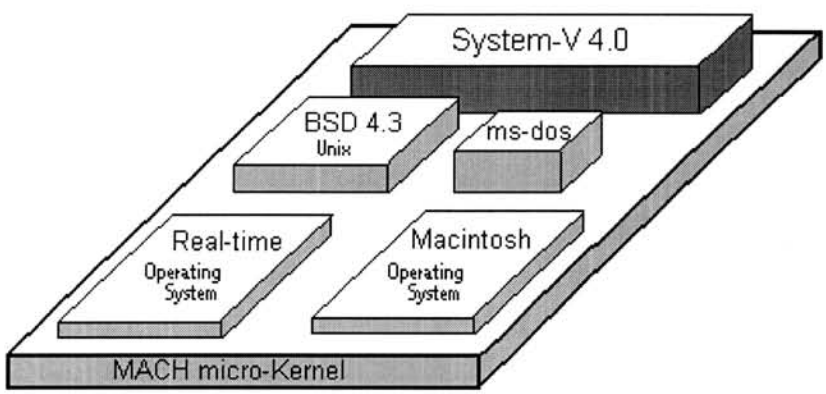


Figura 2.4 Estrutura de Kernel Coletivo de Mach

Chorus divide o sistema em três camadas: aplicações, subsistema e micro-kernel. Um programa de aplicação é uma coleção de objetos (ou atores na terminologia Chorus) que tem seu próprio ambiente de execução (ou subsistema na terminologia Chorus). Um subsistema é também definido como uma coleção de atores. Os subsistemas são suportados pelo micro kernel (ou núcleo na terminologia Chorus) que é localizado em cada

*host*. Chorus/MIX é um subsistema que simula o System-V compatível com Unix.

V-Kernel atua como um software em retaguarda, provendo transparência de rede, gerenciamento de memória e comunicação entre processos. Funções semelhantes ao gerenciamento da comunicação entre processos são implementadas como serviços do kernel dentro do micro kernel. Estas incluem gerenciamento de tempo, gerenciamento de processos, gerenciamento de memória e gerenciamento de dispositivos periféricos.

Kernel coletivo (figura 2.4) é o atual estado da arte para projeto de sistemas operacionais, explorando de alguma forma as vantagens da técnica estruturada. Por exemplo, é utilizada independência entre políticas e mecanismos, onde o micro kernel implementa mecanismos, enquanto processos levam a cabo as políticas.

Visto que a estrutura de kernel coletivos define uma forma para estruturação de sistemas operacionais (isto é, serviços do sistema operacional são implementados no topo de um micro kernel), ele necessita uma disciplina ou uma técnica tal como a estruturada ou a técnica orientada a objetos, sobre a qual o sistema é construído. Estrutura coletiva apresenta também o benefício da separação do kernel em duas partes: mecanismos que manipulam recursos do sistema (tais como memória física, espaços de enderençamento, etc.) e o paradigma de programação que auxilia os programadores a usar efetivamente o sistema [FUJ91].

## 2.6 Modelo Cliente-Servidor

Uma tendência observada nos projetos de sistemas operacionais consiste em movimentar o código para os níveis mais altos, movendo tanto quanto possível as funções do sistema operacional para dentro dos processos do usuário e deixando um kernel mínimo. No modelo cliente-servidor (figura 2.5), adepto desta filosofia, o kernel realiza basicamente a manipulação da comunicação entre clientes e servidores. Esta separação do sistema operacional em partes, cada uma das quais somente manipulando uma faceta do sistema, tais como servidores de arquivos, processos, memória, etc., produzem partes pequenas e facilmente manuseáveis.

Deste modo, para requisitar um serviço, tal como a leitura de um bloco de um arquivo, um processo do usuário (agora dito processo cliente) envia uma requisição ao processo servidor, que então realiza o trabalho e retorna o resultado ao processo cliente. Uma vantagem deste modelo é o fato dos servidores, executando em modo usuário, não terem acesso direto ao *hardware* e, conseqüentemente, se um problema ocorrer no servidor de arquivos e o servidor falhar, isto não implica que o sistema completo irá falhar.

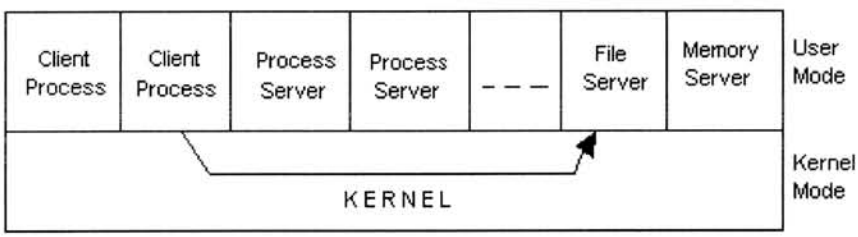


Figura 2.5 Modelo Cliente-Servidor

Outra vantagem do modelo é sua adaptabilidade ao uso em sistemas distribuídos, visto que se um cliente comunica-se com um servidor pelo envio de uma mensagem, o cliente não necessariamente necessita conhecer onde a mensagem é manipulada, se na própria máquina ou em uma máquina remota pertencente a rede.

Deve-se observar, no entanto, que a presença de um kernel que manipula somente o transporte de mensagens entre clientes e servidores não é completamente realístico, já que algumas funções do sistema operacional (tal como carregamento de comando em um dispositivo de E/S) são difíceis, se não impossíveis de serem realizadas a partir do modo usuário. Uma maneira de solucionar este problema é ter alguns processos servidores executando no modo kernel, com acesso total ao *hardware*, mas mantendo a comunicação com outros processo através do mecanismo de mensagens normal. Outra forma é a separação entre políticas e mecanismos, mantendo os mecanismos no kernel e a política de decisão no modo usuário.

Shapiro [SHA86] propôs um novo conceito baseado no modelo cliente-servidor, chamado de *princípio do procurador*[5], como tentativa para facilitar a construção de sistemas distribuídos. Neste modelo, um objeto cliente deve adquirir o objeto procurador que representa o objeto

[5] *do termo Proxy Principle*

servidor e então comunicar-se com o objeto servidor através da invocação local de objetos do próprio cliente. Quando o cliente e o servidor estão executando em diferentes máquinas, um objeto procurador é criado sobre a máquina cliente. Embora um objeto procurador conheça a localização do objeto servidor e é acessível localmente a partir do objeto cliente, ele é definido como parte do servidor.

Uma das carcterísticas desta estrutura é que ela pode ocultar diferenças entre linguagens de programação pelo suporte de objetos implementados em linguagens diferentes de uma maneira uniforme através do procurador.

## 2.7 Estrutura Baseadas em Objetos

Nas estruturas baseadas em objetos, os serviços do sistema operacional são implementados como uma coleção de objetos, os quais são definidos como segmentos protegidos, que encapsulam estados privados de informações ou dados e de um conjunto de operações associadas, através das quais os estados internos são acessados e alterados.

O kernel do sistema geralmente tem a função de gerenciamento dos objetos, gerenciamento de interações entre objetos e a responsabilidade de proteger os objetos contra acessos indevidos. Cada objeto tem associado um tipo o qual estabelece as propriedades do objeto: processos, arquivos, diretórios, etc. HYDRA [WUL74], Medusa [OUT80], Eden [ALM85], Argus [LIS87], CHORUS [BAN85], Clouds [DAS89], e Amoeba [MUL90] são exemplos de sistemas projetados com estruturas baseadas em objetos.

Uma característica da estrutura baseada em objetos, similarmente a estrutura de kernel coletivo, é que ela requer uma disciplina para organizar serviços do sistema operacional como uma coleção de objetos. Outra possível característica é a existência de uma *thread* de controle de modo que objetos coletivamente possam ser tratados como uma unidade única.

Existem diferenças significativas entre os sistemas acima citados, como estruturados com base em objetos, no que diz respeito as características dos objetos, tais como: granularidade, composição, ativação, etc. Estes e outros aspectos são enfocados com especial atenção neste trabalho no capítulo seguinte.

Deve-se notar que o modelo de objetos, não somente tem dado suporte a evolução de estruturas há muito consagradas, como tem sido usado como ponto de partida para o desenvolvimento de conceitos estruturais aos quais seus atores preferem distingüi-los, caracterizando-os como novos. Exemplos destas estruturas são introduzidos nos próximos itens, mas analisados com interesse particular no quarto capítulo.

## 2.8 Estruturas Baseadas no Conhecimento

Sistemas operacionais baseados no conhecimento[6] têm como elemento básico do modelo uma base de conhecimento, a partir da qual propõe-se a criação de um suporte mais inteligente para as aplicações e melhores ambientes para o usuário.

Como salientado anteriormente, o modelo de objetos é utilizado como ponto de partida ao desenvolvimento da estrutura baseado no conhecimento, de modo que as informações semânticas mantidas na base de dados, e tidas como a chave para prover a inteligência e a integração requerida pelo sistema operacional, dizem respeito à caracterização dos objetos e ao inter-relacionamento entre os mesmos. O modelo proposto por Nicol et al [NIC87] assemelha-se ao modelo de redes semânticas de objetos, onde o conhecimento é representado de forma explícita e concisa, novos fatos sobre objetos podem facilmente ser adicionados a rede semântica e todo o relacionamento para um dado objeto pode ser determinado sem uma pesquisa extensiva.

Cosmos [NIC89] representa um exemplo de implementação baseada no modelo, onde todos os recursos são representados como abstração do modelo de objetos, incluindo o kernel do sistema. Uma característica do modelo é a imutabilidade dos objetos, o que significa que objetos assim que criados, não podem ser alterados. Para produzir um novo objeto é necessário transformar um objeto existente pela aplicação de uma seqüência de uma ou mais operações definidas sobre o objeto. Uma característica conseqüente é a produção de um grafo de versões com a história do objeto. Segundo os autores, sistemas derivados deste modelo provêem um estrutura unificada para manipulação da distribuição, além de ser capaz de suportar ambientes de programação mais sofisticados.

---

[6] *O autor reconhece que o uso do termo "base de conhecimento" (no lugar de banco de dados) pode ser controverso, particularmente para os conceitos de inteligência artificial. Entretanto o autor demonstra que o sistema apresenta um comportamento inteligente.*

## 2.9 Estruturas Reflexivas

Sistemas reflexivos são descritos como sistemas capazes de acessar sua própria descrição e alterá-la de modo a alterar seu próprio comportamento.

Fujinami et al [FUJ91] apresentam *Muse* como um modelo capaz de superar as fragilidades de várias outras estruturas. No modelo *Muse*, um objeto é uma simples abstração de um recurso computacional do sistema e pertence a um nível de abstração mais elevado, denominado de meta-espaço, que provê um ambiente de execução.

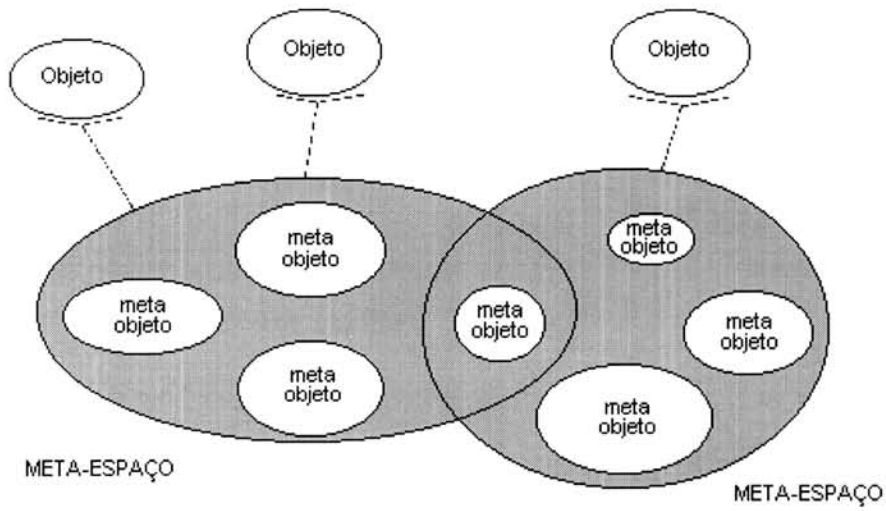


Figura 2.6 Visão conceitual da arquitetura Muse

O modelo básico de *Muse* é a programação concorrente orientada a objetos, onde aplicações, dispositivos e mesmo memória são definidos como objetos ou coleções de objetos. Neste modelo, cada objeto consiste de: Memória local, Métodos que acessam a memória local e Processador(es) virtual(is) que executa(m) os métodos.

A presença de processadores virtuais, que distingue os objetos *Muse* de outras estruturas baseadas em objetos, permite cada objeto ter seu próprio ambiente de execução e permite a introdução do conceito de *computação reflexiva* [ISH91] que provê a habilidade de alterar o comportamento dos objetos dinamicamente. Cada objeto é suportado por um ou mais meta-objetos, que constituem seus meta-espaços (figura 2.6). Um meta-espaço pode ser visto como uma máquina virtual para o objeto, ou um sistema operacional otimizado para o objeto.

A arquitetura *Muse* contém os conceitos de estrutura de kernel coletivo e estruturas baseadas em objetos, porém, diferentemente da

estrutura de kernel coletivo, um objeto é, fundamentalmente, uma entidade e diferentemente da estrutura baseada em objetos, um objeto é definido como uma ferramenta de computação reflexiva.

O nome *Muse* foi posteriomente alterado para Apertos[YOK92] com a finalidade de solucionar conflitos com registros de nomes.

**Capítulo 3**

# Gerência de Objetos

*Situar o problema através de uma análise das características intrinsicamente relacionadas com o aspecto de gerenciamento de objetos. Tais características dizem respeito a estrutura dos objetos, ao gerenciamento da interação entre objetos, e ao gerenciamento de recursos. Uma análise das soluções já propostas é realizada.*

Sistemas cuja proposta é suportar a abstração de objetos a nível de sistemas operacionais, segundo Chin [CHI91], devem ter a habilidade de permitir que objetos sejam mantidos, gerenciados e usados eficientemente. Alias, conforme destacado no primeiro capítulo, este é um dos principais desafios a serem superados pelos sistemas baseados em objetos.

Como as entidades fundamentais dos sistemas baseados em objetos são os próprios objetos é natural que a função essencial destes sistemas seja o seu gerenciamento. Tal processo, no entanto, envolve vários aspectos, tais como:

- Controle das ativações[1], de modo a tornar o efeito sobre objetos persistentes permanentes,
- Controle da execução sincronizada de múltiplas ativações concorrentes intra e interobjetos,
- Proteção dos objetos contra acessos indevidos, e recuperação de objetos na eventualidade de falhas.

## 3.1 Estrutura dos Objetos

Um fator de influência no projeto de sistemas ditos baseados em objetos diz respeito as características dos objetos suportados, ou mais especificamente sua granularidade e composição. A granularidade de um objeto é caracterizada pelo tamanho relativo, pelo custo acarretado, e pela quantidade de processamento executado pelo objeto, enquanto a composição de um objeto é caracterizada pelo relacionamento existente entre objetos e processos.

### 3.1.1 Granularidade

Conforme salientado, a granularidade de um objeto é caracterizada pelo seu tamanho relativo, pelo custo provocado, e pela quantidade de processamento executado pelo objeto. Sob estes enfoques, os objetos são distinguidos entre granularidades alta, média e fina. O caso mais simples é um sistema somente suportar objetos de granularidade alta, que são

---

[1] *Deve-se observar que o termo ativação é sinônimo de invocação para a terminologia de objetos e muitas vezes ambos os termos são usados de forma indiscriminada*

caracterizados por seus tamanhos, pelo grande número de instruções executadas para realizar uma invocação e tendo relativamente poucas interações com outros objetos.

Alguns exemplos de objetos de granularidade alta são: um programa, um arquivo, ou um usuário de um banco de dados. Tais objetos normalmente residem em seus próprios espaços de endereços, o que permite ao sistema prover proteção de *hardware* entre objetos e garantir que erros de software sejam contidos dentro do próprio objeto, a menos de falhas catastróficas.

Um objeto puramente de granularidade alta oferece a vantagem da simplicidade. Alguns prejuízos associados aos sistemas que suportam este esquema podem ser apontados:

- Os objetos de granularidade alta são entidades muito pesadas, visto que um espaço de endereçamento é provido para cada objeto.
- Controle e proteção sobre dados neste nível de granularidade restringe a flexibilidade do sistema e o grau de concorrência do objeto.
- O sistema não provê um modelo de dados consistente. Grandes entidades de dados são representadas como objetos, enquanto entidades menores são representadas como dados abstratos das linguagens de programação convencional.

De modo a prover um controle mais fino sobre os dados, é recomendável que o sistema suporte tanto objetos de granularidade alta, como objetos de granularidade média. Tais objetos podem ser criados e mantidos com custos relativamente baixos, por serem objetos menores que os de alta granularidade e possuírem alto grau de concorrência. Isto ocorre porque, vários objetos de granularidade média podem residir no mesmo espaço de endereçamento de um único objeto de granularidade alta e desta forma o acesso sincronizado aos dados pode ser feito a nível destes objetos da granularidade média. Similarmente, a quantidade de dados copiados para dispositivos de armazenamento secundário na ocorrência de uma ação pode ser reduzida. Um prejuízo de usar objetos de granularidade média é o custo adicional que deve ser imposto ao sistema para manusear um grande número de objetos.

Um controle igualmente fino pode ser provido pelo suporte de objetos de granularidade alta, média e fina. Objetos de granularidade fina

são caracterizados por serem pequenos em tamanho, executarem poucas instruções e possuírem alto grau de interação com outros objetos. Alguns exemplos de objetos de granularidade fina são os convencionais tipos de dados providos pelas linguagens de programação, tais como lógicos, inteiros e números complexos. Objetos de granularidade fina são encapsulados e residem dentro do espaço de endereços de um único objeto, seja este de granularidade média ou alta. O maior problema dos sistemas que suportam objetos de granularidade fina é o fraco desempenho obtido em decorrência do alto grau de *overhead* necessário para tornar quase toda operação uma invocação a um objeto.

### 3.1.2 Composição

A composição de um objeto é caracterizada pelo relacionamento existente entre objetos e processos. Um objeto é dito *passivo* quando os processos atrelados a ele são temporários, ou seja, existem apenas para realizar uma determinada invocação. Um objeto é dito *ativo* quando os processos atrelados a ele são permanentes.

Um aspecto inerente à invocação dos objetos é a concorrência, que pode ocorrer caso múltiplos processos executem simultaneamente dentro de um mesmo objeto, ou seja, múltiplas ativações simultâneas são permitidas. O grau de concorrência do objeto (isto é, o número de processos concorrentemente executando sobre o mesmo objeto) dependerá de fatores como o esquema de sincronização suportado e o tipo de ativação permitida.

#### 3.1.2.1 Modelo de Objetos Passivos

Em um modelo de objetos passivos, os processos e os objetos do sistemas são entidades completamente separadas. Deve-se notar que um processo não está restrito a um único objeto, podendo ser usado para realizar todas as ações requeridas para satisfazer uma dada invocação. Como conseqüência, um processo pode executar vários objetos durante sua existência (ciclo de vida). Quando um processo faz uma ativação para um outro objeto, a execução do objeto corrente é temporariamente suspensa. Conceitualmente, o processo é então mapeado para o endereço do segundo objeto, onde é executado o método apropriado. Quando o

processo completa esta operação, a execução retorna para o primeiro processo e a operação inicial é retomada(figura 3.1).

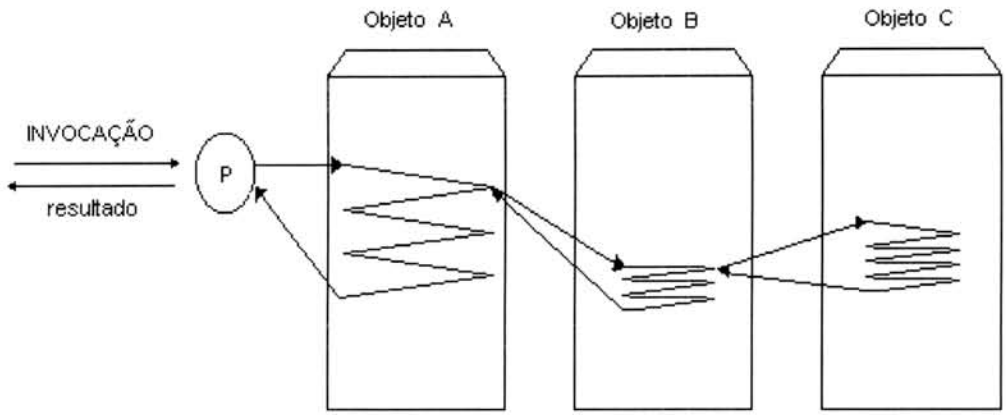


Figura 3.1 Execução de uma ativação no modelo de Objeto Passivo

Uma vantagem do modelo de objetos passivos é que, virtualmente não existem restrições sobre o número de processos que podem ser criados sobre um objeto. Uma desvantagem do modelo é que o custo de mapeamento de um processo entre múltiplos objetos, além de alto, pode ser complexo.

### 3.1.2.2 Modelo de Objetos Ativos

No modelo de objetos ativos, vários processos são criados e atribuídos a um mesmo objeto a fim de manipular as ativações solicitadas. Além dos processos serem restritos aos objetos para os quais foram criados, quando um objeto é destruído, também são destruídos todos os processos atrelados a ele.

Deve-se observar que neste modelo, uma operação não é acessada diretamente por uma chamada ao processo, como em uma chamada tradicional a um procedimento. Ao invés disto, quando um objeto cliente realiza uma invocação a um objeto servidor, um processo do objeto servidor correspondente aceita a requisição e executa o método como um representante do objeto. As interações entre clientes e servidores são da seguintes forma:

- O objeto cliente realiza uma requisição de ativação a um outro objeto especificando o método a ser ativado, juntamente com uma lista de argumentos.
- O objeto servidor aceita a requisição, localiza e executa o método especificado.
- Se durante a execução de um método, uma nova requisição de ativação é requerida, o objeto emite a requisição de ativação e aguarda o resultado. Um processo servidor em um segundo objeto é chamado para executar a nova ativação e assim sucessivamente.
- Quando a operação completar, o processo servidor retorna o resultado ao cliente.

Note que neste enfoque múltiplos objetos podem ser invocados para realização de uma simples ação (figura 3.2).

Existem duas variações para o modelo de objetos ativos:

- Um modelo dito **estático**, em que o número de processos atribuídos para cada objeto instanciado é fixo. Neste caso, quando uma ativação é feita sobre um objeto, ela é randomicamente atribuída a um processo ocioso, que executa o método especificado. No caso de todos os processos estarem ocupados, executando outras tarefas, a requisição é colocada em uma fila de requisições pendentes para serem atendidas assim que possível. Neste modelo o grau de concorrência é limitado pelo número de processos criados para cada objeto.

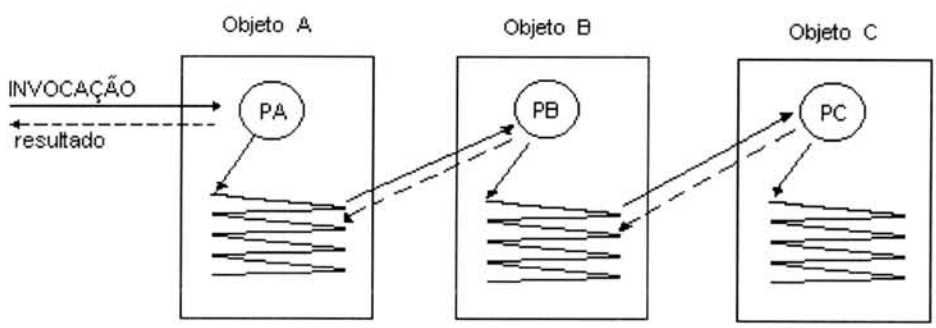


Figura 3.2 Execução de uma ativação no modelo de Objeto Ativo

- No modelo dito **dinâmico**, processos servidores são dinamicamente criados para um objeto a medida que são requeridos de modo que as

requisições nunca fiquem aguardando em filas de espera. Quando a execução de um método estiver completada, os processos que estavam executando são destruídos. Este modelo tem o custo adicional da criação e destruição dinâmica de processos. Evidentemente existem formas de minimizar este custo, por exemplo, mantendo-se um grupo de processos ociosos.

Um problema com o modelo de objetos ativos é a possibilidade de ocorrência de bloqueio perpétuo (*deadlock*), que pode acontecer caso um objeto não disponha de processos servidores suficientes para atender todas as invocações feitas a ele. Por exemplo numa recursão, onde uma única ação pode invocar o mesmo objeto mais vezes do que o número de processos servidores disponíveis, tanto o objeto como a ação serão bloqueadas. O problema de bloqueio perpétuo é menor nos modelos dinâmicos, onde os processos são criados dinamicamente, podendo continuar a ocorrer se for permitido a uma ação executar invocações arbitrariamente.

## 3.2 Controle das Ativações

Uma função importante dos sistemas baseados em objetos é controlar as atividades realizadas a partir de uma ativação sobre os objetos. Uma ativação deve ter as seguintes propriedades:

- *Serializabilidade.* Múltiplas ativações que executam concorrentemente devem ser escalonadas de tal maneira que o efeito resultante seja o mesmo da execução seqüencial.
- *Atomicidade.* A ativação de um método executa com sucesso ou não tem efeito.
- *Persistência.* O efeito de uma ativação completada satisfatoriamente não é perdida mesmo na ocorrência de falhas catastróficas.

Deve-se observar que uma ativação pode produzir várias invocações conexas e afetar múltiplos objetos. Tipicamente, uma ativação somente completa com sucesso quando todas as invocações associadas a ela terminarem. Tal ativação é dita **confiável**, enquanto ativações que falham são ditas **abortadas**. Em uma ativação confiável, todas as modificações feitas sobre objetos persistentes são gravadas em dispositivos de

armazenamento secundários, enquanto com ativações abortadas, as modificações sobre o objeto tornam-se sem efeito.

A garantia de que uma ativação dita confiável executou as alterações solicitadas sobre o objeto, permanentes ou não, é realizada por algum procedimento do sistema, sendo o mais popular o proposto por Gray em [GRA78] denominado de "*Two-phase commit protocol*". O Algoritmo deste procedimento realiza as seguintes etapas:

- Uma mensagem "pré-confiável" é enviada para todos os objetos afetados.
- Um objeto que recebe uma mensagem "pré-confiável", escreve suas modificações na memória secundária e retorna uma mensagem de "admitido".
- Para os objetos que retornarem mensagens de "admitido", uma mensagem "confiável" é enviada. Para os objetos que não responderam a mensagem "pré-confiável" dentro de um prazo estabelecido, é enviada uma mensagem "aborta" e o procedimento é terminado.
- Os objetos que recebem a mensagem "confiável" realizam as trocas permanentes, reinicializam a estrutura de controle associada com a ativação, e retornam uma segunda mensagem de "admitido". Os objetos que recebem mensagens "aborta" tornam sem efeito todas as trocas solicitadas e reinicializam a estrutura de controle.
- Mensagens "confiável" são repetidamente enviadas para o objeto que falhou, enquanto não retornarem todas as mensagens de admitido. Neste ponto todo procedimento de confiabilidade termina.

Note que este protocolo é simples e eficiente, todavia é dependente de um objeto coordenador, o que pode comprometer todo o esquema caso o mesmo venha a falhar.

## 3.3 Sincronização

Outra importante função dos sistemas baseados em objetos é garantir que atividades oriundas de múltiplas ativações, sobre o mesmo objeto, não conflitam ou interfiram entre si. Desta forma, não devem ser permitidas ativações que observem ou modifiquem o estado de um objeto

que tenha sido parcialmente modificado por outra ativação ainda não confiável. A falta desta garantia pode levar a uma condição conhecida como "aborto em cascata". Isto é, qualquer ativação que observar o estado parcial de um objeto resultante de uma ativação que mais tarde será abortada, também deverá ser abortada. Portanto, torna-se necessária a existência de algum mecanismo de sincronização para garantir que todas as ativações tenham a propriedade da serialização e para proteger a integridade do estado dos objetos. Vários esquemas de sincronização são conhecidos, alguns deles classificados como **otimistas** e outros como **pessimistas**.

Em um esquema de sincronização pessimista, o sistema previne-se contra a ocorrência de conflitos, tomando providências antes que o mesmo ocorra. Uma invocação sobre um objeto é temporariamente suspensa caso ela possa interferir em outras ativações em execução, correntemente em execução pelo objeto, e retomada quando todas as ativação conflitantes tornarem-se confiáveis. O mecanismo mais usado pelos esquemas pessimistas são bloqueios tipo Leitura/Escrita, embora semáforos e monitores também são usados.

Em um esquema de sincronização otimista, um objeto não toma nenhum providência para prevenir conflitos, ao invés disto, antes de uma invocação tornar-se confiável ela é testada quanto sua serialização, de modo a garantir que as informações observadas não conflitam com alterações feitas por outras ativação anteriormante tornadas confiáveis. Isto é, o sistema examina se a versão do dado que está sendo tratado pela ativação corrente é a mesma versão da ativação anteriormante tornada confiável. Se nenhum dado foi alterado, a ativação pode tornar-se confiável. Caso tenha ocorrido alguma modificação nos dados, a ativação deve ser abortada. A sincronização otimista permite o máximo grau de concorrência possível dentro de um objeto, visto que ativações não são nunca suspensas, como no esquema pessimista.

O maior problema com o esquema otimista é que algumas ativações terminando completamente com sucesso, podem ser forçadas a abortar. Além disso, múltiplas cópias de cada objeto devem ser mantidas na memória de modo a permitir concorrência e a troca feita para cada ativação confiável deve ser armazenada em uma memória secundária para permitir que o procedimento de confiabilidade verifique a serialização.

O esquema pessimista evita o custo de destruir e refazer requisições às custas da redução no grau de concorrência. Por exemplo, duas ações que examinam e modificam diferentes partes de um mesmo objeto não podem executar concorrentemente, mesmo quando não existem problema de conflito. O esquema otimista, por outro lado, evita o custo de retardar requisições às custas da destruição e reconstrução de requisições. Em conseqüência, o esquema pessimista executa melhor quando conflitos são freqüentes, enquanto o otimista executa melhor quando conflitos são infreqüentes.

## 3.4 Segurança

Providenciar um esquema de segurança capaz de prevenir que ativações não autorizados tenham sucesso é uma importante, mas freqüentemente ignorada, função dos sistemas baseados em objetos. Segurança é especialmente importante em sistemas multiusuários, onde são atribuídos aos usuários diferentes privilégios de acesso a diferentes conjuntos de objetos. O mecanismo de segurança pode ser provido tanto pelo sistema quanto pelo usuário.

Um mecanismo de segurança comum é o esquema de privilégios que incorpora proteção ao esquema de identificação [COH75]. A chave para tal esquema consiste de dois campos: um campo de identificação e um campo de direitos de acesso, especificando respectivamente a identificação de um objeto particular e quais métodos do objeto podem ser ativados. Cada privilégio é válido para um único objeto. O mesmo objeto pode ter múltiplos privilégios, o que permite ao proprietário do objeto estabelecer privilégios de acesso diferentes entre clientes. Cada objeto servidor passa seus privilégios entre os clientes do sistema, de modo que para um cliente fazer uma invocação a um objeto, deve primeiro possuir os privilégio do objeto e passá-los como parâmetros em uma requisição de invocação. Tanto o sistema como o objeto podem ser responsáveis pela verificação, pelo gerenciamento e pela manutenção dos privilégios de acesso, sendo que, em ambos os casos, os privilégios devem ser protegidos contra alterações não autorizadas.

Outro tipo de mecanismo de segurança é o esquema de *procedimento de controle* [BAN85]. Neste esquema, todo objeto tem um procedimento especial através do qual toda requisição de ativação deve

passar e cuja função é verificar a autorização dos clientes, liberando ou negando as requisições. Este esquema é flexível no sentido que ele pode suportar qualquer tipo de esquema de segurança, por exemplo, ele pode usar um mecanismo que garanta o não recebimento de requisições de clientes não previamente cadastrados, ou usar um esquema de senhas para bloquear requisições não autorizadas. Isto permite a cada objeto modelar sua própria segurança, de modo que se um objeto é pouco importante, um esquema de segurança mínima pode ser usado, enquanto múltiplos esquemas de segurança podem ser usados para objetos confidenciais. Note que, neste esquema, violar a segurança é extremamente difícil porque cada objeto é inteiramente responsável pela sua proteção.

## 3.5 Confiabilidade

É importante que um sistema seja capaz de detectar e recuperar falhas ocorridas, sejam elas nos objetos ou no sistema (*hardware*). Esta é uma característica recomendável especialmente em sistemas multiprocessadores e em sistemas multicomputadores, visto que a probabilidade de ocorrência de falhas cresce com o aumento do número de componentes. Existem dois métodos gerais de prover confiabilidade aos objetos. O primeiro é o da **recuperação** da falha no objeto tão rápido quanto possível, limitando o tempo em que ele fica não disponível. O segundo método é o da **duplicação** do objeto por vários processadores (ou estações de trabalho), de modo que o objeto tenha uma alta probabilidade de sobreviver a uma falha em um processador (ou estação). Técnicas de recuperação de objetos incluem esquemas de recuperação por retrocesso[2] [JAL94]. e por avanço[3] [POW83]. Esquemas de duplicação incluem cópias primárias e esquemas de objetos pares.

### 3.5.1 Recuperação de Objetos

No esquema de recuperação por retrocesso, um objeto que falhar é recuperado segundo o último estado consistente armazenado no dispositivo secundário pelo procedimento de confiabilidade. Todos os processos e ativações em execução, quando da ocorrência da falha, são

---

[2] *do termo roll-back*

[3] *do termo roll-forward*

perdidos. Este esquema de recuperação, além de simples e eficiente, apresenta alta probabilidade de recuperação com sucesso.

No esquema de recuperação por avanço, o objeto que falhar e todos os objetos a ele associados são restaurados igualmente segundo o último estado consistente. Todavia, todos os processos e ativações que estavam em execução no momento da falha são reiniciados. Este esquema é mais complexo que o esquema anterior, porque o sistema cria a aparência de que não ocorreu falha.

Devido as múltiplas formas com que um objeto pode falhar e a complexidade da interação entre objetos, não existe garantia que a recuperação será realizada com sucesso. Por exemplo, se falhar o mecanismo que armazena as interações correntes do sistema, todo o procedimento de recuperação poderá estar comprometido. Outro problema associado com este método é que todas as falhas sobre um objeto causadas pelo *software* irão ocorrer novamente quando o objeto for reiniciado, o que pode causar um ciclo contínuo.

### 3.5.2 Duplicação do Objeto

O esquema de duplicação de objetos prevê que cópias de um objeto existam em vários locais diferentes, permitindo que o sistema proporcione completa funcionalidade mesmo na ocorrência de falhas em processadores (ou estações de trabalho). Isto é verdadeiro, visto que a falha em um processador, somente resulta na não disponibilidade das cópias que residem naquele local, enquanto cópias residentes em outras estações continuam aptas a atender ativações de métodos daquele objeto. Existem vários problemas associados com o esquema de duplicação, incluíndo consistência da informação e sincronização de atividades de múltiplos clientes. Partições na rede podem causar a destruição de um sistema de duplicação.

Uma variante simples para este esquema, porém limitante, é permitir que somente objetos imutáveis sejam duplicados. Visto que os estados dos objetos imutáveis somente podem ser examinados pelos clientes, jamais modificados, este tipo de objetos pode ser duplicado sem a preocupação com consistência e sincronização.

Uma alternativa é o esquema de cópia primária, onde uma cópia do objeto é dita ser a cópia primária, enquanto as outras são ordenadas e

mantidas em estações separadas como cópias secundárias. Desta forma, ativações que não modifiquem o estado do objeto podem ser realizadas por qualquer cópia, enquanto ativações que modificam o estado do objeto devem ser atendidas pela cópia primária, que então propaga as modificações para as demais cópias secundárias. Existem duas variações para o esquema de cópias primárias: um esquema de cópia primária estática, onde as ativações que modificam o estado do objeto são suspensas até a restauração da cópia primária; e um esquema de cópia primária dinâmica, que substitui a cópia primária, no caso de falha, por uma das cópias secundárias.

Existem vários problemas associados com o esquema de cópia primária dinâmica. Primeiro, quando uma cópia secundária substitui uma cópia primária ela deve assumir a identificação e recursos, tais como portas do objeto que falhou, de modo que o cliente não perceba qualquer troca no objeto servidor. Segundo, uma partição na rede que separe a cópia primária de algumas das cópias secundárias, pode resultar na criação de dois conjuntos de objetos cada um dos quais com uma cópia primária, podendo resultar em conflitos quando da recuperação.

O terceiro esquema de duplicação é conhecido como esquema de objetos pares, onde todas as cópias são consideradas iguais, não existindo a designação de objeto primário e secundários. Neste esquema, qualquer ativação de alteração do estado do objeto não é executada por qualquer cópia, permitindo que a falha de várias cópias sejam toleradas sem impedir que ativações que modifiquem o estado objeto sejam processadas. A ocorrência de partição na rede é menos problemática. A cooperação de alguns, ou de todas as cópias, são requeridas a fim de processar as requisições.

## 3.6 Gerenciamento da Interação entre Objetos

Um sistema baseado em objetos também é responsável pelo gerenciamento da interação entre objetos cooperantes. Quando uma requisição é ativada, o sistema deve localizar o objeto especificado, tomar as providências para que o método apropriado seja executado e possivelmente retornar o resultado. Esta tarefa não diz respeito ao gerenciamento de objetos propriamente. É importante analisarmos os

mecanismos que devem ser provido para localização dos objetos, para manipulação das interações e para detecção de falhas nas ativações.

### 3.6.1 Localização dos Objetos

Um sistema baseado em objetos deve prover a propriedade de *transparência de localização*, de modo que o cliente não necessite conhecer a localização física de um objeto, a fim de invocar um de seus métodos. Quando uma ativação é realizada, o sistema deve determinar qual o objeto que está sendo invocado e qual a localização física do mesmo, a fim de entregar a ele a requisição. Note que todo objeto é identificado unicamente pelo sistema e esta identificação não pode ser alterada enquanto o objeto estiver em uso. Quando a migração de objetos for desejável, é necessário que o mecanismo provido para localização seja flexível o suficiente para permitir objetos moverem-se de um local físico para outro.

Uma forma é embutir a localização do objeto junto com sua identificação. Desta forma, quando uma invocação é realizada, o sistema simplesmente examina a identificação do objeto a fim de determinar o local físico onde o objeto reside. Este é um esquema eficiente que apresenta, porém, a restrição de não permitir que um objeto migre, visto que requer a troca da identificação do objeto, conseqüentemente, um objeto é fixo a um local físico particular ao longo de sua vida útil.

Uma segunda forma é o esquema de servidor de nomes distribuído onde o sistema cria um grupo de objetos servidores de nomes que cooperam entre si, de modo a, coletivamente, conter informações atualizadas sobre a localização de todos os objetos do sistema. Existem duas variações para este esquema:

- Na primeira, todos os servidores de nomes mantêm uma coleção completa das informações de localização, de modo que qualquer servidor pode responder uma requisição de localização. O problema que se apresenta, é que as informações mantidas nos servidores podem apresentar-se ligeiramente fora de sincronismo, visto que o processo de atualização não é uma operação instantânea e a manutenção da consistência da informação entre os múltiplos servidores de nomes pode ser difícil.

- Na segunda variação, informações parciais são mantidas em cada servidor, de modo que se uma requisição de localização não puder ser atendida por um servidor, ela é delegada para outro. O maior problema deste esquema é que no mínimo um servidor deve ser notificado toda vez que um novo objeto é criado ou movido de um local físico para outro.

Outra forma de implementar mecanismos para localização de objetos é o esquema conhecido como *cache/broadcast*. Uma pequena *cache* é mantida em cada local físico (estação ou memória) para armazenar a última localização conhecida de alguns dos objetos remotos mais recentemente referenciados.

Quando um cliente realiza uma invocação remota, a *cache* é examinada para determinar se ela tem alguma informação a respeito da localização do objeto invocado. Caso a localização seja conhecida, a invocação é enviada para aquele local; caso nenhuma informação sobre a localização do objeto está contida na *cache*, uma mensagem é distribuída pela rede questionando a localização do objeto. Toda estação (ou processador) que receber a requisição, realiza uma pesquisa interna buscando informações sobre o objeto especificado. Se o objeto é encontrado, uma mensagem retorna para a estação de origem que atualiza sua *cache*.

Este esquema pode ser muito eficiente, visto que as localizações dos objetos podem ser encontradas localmente. Ela é também flexível, visto que permite um objeto ser movido de um local físico para outro, evitando o custo de ter que notificar outras estações ou servidores de nomes distribuídos. Um problema com este esquema é que requisições *broadcast* irão perturbar toda rede, interrompendo todas as estações mesmo que somente uma delas esteja diretamente envolvida com a localização requisitada.

### 3.6.2 Manipulação de Ativações a Nível de Sistema

Quando um cliente realiza uma invocação sobre um objeto, o sistema é responsável pela execução dos passos necessários para entregar a requisição ao objeto servidor e retornar o resultado para o objeto cliente. Deve-se notar que o sistema de manipulação de invocações depende inteiramente do modelo de objeto suportado. Tradicionalmente dois

esquemas são usados: o esquema de **troca de mensagens** e o esquema de **ativação direta**.

### 3.6.2.1 Troca de Mensagem

Sistemas adotando o modelo de objetos ativos suportam tipicamente o esquema de troca de mensagens para manipular a interação entre objetos. Quando um cliente realiza uma invocação sobre um objeto, os parâmetros da invocação são empacotados dentro da mensagem de requisições. Esta mensagem é, então enviada para um processo servidor, ou para uma porta associada com o objeto invocado. Um processo servidor no objeto invocado aceita a mensagem, desempacota os parâmetros e executa a operação especificada. Quando a operação completar, o resultado é empacotado dentro da mensagem de resposta, que é então enviada de volta ao cliente.

Sistemas baseados em objetos diferem da maioria dos sistemas distribuídos em que dois processos interagindo não dispendem esforço para estabelecer e liberar conexões, por serem estáticas, porém pesadas. Em vez disto, o acoplamento entre clientes e servidores é leve e dinamicamente realizado a cada invocação. Este enfoque é mais apropriado para comunicações do tipo **requisita/responde** comum para os sistemas baseados em objetos. Também é mais apropriado para ambientes multicomputadores, porque os mecanismos requeridos para comunicação entre máquinas são mais naturalmente mapeados, o que permite aos sistemas baseados em objetos suportar migração de objetos mais facilmente. Uma desvantagem do esquema é o *overhead* da troca de mensagens para invocação entre máquinas.

### 3.6.2.2 Ativação Direta

A utilização do esquema de ativação direta é tradicionalmente utilizada nos sistemas que suportam o modelo de objetos passivos, para manipular a interação de objetos. No modelo de objetos passivos, um processo único é responsável pela execução de todas as operações associadas com uma ativação. Como resultado, um processo migra entre operações e objetos sempre que a ativação correpondente faz uma invocação.

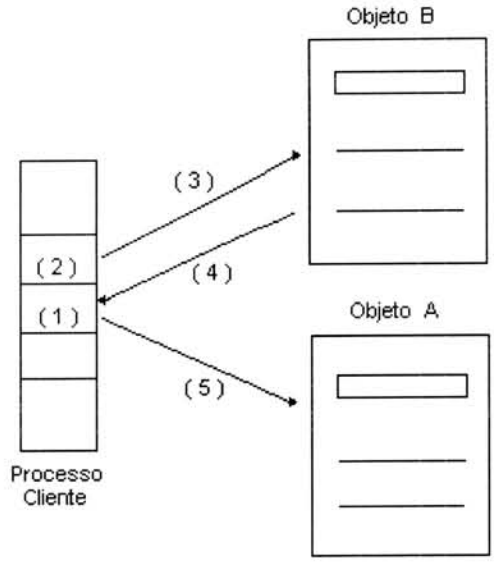


Figura 3.3 Requisição Local, Invocação Direta

Quando um processo invocar um objeto servidor residindo no mesmo local físico, são efetuados os seguintes passos (figura 3.3):

- O estado do processo e o objeto correspondente são armazenados em uma pilha de processos. O sistema pode proteger a pilha para garantir que as informações não sejam violadas por ativações subseqüentes.
- Os parâmetros da invocação são adicionados a pilha.
- O objeto invocado é carregado na memória, e uma chamada de procedimento é realizada para que o processo inicie a execução do código apropriado.
- Quando a operação termina, os resultados são retornados ao cliente e o processo é restaurado para o estado anterior a invocação.

Quando um processo invocar um objeto servidor que reside em um local físico diferente, são efetuados os seguintes passos (figura 3.4):

- Uma mensagem contendo os parâmetros da ativação são criados e enviados para o local físico (estação ou processador) onde o objeto servidor reside.

- A estação (ou processador) que recebe a mensagem cria um processo operário para executar como representante do processo original.
- Quando a operação terminar, uma mensagem contendo os resultados da invocação são criados e retornados para a estação onde o processo original reside. O processo operário é então terminado.

Uma invocação sobre um objeto local é similar à chamada de procedimento, enquanto uma invocação sobre um objeto remoto é similar a uma chamada remota de procedimento. O esquema de ativação direta pode provocar menos *overhead* que o esquema de troca de mensagens quando oriundas de invocações locais, visto que tais interações são relativamente eficientes. Por outro lado, quando proveniente de invocações remotas, o esquema de ativação direta tem o custo adicional de criação e destruição do processo operário.

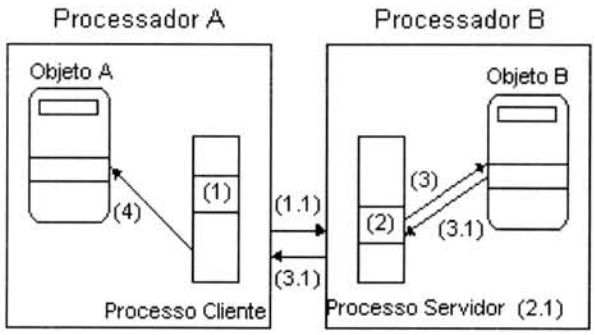


Figura 3.4 Requisição Remota, Invocação Direta

### 3.6.3 Detecção de Falhas nas Ativações

Falhas nas ativações podem ser classificadas tanto como falhas por **falta de objeto** como **falhas transientes**. Uma falta de objeto é caracterizada como uma falha que pode ser detectada antes da invocação ser iniciada, sendo que o tipo mais comum de falta de objeto ocorre quando o objeto invocado não pode ser localizado. Tais falhas são relativamente fáceis de serem detectadas e manipuladas.

Uma falha transiente ocorre enquanto a ativação está sendo executada. Podendo também ocorrer após o objeto servidor ter aceito a requisição de ativação, mas antes da ativação torna-se confiável, ou seja, antes das modificações feitas sobre o objeto completarem satisfatoriamente. Estas falhas são muito mais difíceis de serem detectadas e manipuladas, porque uma invocação pode ocorrer de muitas maneiras diferentes. O sistema deve prover mecanismos, tanto para objetos clientes como para objetos servidores, de modo a detectar e recuperar falhas transientes. Os esquemas de detecção de falhas mais comumente utilizados são os que usam o controle por tempo decorrido e os que usam o esquema de inspeção, seja através de um objeto específico, ou através de uma ativação de inspeção.

Se uma falha não é detectada pelo objeto cliente, ele e as ativações correspondentes podem ficar bloqueadas indefinidamente. Conseqüentemente, é necessário que o cliente seja capaz de detectar a ocorrência de falhas e iniciar o procedimento de recuperação, quando as mesmas acontecerem. O procedimento de recuperação tipicamente libera os recursos utilizados pelo cliente e notifica a ativação correspondente da ocorrência da falha, podendo também tentar reiniciar a ativação quando da restauração. A não detecção de falha pelo objeto servidor pode ocasionar a retenção de recursos úteis do sistema desnecessariamente. Uma ativação iniciada por um objeto que não aguarda pelos resultados é referida como órfão. Uma ativação órfão pode ser gerada tanto por uma falha no objeto cliente, como por uma transação abortada, ou por uma falha na rede.

## 3.7 Gerenciamento de Recursos

Sistemas baseados em objetos, semelhantemente a outros sistemas, devem prover mecanismos para gerenciamento de recursos físicos do sistema, incluíndo memória primária, dispositivos de armazenamento secundário, processadores e/ou estações de trabalho. Especificamente para sistemas baseados em objetos, tais mecanismos devem prover a representacão dos objetos na memória primária e secundária, para transferência de objetos entre recursos e a forma como os mesmos são atribuídos aos processadores. Esta seção analisa os aspectos de gerenciamento de recursos relacionados com objetos.

### 3.7.1 Gerenciamento de Memória Primária e Secundária

São ditos objetos voláteis aqueles objetos que são perdidos quando o local onde os mesmos residem falha. Tais objetos são temporários, residem unicamente na memória primária e são relativamente baratos de manter e usar. Objetos que têm alta probabilidade de sobreviver a falhas são ditos persistentes. Tais objetos residem na memória secundária, ainda que, uma ou mais cópias destes objetos possam residir na memória primária. Deve-se notar que as ações modificam os objetos persistentes na memória, sendo que a versão da memória secundária é atualizada somente quando a ativação torna-se confiável. Isto garante a manutenção de uma versão estável e confiável de cada objeto. Devido a estes *overheads* adicionais, os objetos persistentes são mais caros de serem mantidos que os objetos voláteis.

Um objeto persistente que reside tanto na memória primária como na secundária é dito ser um objeto **ativo**, enquanto um objeto persistente que é mantido unicamente na memória secundária é dito ser **inativo**. Quando uma invocação é realizada sobre um objeto inativo, uma cópia volátil do objeto é criada e carregada na memória para tornar o objeto ativo. Novos processos são criados para a versão volátil caso necessário. Quando o procedimento de confiabilidade executar com sucesso sobre um objeto ativo, o objeto é copiado para a memória secundária, podendo então ser desativado e a memória reutilizada pelo sistema. Este carregamento e salvamento dos objetos pára e da memória são executados de forma transparente pelo sistema, de modo a ocultar o fato de que um dispositivo de memória secundária é usado para dar ao sistema a aparência de que os objetos estão sempre disponíveis quando invocados.

#### 3.7.1.1 Representação de Objetos na Memória Primária

A forma como um objeto é representado na memória depende do esquema de sincronismo e do esquema de gerenciamento de ativações suportado pelo sistema. O esquema de sincronismo influencia o número de versões que é mantida para cada objeto, enquanto o gerenciamento de ativações tem influência sobre a representação de cada versão.

Quando um sistema baseado em objetos suporta o esquema de sincronização pessimista, uma única versão de cada objeto é mantida na memória, sendo que neste esquema, todas as ativações modificam a

mesma versão volátil do objeto. Quando o esquema de sincronização otimista é suportado, múltiplas versões de cada objeto são criadas e mantidas na memória, sendo que neste esquema, a toda ativação é atribuída sua própria versão volátil do objeto sobre a qual as modificações são realizadas. Isto permite que múltiplas ativações para o mesmo objeto sejam realizadas simultaneamente, garantido que não haverá interferência entres elas.

A representação de cada objeto na memória depende da necessidade ou não de suportar o esquema de transações aninhadas. Em um sistema que suporta esquema de requisições ou esquema de transações, a forma tradicional de representar uma versão na memória é uma cópia exata do objeto correspondente. Caso uma ação falhe, a versão volátil pode simplesmente ser descarregada. Este enfoque não é adequado quando o esquema de transações aninhadas é suportado, visto que múltiplas sub-ativações de uma ativação podem modificar o mesmo objeto.

O problema é que as trocas realizadas sobre o objeto não são tornadas permanentes enquanto o nível superior da ativação não estiver confiável e todas as ativações decorrentes completarem ou falharem de forma independente umas das outras. Se uma subativação modifica um objeto e é forçado a abortar, a troca realizada deve ser desfeita e o objeto restaurado ao estado anterior a execução da subativação. Conseqüentemente, um mecanismo adicional é necessário para desfazer as trocas realizadas pela subativação que falhou. Este esquema deve incluir o uso de *logs* desfaz/refaz ou manter um objeto standard imutável com a versão criada pela última subativação completada satisfatoriamente.

#### 3.7.1.2 Representação de Objetos na Memória Secundária

Em um sistema baseado em objetos, é imperativo que sejam mantidas na memória secundária informações suficientes para que um objeto persistente possa ser restaurado a um estado consistente, no caso da ocorrência de falhas. O sistema pode armazenar o estado completo de um objeto sempre que ele tornar-se confiável (esquema de *checkpoint*), ou ele pode simplesmente armazenar as trocas realizadas no objeto a partir de algum estado previamente armazenado (esquema de *logs*).

No esquema de *checkpoints*, o estado completo de um objeto modificado é armazenado na memória secundária quando a ativação

correspondente tornar-se confiável. Esta operação é realizada em dois estágios. Em um estágio pré-confiável, a versão modificada do objeto é armazenada na memória secundária sem destruir a anterior. Em um segundo estágio a versão modificada substitui a versão antiga. Caso a ativação abortar, a versão modificada é destruída, permanecendo a versão antiga inalterada.

Uma vantagem deste esquema é o uso eficiente da memória secundária, visto que somente uma cópia de cada objeto é mantida. O problema é o *overhead* executado, já que o estado completo do objeto modificado é armazenado sempre que uma ativação tornar-se confiável. Este problema torna-se mais crítico quando somente pequenas alterações são realizadas sobre objetos com um estado muito grande. Outra desvantagem é manter somente o estado mais recente, o que torna inviável o suporte de esquemas de controle de concorrência otimista, visto que o mesmo requer um conhecimento histórico do objeto, a fim de que o teste de serialização possa ser realizado.

Uma variante deste esquema representa os objetos persistentes na memória secundária, como uma coleção de pontos de verificação[4]. Ao invés de destruir a versão antiga de um objeto quando uma nova versão é armazenada, a nova versão é adicionada a uma seqüência de pontos de verificação, permitindo que versões anteriores sejam examinadas. Uma desvantagem deste esquema é o espaço de memória adicional requerido e a outra desvantagem é a introdução do problema de determinar quando uma versão antiga pode ser removida.

No esquema de *Logs*, sempre que um objeto persistente for modificado, as trocas relativas são armazenadas em um *log* comum na memória secundária. A quantidade de informação armazenada na memória secundária dependerá de um esquema particular, todavia o custo e o overhead de armazenamento é claramente menor que o esquema de pontos de verificação. Deve-se armazenar informação suficiente de modo que, um objeto que falhar, possa ser restaurado ao último estado confiável. Uma desvantagem deste esquema é a quantidade de *overhead* de manutenção e recuperação dos objetos.

Uma variante mais simples é conhecida como *redo log*. Neste esquema, um ponto de verificação básico para todo objeto persistente é

---

[4] *do termo checkpoint*

armazenado, de modo que as alterações são realizadas a partir deste ponto e do status corrente das ativações que realizaram estas alterações. Quando um objeto é modificado e as ativações que realizaram as modificações tornarem-se confiáveis, as alterações realizadas sobre o objeto são armazenadas num *log*. Por exemplo, um *redo log* pode armazenar um novo valor de um dado atualizado ou pode armazenar operações realizadas sobre o objeto. Caso um objeto falhar, ele é restaurado segundo o último estado consistente pela execução das modificações realizadas sobre o ponto de verificação básico. Periodicamente um novo ponto de verificação é armazenado para cada objeto e os *logs* antigos são liberados.

### 3.7.2 Gerenciamento dos Processadores

Administrar o uso dos processadores é uma função muito importante do sistema. O objetivo básico é maximizar a taxa de *throughput* (quantidade de processamento) do sistema, minimizando o tempo que um objeto tem que esperar para ser alocado a um processador. A tarefa de atribuir objetos a processadores é dificultada por dois objetivos conflitantes: primeiro, objetos devem ser atribuídos a diferentes processadores, levemente carregados, de modo a executarem concorrentemente; segundo, objetos que interagem freqüentemente devem ser atribuídos ao mesmo ou a processadores próximos de modo a reduzir o custo de comunicação. Assim, o benefício de executar objetos de um programa em múltiplos processadores é contrabalanceado pelo custo adicional da comunicação entre processadores. Para obter um desempenho ótimo, os objetos, de um programa baseado em objetos, devem ser atribuídos a um grupo de processadores levemente carregados e próximos.

#### 3.7.2.1 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 reatribuído para qualquer processador do mesmo tipo ao qual ele foi originalmente criado. A exceção é encontrada nos objetos imóveis, cuja localização é codificada dentro da identificação do objeto.

Tais objetos são sempre reatribuídos ao mesmo processador ao qual ele foi originalmente criado.

O esquema de escalonamento de objeto pode ser explícito ou implícito. No esquema explícito, o usuário é responsável pela especificação do processador para o qual o objeto deverá ser alocado. No esquema implícito, o sistema é responsável pela determinação do processador ao qual o objeto deve ser alocado. Existem poucos algoritmos práticos para escalonamento implícito de objetos. Esquemas mais complexos examinam a carga dos processadores para determinar a melhor localização para colocar um objeto, tradicionalmente o processador escolhido é o de menor carga. Quando um grupo de objetos que interagem entre si é criado, tal como num programa baseado em objetos, é buscado um grupo de processadores levemente carregados e cada objeto é alocado a um processador. Uma desvantagem de usar informações sobre carga para determinar onde atribuir objetos é que a obtenção e manutenção de informações apuradas sobre a carga dos processadores pode ser relativamente dispendiosa. Caso as informações não sejam apuradamente mantidas, podem ser atribuídos alguns objetos a processadores fracamente carregados e os subseqüentes serem sobrecarregados.

#### 3.7.2.2 Migração de Objetos

Um esquema de migração de objetos permite objetos moverem-se ou migrarem de um processador para outro a qualquer momento, em alguns casos mesmo quando em meio a uma invocação. O trabalho executado por um objeto que migrou não deve ser perdido, nem mesmo qualquer tentativa de ativação do objeto deve ser abortada. A vantagem da migração de objetos é aumentar o desempenho e diminuir a ociosidade. Por exemplo, objetos podem ser movimentados de um processador altamente carregado para um processador mais levemente carregado. O objetivo da migração pode também habilitar objetos que interagem fortemente serem movimentados para o mesmo processador a fim de diminuir o custo de comunicação.

A migração e execução de processos de um processador para outro normalmente é uma tarefa convencional dos sistemas distribuídos, mas extremamente difícil. Ainda que não seja difícil mover o código em execução, o problema pode estar em mover as informações dependentes

da máquina, tais como valores de relógio, caminhos lógicos de comunicação e estruturas de dados mantidas na memória. Um sistema baseado em objetos simplifica estes problemas porque os objetos definem claramente as entidades que podem ser movimentadas e encapsula os componentes que devem ser movidos como unidades. Além disto, informações dependentes de máquina normalmente são mínimas e a propriedade de transparência de localização permite ao sistema determinar a nova localização de um objeto que foi automaticamente movido.

No entanto, existem vários problemas associados com a migração de objetos. Primeiro, o custo de migrar um objeto pode ultrapassar o benefício da migração. A migração de um objeto de um local para outro pode ser uma tarefa árdua, e um objeto não pode executar enquanto está sendo movimentado. Segundo, requisições de invocação enviadas para um objeto enquanto está sendo movimentado, devem ser aceitas pelo sistema e enviadas ao objeto quando ele tornar-se ativo novamente. Terceiro, um objeto não deve ser continuamente movimentado, sob pena de processar muito pouco. Finalmente, cópias de um objeto não devem ser movidas para o mesmo processador.

Mecanismo de migração de objetos são tentativas de reduzir a carga sobre os processadores. Quando a carga de um processador excede algum limite, o sistema tenta encontrar um ponto de equilibrio com um, ou mais, processador menos carregado, para o qual são movidos alguns objetos ativos. A migração, preferencialmente, deve ocorrer para processadores próximos, o que minimiza a distância da movimentação do objeto e mantém o objeto relativamente próximo aos objetos com os quais o mesmo provavelmente interage. Caso seja encontrado o processador adequado, o sistema determina quais objetos devem migrar, sendo esta decisão normalmente baseada no tamanho do objeto, no tempo de processamento estimado, no número de vezes que o objeto já migrou, ou no custo da migração. A migração de um objeto normalmente segue a rotina de ser suspenso, migrado e então reativado.

## 3.8 Comentários

O projeto de sistemas operacionais é uma tarefa complexa e difícil, sendo que a funcionalidade e configuração a ser provida pelo sistema operacional devem ser cuidadosamente pensadas e planejadas, não

existindo um conjunto de características pré definidas para resolver as necessidades de todos os sistemas operacionas baseados em objetos. Ao invés disto, muitas das características providas pelos sistemas operacionais dependem das aplicações. A tarefa de desenvolvimento de um sistema operacional é simplificada pelo uso de objetos, visto que objetos são entidades autônomas e que servem como unidades para proteção, recuperação, segurança, sincronização e mobilidade.

A primeira vantagem de usar um sistema baseado em objetos é que ele alivia muitos dos problemas associados com criação e execução de programas. A abstração de objetos serve como uma ponte entre o programador e o sistema, criando uma primitiva comum que reduz a complexidade da interface homem/máquina, sendo esta uma característica fundamental destes sistemas. Ele também simplifica a interface do sistema operacional para proporcionar a execução eficiente. Segundo Chin [CHI91], o modelo de objetos é uma tendência em direção a: "ao invés do usuário obedecer às necessidades do computador, o computador deve ser construído para obedecer as necessidades de seus usuários".

**Capítulo 4**

# Reflexão sobre Objetos

*Sintetiza os conceitos de reflexão computacional de adotados no modelo proposto de modo a fornecer a base através da qual o modelo estrutural proposto foi projetado e implementado.*

## 4.1 Introdução

A noção de reflexão em sistemas orientados a objetos é mais explorada nas linguagens de programação, por exemplo, CLOS e Smalltalk apresentam a habilidade de realizar processamento sobre si mesmas e em particular de estender, em tempo de execução, a própria linguagem. Estas propriedades são alcançadas através do uso de meta-classses. Cada objeto no sistema tem um meta-objeto que descreve a própria classe, através do qual o programador pode alterar o comportamento da classe.

Pode-se (superficialmente) descrever um sistema reflexivo como um sistema capaz de acessar sua própria descrição e alterá-la de modo alterar seu próprio comportamento. A consolidação deste processo ocorre em três estágios distintos:

1. O primeiro estágio, conhecido como reificação[1], consiste em obter uma descrição abstrata do sistema tornando-a suficientemente concreta para permitir operações sobre ela.
2. No segundo estágio, a reflexão computacional[2], utiliza esta descrição concreta para realizar alguma manipulação.
3. Finalmente, no terceiro estágio, é modificada a descrição reificada com os resultados da reflexão computacional[3], retornando a descrição modificada ao sistema. Desta forma, as operações subseqüentes refletirão as alterações efetuadas sobre a descrição reificada do sistema.

Uma análise aprofundada dos mecanismos e do uso da reflexão foge ao escopo deste trabalho, todavia uma sintese conceitual[4] da reflexão auxiliará no entendimento da proposta.

O conceito de Patti Maes [MAE87] sobre reflexão no modelo de orientação a objetos é que a reflexão computacional representa a atividade executada por um sistema quando faz computações sobre (e possivelmente afetando) suas próprias computações.

---

[1] *Traduzido do termo reification, utilizado na literatura sobre computação reflexiva.*

[2] *Reflexão computacional, representa o processamento realizado pelo sistema sobre si mesmo.*

[3] *Deve-se notar que reflexão computacional não implica necessariamente em alteração do sistema.*

[4] *Parte significativa desta síntese conceitual foi extraída de relatório interno da UFRGS/RS produzido pela professora Maria Lúcia Blanck Lisboa.*

Para Steel [STE94], a reflexão computacional é a capacidade de um sistema computacional de interromper o processo de execução (por exemplo, quando ocorre um erro), realizar computações no meta-nível e retornar ao nível de execução traduzindo o impacto das decisões, para então retomar o processo de execução .

Nesta definição é introduzido o conceito de níveis de computação, visto que a ocorrência de um evento (por exemplo, um erro) no processo de execução reflete-se em um nível superior (meta-nível) de computação. Para realizar computações sobre si mesmo, o sistema necessita manter conhecimento sobre si mesmo, tanto da sua estrutura de dados quanto da descrição das ações a serem realizadas sobre estes dados.

## 4.2 Arquitetura Reflexiva

A reflexão computacional define uma arquitetura em níveis, denominada arquitetura reflexiva, composta por um meta-nível, onde se encontram as estruturas de dados e as ações a serem realizadas sobre o sistema objeto, localizado no nível base. Assim, em uma arquitetura reflexiva, um sistema computacional possui dois componentes interrelacionados: um subsistema objeto, que faz computações sobre um domínio externo ao sistema, e um subsistema reflexivo, que faz computações sobre o sistema objeto. Os dados do nível base são usados no meta-nível para a realização de computações reflexivas que podem interferir nas computações de nível base.

A interação entre níveis exige uma conexão entre o componente reflexivo e o sistema objeto. Esta conexão do componente reflexivo (meta-nível) que o mesmo tenha conhecimento sobre o sistema objeto (nível base). Estas informações variam de acordo com os objetivos da reflexão computacional: o componente reflexivo pode atuar sobre a estrutura ou sobre o comportamento do sistema objeto.

No modelo de orientação a objetos, um sistema computacional é composto por uma coleção de objetos. Cada objeto possui uma estrutura estática, descrita em sua classe, e capacidade de realizar computações sobre seus próprios dados. A realização de uma computação inicia quando um

objeto recebe uma mensagem indicando a computação a ser realizada e fornecendo os argumentos necessários à realização. Em uma computação não reflexiva, o objeto realiza computações que dizem respeito ao domínio da aplicação.

A computação reflexiva pode ocorer tanto a nível de classes quanto a nível de objetos. No primeiro caso, o meta-nível é composto por meta-classes, as quais contém informações sobre os aspectos estruturais do nível base: descrição de variáveis e de métodos que irão compor todas as suas instâncias. Esta estrutura pode ser alterada e, em decorrência disso, todas as instâncias também serão alteradas. No segundo caso, o meta-nível é composto por meta-objetos, que contém informações sobre os aspectos de comportamento dos objetos de nível base, como por exemplo, como um determinado objeto trata uma mensagem. As seções seguintes descrevem mais detalhadamente estes conceitos.

## 4.3 O Modelo de Meta-Classes

Algumas linguagens orientadas a objetos utilizam meta-classes para descrever a estrutura de todas as suas classes, no sentido de que as informações necessárias para a construção de uma classe se encontram em sua meta-classe. Mais especificamente, meta-classes podem ser assim definidas, com base em [BOO94], [GRA89] ou, simplesmente, uma meta-classe é a classe de uma classe; se a classe é obtida por instanciação de uma meta-classe, então uma classe é tratada como um objeto. A propriedade de ser instanciável como uma nova classe permite que classes possuam capacidade de realizar computações sobre seus próprios dados, fornecendo serviços a seus clientes. E também as distinguem de classes tradicionais (que descrevem a estrutura de objetos), visto que o seu domínio é a própria classe; seus dados referem-se a nomes de métodos, instâncias, heranças e seus métodos podem acessar esses dados. Existem diversas possibilidades de estruturação de linguagens com meta-classes:

1. Objetos são classes

Todas as classes podem ser vistas como objetos e todos os objetos podem ser vistos como classes. Se um objeto pode ser visto como uma classe, isto significa que o próprio objeto contém informações sobre a sua estrutura, dispensando a existência de uma classe superior (a meta-classe) para descrever a estrutura dos objetos. Exemplo: a linguagem Self, onde cada objeto contém a sua própria descrição.

2. Classes são instâncias de uma única meta-classe

Todos os objetos são instâncias de uma classe e todas as classes são instâncias de uma classe ancestral comum, denominada de meta-classe. A meta-classe fornece uma interface para a estrutura de todas as classes. Isto permite que as classes sejam consideradas como objetos do programa acessíveis ao programador.

3. Todas as classes possuem meta-classes

Uma classe sempre possui uma meta-classe e todas as meta-classes possuem uma classe ancestral comum. Portanto, existem diversos níveis de classes: (a) a classe de um objeto; (b) a meta-classe da classe do objeto; (c) a meta-classe, da qual foi especializada a meta-classe; (d) a Meta-classe; (e) a classe da Meta-classe. Exemplo: a linguagem SmallTalk.

Em linguagens que estruturam as suas classes a partir de meta-classes, estas propriedades são herdadas pelas classes, de forma que todas as classes são especializações de suas meta-classes. Na árvore de herança, a meta-classe é a super-classe de todas as classes e, conseqüentemente, as classes de todos os objetos são descendentes diretas de alguma meta-classe. Isto significa que a alteração em alguma meta-classe da árvore de herança pode ser percebida por todos os descendentes da árvore de herança.

A possibilidade de alterar a meta-classe, refletindo-se em todas as suas classes descendentes pode ser restringida na implementação da linguagem.

## 4.4 Reflexão Estrutural de Classes

Meta-classes, quando acessíveis aos usuário, permitem a realização de reflexão estrutural, que pode ser assim definida, com base em [FER89]. Informações e transformações típicas de reflexão estrutural são: (a) obter informações sobre a classe x: sua classe ascendente, suas descendentes, suas instâncias, seus métodos e interfaces; (b) alterar a classe x: modificar suas variáveis e seus métodos; e ainda, (c) atuar sobre classes: criar novas classes, eliminar classes existentes e renomear classes.

## 4.5 O Modelo de Meta-Objetos

Neste modelo, a reflexão computacional se realiza por associação de um objeto de nível base a um objeto de meta-nível. O objeto de meta-nível, denominado de meta-objeto, contém meta-informações sobre os objetos de nível base; pode-se dizer que um meta-objeto, ao referir-se a um objeto, representa um outro objeto, denominado de referente. Objetos e meta-objetos são interconectados de tal forma que uma modificação em qualquer um deles provoca efeitos no outro, de forma dinâmica. Esta associação causa-efeito é conhecida como conexão causal.

Genericamente, o conceito de meta-objeto é assim colocado por Foote [FOO92]: meta-objetos são objetos que definem, implementam, dão suporte ou participam de qualquer maneira da execução da aplicação, ou objetos de nível base .

O modelo de meta-objetos se diferencia do modelo de meta-classes, principalmente por sua associação por objetos e não por classes de objetos. Qualquer objeto da aplicação pode ser associado a um ou mais meta-objetos, que definem, implementam ou participam de diferentes formas da execução dos objetos de nível base. Suas principais características são:

- Individualidade: a cada objeto de nível base pode ser associado um meta-objeto. O objeto associado a um meta-objeto é aqui denominado de objeto reflexivo. Do ponto de vista de estruturação de aplicações, isto significa que podem ser selecionadas determinadas instâncias de classes para a

realização de reflexão computacional, permanecendo não-reflexivos outros objetos instanciados a partir das mesmas classes.

- Separação de Classes: a classe do meta-objeto é distinta da classe de seu referente, permitindo que objetos de uma mesma classe sejam associados a diferentes meta-objetos; inversamente, meta-objetos instanciados a partir de uma mesma classe podem ser associados a objetos de nível base de diferentes classes.
- Reflexão comportamental: a classe de um meta-objeto contém informações sobre seu referente, incluindo suas variáveis e seus métodos. Portanto, o comportamento se um objeto de nível base pode ser facilmente modificado pelo seu meta-objeto.
- Associação dinâmica: a associação entre um meta-objeto e um objeto é feita em tempo de execução. Quando um objeto reflexivo é instanciado, ocorre também a instanciação de seu meta-objeto. Dependendo do protocolo de associação (MOP- Metaobject Protocol), é possível mudar o meta-objeto associado de forma dinâmica. Esta característica permite que um objeto assuma diferentes comportamentos ao longo do processo de execução, dependendo das classes dos meta-objetos sucessivamente associados a ele.
- Definição circular: um meta-objeto é um objeto, visto que é obtido por instanciação. Portanto, um meta-objeto pode ser tratado como um objeto comum e, inclusive, pode ter seu próprio meta-objeto. Esta característica permite a estruturação de uma torre de reflexão computacional.

Das colocações acima destacam-se como importantes as seguintes características: um meta-objeto é um objeto instanciado a partir de uma classe. Este objeto descreve alguns aspectos (não necessariamente todos) de um outro objeto ao qual se refere, e participa do processo de execução de seu objeto referente. Em síntese, um meta-objeto pode ser definido como um objeto que representa aspectos estruturais e comportamentais de um objeto O, a ele conectado.

Na definição de uma arquitetura reflexiva, Ferber [FER89], destaca as seguintes questões:

1. quais entidades devem ser transformadas em algo que possa sofrer operações no meta-nível?

2. como o relacionamento entre o nível base e o meta-nível é implementado?

3. quando o sistema passa para o meta-nível?

A primeira questão refere-se aos dados que podem ser manipulados no meta-nível. A atividade computacional do nível base é transformada em dados para o nível superior, determinando quais computações podem ser realizadas. Esta operação de transformação é denominada de reificação. Com base em [FER89], [NAK92] [MAD92], reificação é a transformação de atributos de um programa orientado a objetos em dados disponíveis ao próprio programa.

Os objetos reificados constituem as meta-informações sobre as quais são realizadas as computações reflexivas. Aqui, novamente, dependendo do objetivo da reflexão, podem ser determinadas quais meta-informações são necessárias.

A segunda (2) e a terceira (3) questões acima indagam sobre quem e como inicia o processo de reflexão. Maes[MAE88] aponta duas soluções: a responsabilidade pode ser atribuída ao objeto de nível base, que neste caso contém código mencionando seu meta-objeto ou pode ser atribuída ao sistema. Neste último caso, o meta-objeto é ativado pelo sistema quando ocorre algum evento envolvendo o objeto referente, como uma mudança de estado de variáveis de instância (variáveis reflexivas) ou o recebimento de uma mensagem dirigida a um método (método reflexivo).

Na interceptação de mensagens, todas as mensagens enviadas ao objeto, ou mais especificamente, a algum método reflexivo do objeto, são delegadas ao seu meta-objeto, passando este a ser o responsável pelo tratamento da mensagem. A realização da computação no meta-nível depende de informações dinâmicas, obtidas durante o processo de execução. O mecanismo de interceptação de mensagens fornecido pelo protocolo de meta-objetos (MOP- Metaobject Protocol) adotado na linguagem de programação reflexiva deve fornecer as seguintes informações básicas:

- Sobre o método: interface do método da aplicação destinatário da mensagem.
- Sobre a chamada: argumentos que o cliente forneceu na mensagem.

A reificação destes componentes possibilita que o meta-objeto re-envie a mensagem ao objeto da aplicação para a execução do método original.

## 4.6 Reflexão Comportamental de Objetos

Na reflexão comportamental de objetos, também conhecida como reflexão computacional, a função básica do meta-objeto é explicitar como o objeto reage diante de uma mensagem, possibilitando a intervenção no estado da computação. Esta intervenção é a essência da computação reflexiva dinâmica, buscando coletar e registrar informações sobre o processo de execução, a exemplo de estatísticas sobre desempenho, informações para fins de depuração e monitoração da execução, ou com a finalidade de modificar o curso do processo de execução, como por exemplo determinar qual computação deve ser feita a seguir, ativar computações alternativas (através de monitores e 'daemons').

No contexto deste trabalho, os objetos controlados por meta-objetos são denominados objetos reflexivos. Para realizar a reflexão comportamental de objetos, são necessárias poucas informações sobre a sua estrutura dos objetos [MAL92] ; acesso a informações estruturais sobre objetos individuais e reificação de métodos na forma de objetos são suficientes. Seguindo esta linha de reflexão, pode-se afirmar que:(a) é possível obter distintas granularidades de reflexão comportamental; (b) a reflexão comportamental pode ser realizada sem interferir no encapsulamento do objeto.

Como o comportamento de um objeto é ditado por seus métodos, a reificação individualizada de uma mensagem que ativa um método e seu conseqüente tratamento pelo meta-objeto, faz com que a reflexão seja restrita ao método chamado, ou seja, torna-se uma reflexão computacional particular de cada método e não de todo o objeto. Os métodos de um objeto reflexivo são divididos em dois grupos, os reflexivos e os não-reflexivos, onde somente os primeiros têm a sua execução controlada pelo meta-objeto.

A propriedade de encapsulamento é essencialmente estática e assegura que um objeto seja visível apenas por suas interfaces, tornando invisível a sua implementação interna (suas variáveis de instância e seus métodos). Na

reflexão comportamental realizada por interceptação de mensagens, esta propriedade não é alterada sob o ponto de vista do programa de aplicação, visto que a estrutura interna do objeto permanece inviolada, apenas aspectos de seu comportamento podem ser substituídos externamente. Ao contrário da reflexão comportamental, a reflexão estrutural viola o encapsulamento do objeto, visto que pode substituir seus métodos por outros métodos, e mesmo alterar a classe de uma instância já existente.

## 4.7 Torre de Reflexão

A arquitetura reflexiva que admite diversos meta-níveis é dita ser uma torre de reflexão: cada meta-objeto pode ter um ou mais meta-objetos a ele associado. Meta-objetos, ao serem considerados objetos, podem enviar e receber mensagens de outros meta-objetos. Quando um meta-objeto recebe diretamente mensagens de outros meta-objetos, realiza-se uma computação reflexiva do sistema.

Cada nível da torre de reflexão constitui um domínio Di, considerado como domínio base do domínio $D_{i+1}$, seu meta- domínio situado no meta-nível imediatamente superior. Cada domínio Di é, ao mesmo tempo, o domínio base do domínio do nível superior $D_{i+1}$, e o meta-domínio do domínio $D_{i-1}$, situado no nível inferior, exceto por $D_0$, constituído por referentes que podem ser usados apenas no nível base [ANC95].

Conceitualmente, a torre de reflexão é infinita, na qual cada meta-nível define o comportamento do nível base imediatamente inferior. Porém, devido à complexidade e ao custo computacional da torre reflexiva, o número de níveis não deve ultrapassar de três: o nível da aplicação, o meta-nível e o meta-meta-nível.

## 4.8 Comentários

A reflexão computacional permite fazer computações sobre uma computação. Em linguagens orientadas a objetos, a reflexão computacional é realizada através de meta-classes ou meta-objetos, que se distinguem por sua

abrangência; a reflexão realizada por meio de uma meta-classe altera todas as instâncias da classe, enquanto que a reflexão realizada por meio de um meta-objeto refere-se a uma única instância. Em ambas as formas, o objetivo é fornecer informações sobre a representação das propriedades das classes ou objetos da aplicação.

Protocolos de meta-objetos fornecem ao programador da aplicação uma interface ao compilador da linguagem usada na aplicação, permitindo a programação em meta-nível. A programação em meta-nível justifica-se quando é necessário dispor de informações especiais sobre a representação do programa, de forma independente do domínio da aplicação.

Capítulo 5

# O Modelo de Estrutura Reflexiva

*Descreve o modelo de estrutura conceitual proposto como base da implementação de sistemas operacionais multiprocessados. O modelo está baseado no conceito de computação reflexiva e visa a busca de um ambiente adequado ao paradigma de orientação a objetos .*

## 5.1 Introdução

O surgimento do modelo de objetos introduziu novas necessidades aos ambientes de execução de modo a suportar o conceito. A solução normalmente adotada nestes sistemas para suportar a abstração das linguagens orientadas a objetos é construir uma camada sobre o núcleo (run time) com a finalidade de implementar o gerenciamento de objetos. Esta solução foi adotada por exemplo em COOL [LEA93], entretanto os resultados alcançados e demonstrados por Roger Lea mostram que o principal problema ainda existente residiu exatamente na incompatibilidade entre os mecanismos do sistema e os modelos das linguagens.

Um novo modelo de estrutura conceitual para construção de sistemas operacionais foi proposto juntamente com o projeto Aurora[ZAN93a], com o objetivo de contribuir na busca de uma plataforma adequada ao paradigma de objetos e visando proporcionar através da exploração eficiente das arquiteturas multiprocessadoras, um suporte de software para a modelagem de ambientes apropriados à execução de modelos orientados a objetos.

O capítulo primeiro deste trabalho apontou a necessidade de evolução da atual tecnologia de gerenciamento de objetos, neste mesmo capítulo, uma análise das soluções já propostas evidenciou questões não resolvidas no escopo dos sistemas operacionais, como por exemplo o problema de gerenciamento de objetos, entre outros.

Partidário da colocação de autores já referenciados, nossa opinião é que o principal problema no gerenciamento de objetos é causado pela diferença entre as abstrações providas pelo sistema operacional e aquelas oferecidas pelas linguagens de programação, isto porque a maioria dos sistemas operacionais existentes não foram projetados para suportar as linguagens de programação modernas, particularmente as linguagens orientadas a objetos.

Nossa convicção da existência de um problema de conceituação nos levou a busca de um novo modelo de estrutura conceitual, visto que todos os aspectos do ambiente computacional, desde sua concepção estática de projeto, modelagem, e até sua evolução dinâmica estão baseados nos conceitos e abstrações definidos a nível do modelo de estrutura conceitual.

Salientamos também na seção 1.2 a existência de diversos aspectos do ambiente computacional que estão diretamente relacionados com o modelo de abstração.

A figura 5.1 apresenta estes aspectos, a saber:

a) o modelo conceitual sobre o qual os sistemas operacionais são construídos;

b) a utilização dos recursos do sistema operacional pelas linguagens de programação;

c) o aspecto espelhado pelo modelo de código gerado pelos compiladores e esperado pelo sistema operacional;

d) o aspecto, que diz respeito aos ambiente de programação e interfaces.

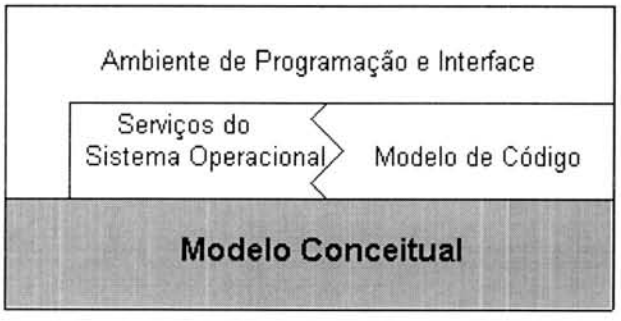


Figura 5.1 Aspectos do Ambiente Computacional

Deve-se notar que este é o contexto de um ambiente computacional, ou seja, de um sistema operacional. O escopo deste trabalho está limitado ao aspecto do modelo conceitual. Todavia, ao concentrarmos nossos esforços na busca de uma solução para este aspecto, estamos abordando o coração do problema, visto que o modelo conceitual se constitui na abstração base a partir da qual o sistemas operacional e, por conseguinte, todo o ambiente computacional é modelado e construído. O projeto Aurora representa a consolidação de um ambiente computacional baseado no modelo desta proposta onde são explorados todos os aspectos referenciados. A seção 5.2, a seguir, apresenta o modelo proposto neste trabalho.

## 5.2. Descrição do Modelo

Salientamos anteriormente a necessidade de evolução da atual tecnologia de gerenciamento de objetos e o surgimento de Aurora vem ao encontro deste afã de contribuir na busca de um modelo adequado ao paradigma. A sustentação do modelo aqui proposto reside na convicção de que a busca de um modelo adequado ao suporte de objetos deve necessariamente levar em consideração a uniformidade entre as abstrações providas pelo sistema operacional e pelas linguagens de programação.

A base deste modelo é a reflexão computacional, usada para estruturar o sistema operacional Aurora, objetivando os seguintes benefícios:

- O modelo reflexivo proporciona a construção de ambientes computacionais onde todos os níveis podem ser baseados na mesma abstração conceitual.
- Permite fazer uma clara separação entre os mecanismos de suporte a objetos e a política que estabelece como estes mecanismos são utilizados.
- Introduz simplicidade ao modelo conceitual, provendo uma maneira simples para alterar o comportamento de todos os aspectos do sistema, de modo que se tudo no sistema é suportado através de reflexão, então todos os aspectos do sistema podem ser reificados e manipulados.
- Reflexão contribui para a construção de um ambiente verdadeiramente aberto e adaptável. Como o ambiente do sistema operacional evolui, pode-se evoluir o sistema operacional e manipulá-lo.

Esta seção apresenta o modelo desta proposta, onde primeiramente ele sintetiza as abstrações básicas envolvidas, para então descrever o modelo conceitual propriamente dito.

### 5.2.1. Abstrações Básicas do Modelo

Esta seção descreve as abstrações básicas utilizadas no modelo sob o ponto de vista do sistema operacional. Os aspectos do ambiente computacional acima ressaltados, podem ser enfocados igualmente sob três pontos de vista distintos: sob o ponto de vista do sistema operacional, do programador e do usuário. Todavia, o ponto de vista do usuário pode ser representado pelas abstrações do programador, enquanto o ponto de vista do programador pode ser representado pelas abstrações do ponto de vista do sistema operacional.

Aurora utiliza o modelo de objetos como entidade fundamental para alcançar esta uniformidade de perspectiva, de modo que objetos são as únicas entidades presentes em todos os níveis do ambiente, modelando desde aplicações do usuário, serviços do sistema operacional e mesmo recursos do sistema.

Conceitualmente, um objeto é na essência uma entidade que encapsula um estado privado de informações ou dados e um conjunto de operações ou procedimentos que manipulam estes dados. O modelo de *meta-objetos* [FER89] introduziu o conceito de separação entre objetos e *meta-objetos*, onde *meta-objetos*, no *meta-nível*, definem, implementam, dão suporte ou participam de alguma maneira da execução dos objetos no nível base. Aurora, apoiado na idéia de que o nível de objetos e o nível de abstração das linguagens podem ser separadamente descritos e representados, ampliou o conceito de separação propondo a existência de entidades para representar estados e de entidades para representar a abstração da classe, isto é, englobando características semânticas e computacional.

#### 5.2.1.1 *Objetos, Meta-Objetos* e *Meta-Espaços*

De uma forma simplificada, objetos no modelo podem ser descritos como entidades que representam estados, ou seja, são as entidades criadas para modelar as abstrações da informação, ou dados. *Meta-objetos,* por sua vez, podem ser descritos como as entidades que descrevem o comportamento dos *objetos,* ou seja, são as entidades criadas para modelar as abstrações funcionais.

Neste sentido cada *objeto* no sistema, representa uma peça da funcionalidade do ambiente e possui um *meta-objeto* associado que descreve seu comportamento. Devemos observar que conceitualmente o relacionamento estado/comportamento é uma relação um-para-um. A generalização deste relacionamento para uma relação um-para-vários, permite que cada *objeto* tenha associado a ele um grupo de *meta-objetos*. Este grupo de *meta-objetos* é chamado de *meta-espaço* e representa o conjunto de funcionalidades disponíveis aos objetos.

A figura 5.2 apresenta as abstrações descritas; nela pode ser visualizado um simples objeto, cujo comportamento está definido pelo conjunto de *meta-objetos* que compõem o *meta-espaço*. Um *meta-espaço* pode ser visto como um sistema operacional dedicado a execução do objeto, visto que todo o comportamento do objeto está representado dentro dele pelo conjunto de *meta-objetos*.

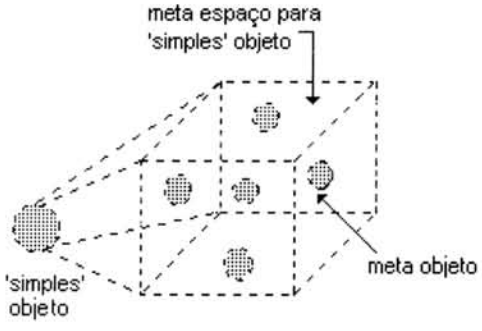


Figura 5.2. Visualização de um 'simples' objeto

5.2.1.2 Meta-Hierarquia

O fato de objetos serem as únicas entidades presentes em todos os níveis do ambiente, implica que *meta-objetos* sejam de fato também objetos, possuindo igualmente um comportamento definido que é representado por um conjunto de *meta-objetos* associados. Esta uniformidade de abstrações produz um relacionamento hierárquico em tempo de execução potencialmente infinito. Este relacionamento conhecido como meta-hierarquia ou torre de reflexão [FER89] está visualizado na figura 5.3, onde pode ser visto um segundo *meta-*

*espaço* associado a um *meta-objeto* que já pertence a outro *meta-espaço.*

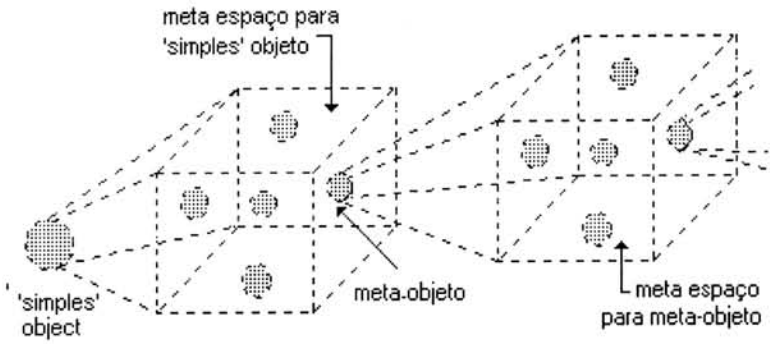


Figura 5.3 Visualização da Meta-Hierarquia

Deste modo, cada nível da meta-hierarquia constitui um nível $N_i$, considerado como nível base do nível $N_{i+1}$, seu meta-nível imediatamente superior. Por sua vez o nível $N_i$ é, ao mesmo tempo meta-nível do nível $N_{i-1}$, situado no nível inferior, exceto para o nível inicial $N_0$.

### 5.2.2. Dinâmica do Modelo

O resultado da aplicação de programação reflexiva às abstrações conceituais acima descritas foi a concretização de uma estrutura voltada ao suporte de modelos orientados a objetos e cuja característica notável é proporcionar um ambiente computacional adaptável, ou seja, onde o comportamento do sistema pode ser alterado de uma forma sistemática e controlada. Uniformidade de abstrações é outra característica encontrada, sendo o *objeto* a única entidade presente para representar cada peça da funcionalidade no sistema.

Conceitualmente, um objeto é dotado de propriedades estáticas e dinâmicas. Afora os aspectos de modelagem afeitos às linguagens de programação, a nível do ambiente de execução (essência desta proposta) podemos caracterizar as propriedades estáticas do objeto através do conceito de instância [NIE89], enquanto as propriedades dinâmicas (relativas ao tempo de vida dos objetos) estão diretamente relacionadas aos aspectos de gerenciamento dos objetos. Tais

aspectos dizem respeito principalmente (mas não somente, vide capítulo 3) ao conceito de ativação [NIE89].

É importante salientar que a abstração conceitual de *meta-hierarquia* acima descrita, absorve tanto o aspecto estático quanto o aspecto dinâmico do modelo de objetos. O aspecto estático é conseqüência da criação de uma instância do objeto, a partir da qual novos *meta-espaços* são criados (ou modificados) para representar *objetos* e *meta-objetos*. O aspecto dinâmico é influenciado pela ativação de objetos, através da qual *objetos* são alterados em função de seu comportamento[1] e *meta-espaços* são modificados em função da reflexão computacional[2].

Em tempo de execução, a hierarquia de *meta-espaços* é modelada segundo as necessidades da aplicação até o infinito teórico. Este comportamento evolutivo expressa na verdade o aspecto dinâmico da arquitetura. A noção de *meta-espaço* terminal é utilizada como ponto de partida desta recursão potencialmente infinita. Visto que o *meta-espaço* provê recursos e gerenciamento de recursos para objetos, em tempo de execução existirão várias hierarquias dependentes do *meta-espaço* em questão; entretanto, um *meta-espaço* que é comum a todos eles é o *Meta-Core*.

*Meta-Core* pode ser considerado como um *meta-espaço* terminal e ser levemente associado ao conceito de *micro-kernel* implementado em sistemas operacionais como MACH [BLA86] e CHORUS [HER88]. Entretanto, ao invés de prover um conjunto de recursos, *Meta-Core* provê um conjunto mínimo de funcionalidades visando implementar unicamente suporte a reflexão computacional do modelo *objeto/ meta-objeto.*

Uma possível representação da arquitetura abstrata descrita pode ser visualizada na figura 5.4. Nela pode ser visto o *meta-espaço* terminal *Meta-Core* que atua como base `a construção de uma hierarquia de *meta-espaços* que implementam suporte ao modelo conceitual, suporte ao modelo computacional e suporte a aplicação. Deve-se notar que a organização hierárquica da figura 5.4 é

---

[1] *Note que uma ativação a um objeto não necessariamente modifica o objeto.*

[2] *Reflexão computacional não necessariamente implica em alteração no comportamento do objeto*

hipotética, visto que a mesma é construída dinamicamente e segundo as necessidades da aplicação, dependente, portanto de implementações e aplicações particulares e não do modelo da estrutura conceitual.

As operações básicas implementadas pelo *meta-objeto* terminal (*Meta-Core*) para suportar o modelo *objeto/ meta-objeto* realizam basicamente duas funções:

1. Realizar a meta-computação, isto é, suspender a execução do *objeto* e transferir o controle para o *meta-objeto*, isto é, para o meta-nível.
2. Retornar da meta-computação, isto é, transferir o controle da execução do meta-nível para o *objeto*.

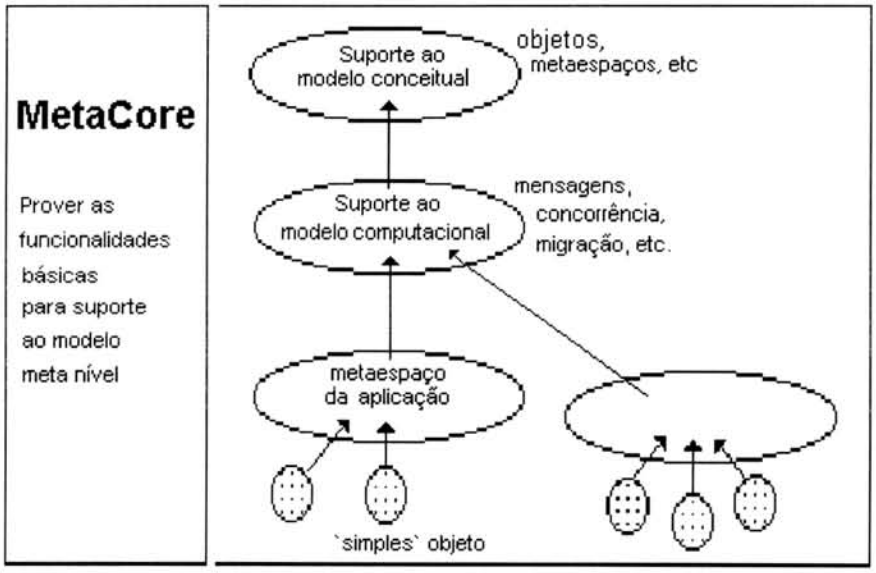


Figura 5.4 Organização hierárquica de *meta-espaços*

A implementação destas operações pode ser analogamente comparada ao conceito de primitivas dos sistemas operacionais tradicionais, estando as mesmas acessíveis a todos os *meta-espaços* existentes no ambiente computacional. Entretanto, ao invés de serem ativadas toda vez que algum serviço do sistema operacional é necessário, estas operações são executadas para a efetivação da reflexão computacional, transferindo o controle da execução do nível do objeto para o meta-nível ou vice versa.

É importante perceber que estas primitivas implementam de forma intrínseca ao modelo proposto, o mecanismo chave da reflexão computacional que é a efetivação do relacionamento entre objetos e *meta-objetos* no nível do modelo conceitual.

Todavia, ao mesmo tempo em que as primitivas suportadas por *Meta-Core* permitem a existência de um sistema reflexivo em todos os níveis, tornam-se limitantes sobre si mesmo. O fato de *Meta-Core* ser um *meta-espaço* terminal implica em que ele não possui meta-nível e em conseqüência a reflexão computacional sobre ele não pode ser realizada através de suas próprias primitivas. A alternativa para dotar *Meta-Core* de reflexão computacional é a utilização do conceito de meta-classes.

Para melhor entendermos a dinâmica do modelo, vamos analisar em particular como o relacionamento *objeto/ meta-objeto* é levado a cabo. Este relacionamento representa a reflexão computacional, visto que todo o processo de reificação[3] é realizado no meta-nível.

#### 5.2.2.1. Instanciação de Objetos

Conceitualmente, a instanciação de um objeto significa a criação de uma instância de uma classe particular de objetos. Deste modo, o processo de criação de um objeto implica acessar as informações da classe a fim de determinar a estrutura semântica da mesma e concretizá-las na forma de objeto.

O fato deste processo ser levado a cabo no meta-nível requer a transferência do controle da execução do nível de objeto para o nível do *meta-objeto*. Esta transferência é realizada através das primitivas do *Meta-Core*. A partir da efetivação do processo de instanciação do objeto, o mesmo passa a fazer parte de um *meta-espaço* do ambiente computacional, podendo ser ativado por outros objetos do sistemas, ou seja, ser escalonado para executar.

Deve-se notar que a instanciação de um *objeto* por definição conceitual do modelo de *meta-objetos* requer que a instanciação do

---

[3] *A definição conceitual de reificação é encontrada no capítulo 4*

*meta-objeto* que implementa seu comportamento seja realizada de forma simultânea. Este processo é garantido pelo *meta-espaço* que implementa suporte ao gerenciamento de objetos, discutido mais adiante.

A figura 5.5 mostra de uma forma gráfica a dinâmica de uma instanciação. Nela pode ser visualizada a meta-computação realizada através do *meta-espaço* terminal *Meta-Core.* para efetuar o relacionamento *objeto/ meta-objeto,* isto é, transferir o controle da execução do objeto para o *meta-objeto,* e retornar a execução ao *objeto.*

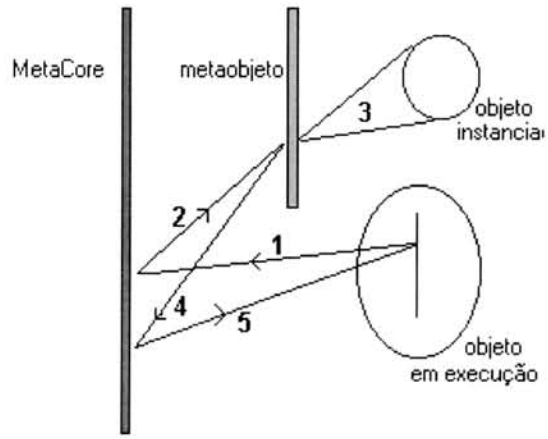


Figura 5.5 Dinâmica da Instanciação de objetos

Deve-se notar que o número de *meta-objetos* envolvidos na instanciação de um *objeto* é indeterminado, visto depender por exemplo da necessidade ou não de criação de um novo *meta-espaço.* Este processo ocorrerá no caso da classe do objeto instanciado não pertencer a nenhum dos *meta-espaços* existentes no sistema.

O processo de instanciação de um objeto é mostrado de forma algorítmica abaixo:

***Início*** { Processo de instanciação de um objeto }
"Localizar a classe do objeto";
***Caso*** <classe não pertence a um *meta-espaço*>
"ativar criação do *meta-espaço*";
"inserir o *meta-objeto* no *meta-espaço*".
"inserir o objeto no *meta-espaço*";
***Fim.***

As facilidades básicas que devem ser providas para gerenciamento de *meta-espaços* estão descritas na tabela 5.1. Tais facilidades são utilizados por exemplo na instanciação de um objeto e, após determinada a necessidade de criação de um novo *meta-espaço* para suportar o objeto.

| Facilidades básicas |
| --- |
| Criar um *meta-espaço* |
| Destruir um *meta-espaço* |
| Incluir/Excluir um *meta-objeto* ao *meta-espaço* |
| Incluir/Excluir um objeto ao *meta-espaço* |
| Examinar o conteúdo do *meta-espaço* |

Tabela 5.1. Gerenciamento de *meta-espaços*

5.2.2.2. Ativação de Objetos

Conceitualmente, a ativação de um objeto significa o envio de uma mensagem do *objeto* origem ao *objeto* ativado. Na proposta deste modelo todo o processamento realizado para fora do objeto é transferido para o meta-nível, de modo que o *meta-espaço* deve prover a implementação de uma base para mensagens.

Quando uma mensagem é enviada, o controle da execução é transferida do objeto que executa o envio da mensagem para o *meta-objeto* que implementa o tratamento de mensagens. Este processo de transferência da execução do *objeto* para o *meta-objeto* é realizado igualmente através das primitivas de meta-computação no *Meta-Core.*

A figura 5.6 visualiza a dinâmica de uma ativação. Nela pode ser visualizada a meta-computação realizada através do *meta-objeto* terminal *Meta-Core,* para efetuar o relacionamento *objeto/ meta-objeto* e a transferência do controle da execução para o objeto ativado. Uma das possíveis funções visualizadas no meta-nível como

conseqüência do envio de mensagens é gerenciar a localização do objeto ativado.

Deve-se notar que a transferência da execução não necessariamente implica na execução imediata da mensagem, visto que condições de escalonamento, competição e sincronização poderão retardar sua execução até que alguma condição seja satisfeita. Todavia, este é um aspecto dependente de implementação e, conseqüentemente, foge ao escopo do modelo conceitual.

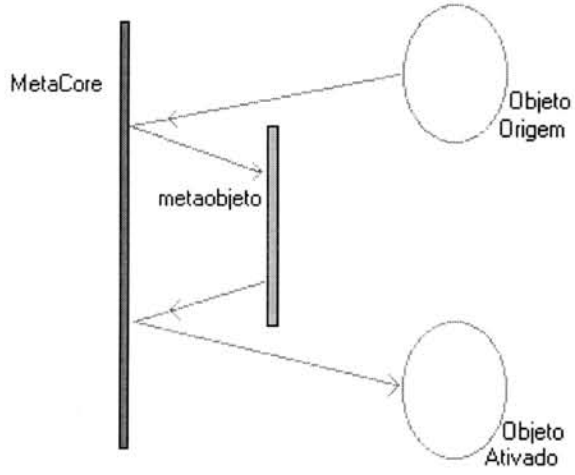


Figura 5.6 Dinâmica da ativação de objetos

### 5.2.3 Suporte a multiprocessamento

A fim de incorporar a exploração do paralelismo das arquiteturas multiprocessadas de forma eficiente e sem descaracterizar o modelo reflexivo alcançado, nós incorporamos ao *Meta-Core* um novo conceito ao qual denominamos de *primitiva multinodo*.

Este conceito significa que o processamento da primitiva pode continuar sua execução em um processador diferente daquele que iniciou a execução[4]. Esta habilidade pode ser vista na figura 5.7, onde

[4] *Tradicionalmente, primitivas são executadas de forma atômica, até onde vai nosso conhecimento, execução em múltiplos processadores é um conceito inédito para primitiva.*

a execução do retorno da meta-computação é transferida para um ambiente remoto.

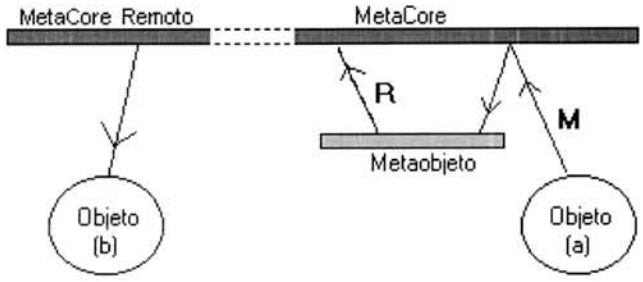


Figura 5.7 Visualização do mecanismo de *primitiva multinodo*

O número de nodos envolvidos no processamento de uma primitiva R é indeterminado, visto depender de fatores como a topologia de comunicação da rede de processadores. Todavia, deve-se notar que tarefas como a determinação da localização do nodo onde o objeto destino reside, é implementado fora do Meta-Core, o que o torna independente de políticas e abstrações.

Mecanismos de *hardware* para comunicação entre nodos, todavia são conhecidos única e exclusivamente pelo Meta-Core, o que torna este aspecto de certa forma dependente de implementação, entretanto factível de ser implementado eficientemente, visto encontrar-se no nível mais baixo do sistemas operacional.

## 5.3 Comentários

Este capítulo apresentou o novo modelo de estrutura conceitual para implementação de sistemas operacionais multiprocessados. Características notáveis podem ser facilmente identificadas no modelo, e entre estas salientam-se:

- O modelo provê naturalmente recursos ao modelo de *meta-objetos*, permitindo a construção de sistemas operacionais reflexivos em todos os níveis do ambiente.

- Provê uma clara separação entre objetos e os mecanismos de controle dos objetos, permitindo a construção de modelos flexíveis de sistema operacional multiprocessados, onde o ambiente pode ser modelado de acordo com as características da aplicação paralela.

- Mecanismos de baixo nível para gerenciamento da interação entre objetos, minimizando o tempo de latência das comunicações, anseio de toda aplicação paralela sobre multiprocessadores.

- Uniformidade no tratamento de recursos, permitindo a inexistência de distinção entre as abstrações providas pelo sistema operacional e aquelas visualizadas pelo usuário.

**Capítulo 6**

# Validação do Modelo

*Detalha os componentes de uma modelagem orientada a objetos para validação do modelo conceitual descrito no capítulo 5.*

## 6.1 Enfoque da validação

No capítulo anterior foi apresentado o modelo conceitual. Este capítulo descreve os componentes de uma modelagem orientada a objetos para validação do modelo conceitual proposto. É importante observar que a modelagem aqui apresentada é utilizada no escopo de Aurora e portanto representa uma visão particular de projeto e implementação.

Todavia, o objetivo deste capítulo é mostrar a simplicidade, abrangência e o poder do modelo como ferramenta para a construção de sistemas operacionais. São apresentadas ainda algumas repercussões da utilização do modelo sobre os aspectos envolvidos (figura 5.1) na construção de ambientes computacionais. A seção 6.2 apresenta a modelagem e implementação do modelo, a seção 6.3 apresenta uma visão superficial do projeto Aurora, na seção 6.4 é apresentada a validação do modelo.

## 6.2 Implementação do modelo

É fundamental para o entendimento desta proposta ter uma visão clara dos limites entre o modelo conceitual e o ambiente do sistema operacional. Tal delimitação nem sempre é visível a um contato superficial com um ambiente computacional, todavia é fundamental para a percepção dos limites de abrangência desta implementação.

A concretização da *meta-hierarquia* abstrata apresentada na figura 5.4 é mostrada na figura 6.1. Nela estão caracterizados *meta-espaços* que implementam abstrações do modelo conceitual, abstrações do modelo computacional e o *meta-espaço* que representa o ambiente de execução dos objetos da aplicação. Podemos observar também a presença de *activity*. Uma *activity* é similar ao modelo de *thread* enquanto a CPU é abstraída em uma estrutura conhecida como *context*.

É importante observar que, baseada no conceito de computação reflexiva, a hierarquia de *meta-espaços* é construída de acordo com as

necessidades da aplicação. Por exemplo, $m^{Zero}$ somente conterá o *meta-objeto* $m^{nth}$ se o ambiente necessitar suporte a objetos concorrentes.

Por definição do modelo conceitual, apresentado no capítulo anterior, o ponto de partida da *meta-hierarquia* de *meta-espaços* está baseado em um *meta-espaço* terminal (figura 6.1) chamado de *Meta-Core*. Conforme discutido, este *meta-espaço* terminal em tempo de execução atua de forma similar ao modelo de micro-kernel, provendo todavia um conjunto mínimo de funcionalidades que visam implementar unicamente o suporte ao modelo *objeto/ meta-objeto,* suporte a processamento paralelo e tratamento de interrupções.

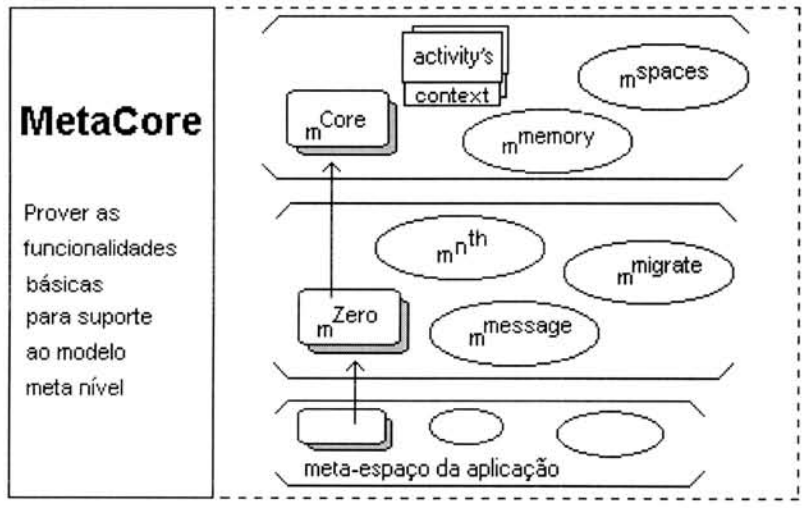


Figura 6.1 Concretização da *Meta-Hierarquia*

### 6.2.1 Suporte a meta-computação (*MetaCore*)

A figura 6.2 apresenta a modelagem do ambiente em termos do relacionamento entre o *meta-objeto* terminal, *Meta-Core*, e o modelo *objeto/ meta-objeto.* Na figura podemos visualizar como a interação entre objetos é realizada através 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* através da primitiva M, provocando a transferência do controle da execução para um *meta-espaço* no meta-nível.

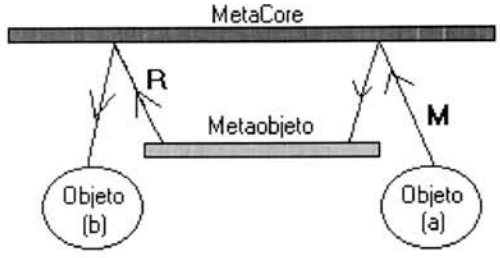


Figura 6.2 interação entre objetos via *Meta-Core*

Este modelo proporciona uma implementação através 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, enquanto uma descrição completa da modelagem e implementação da classe *Meta-Core* pode ser encontrada no anexo 1.

```
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 metaobjeto. Significado e definição dos parâmetro da primitiva são mostrados a seguir:

*Meta-Activity* contém informações que identificam o metaobjeto no metanível.

*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 metanível para o nível do objeto. Significado e definição do parâmetro da primitiva é mostrado 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 do *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 do *meta-espaço* na meta-

hierarquia, de modo que o *Meta-Core* nada conhece a respeito dos objetos que existem no sistema.

### 6.2.2 Gerênciamento de objetos

$m^{Core}$ representa o *meta-espaço* modelado para inplementar suporte ao modelo conceitual, de modo que todo aspecto relativo ao gerenciamento de objetos é realizado neste nível visto que todo o ambiente compreende em entidades do tipo objetos. Deste modo $m^{Core}$ é responsável pela identificação, instanciação e destruição dos objetos do ambiente, independentemente dos mesmos representarem objetos da aplicação ou do sistema.

Deve-se notar que conceitualmente existem outros aspectos envolvidos no gerenciamento de objetos tais como; segurança, confiabilidade, gerência de recursos e outros. Todavia tais aspectos são dependentes de implementação, isto é, fazem parte do nível do sistema operacional e conseqüentemente estão além do escopo do modelo conceitual.

Um aspecto relevante a observar é que o modelo de objetos está baseado em entidades e interações, de modo que a interação entre objetos representa um aspecto fundamental em ambientes orientados a objetos. Devemos notar todavia que o modelo proposto absorve o aspecto de interação entre objetos naturalmente através das primitivas M e R implementadas a nível de *Meta-Core*.

#### 6.2.2.1 Estrutura do Objeto

Características estruturais dos objetos foram discutidas na seção 3.1. Na modelagem aqui descrita, a noção de objeto é abstraída através de *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 usado para

representar os objetos em execução e contém a seguinte representação básica:

```
class Activity {
protected:
        CPUContext      Context;
        SID*            Identify;
        Activity*       Meta;
        EntryTable*     Execqueue;
};
```

As informações associadas a *Activity* tem o seguinte significado e atribuições:

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

Deve-se notar que Aurora implementa no nível do sistema operacional uma clara distinção entre a instanciação de um objeto e a sua ativação, de modo que instanciação e ativação são separados em *meta-espaços* distintos. $m^{Core}$ é ativado na instanciação dos objetos, de modo que no momento em que um objeto é instanciado, os mecanismos responsáveis pelo gerenciamento de *meta-espaços*,(*meta-objeto* $m^{spaces}$) são ativados a fim de verificar no sistema a existência de um *meta-objeto* que represente o modelo instanciado. Caso encontrado, o objeto é inserido no *meta-espaço* a que o *meta-objeto* pertence, caso contrário um novo *meta-espaço*, contendo o objeto instanciado e o *meta-objeto* referenciado, é inserido no sistema.

A tabela 6.1 descreve as principais facilidades providas pelo gerenciamento de *meta-espaços*. Tais facilidades são utilizadas na instanciação de um objeto e, após determinada a necessidade de criação de um novo *meta-espaço* para suportar o objeto. Deve-se

notar que os mecanismos responsáveis pelo suporte a multiprocessamento também são ativados, a fim de determinar sobre qual processador o novo *meta-espaço* será criado.

| Primitiva | Função |
|---|---|
| NewMetaSpace | Criar um objeto descritor de meta-espaço |
| KillMetaSpace | Deletar o objeto descritor de meta-espaço |
| NewMetaObject | Acrescenta um novo meta-objeto ao meta-espaço |
| NewObject | Implementa criação dinâmica de objetos |
| ShowMetaSpace | Examinar o conteúdo do meta-espaço |
| CheckMetaSpace | Utilizado para localizar objetos e/ou meta-objeto |

Tabela 6.1. Gerenciamento de meta-espaços

A descrição da interface da classe $m^{space}$ é encontrada abaixo, enquanto sua implementação completa pode ser encontrada em anexo.

```
class mspace {
        mError NewMetaSpace ( MetaAtivity* mSpaces );
        mError KillMetaSpace ( MetaAtivity* mSpaces );
        mError NewMetaObject ( MetaAtivity* mObject );
        mError NewObject ( Ativity* object );
        bool ShowMetaSpace( MetaAtivity* mSpaces );
        bool CheckMetaSpace ( MetaAtivity* mSpaces );
}
```

### 6.2.3 Migração

Suporte à migração é um aspecto básico e fundamental do modelo, visto o ambiente alvo ser sistemas operacionais e sendo uma das metas desta proposta proporcionar mecanismo de interação entre a aplicação e o sistema mais próximos do modelo de objetos, visualiza-se o conceito de herança de escopo como o mecanismo para a efetivação desta interação. Deste modo a migração é implementada

através de *meta-objetos* "standard" de modo que um objeto pode automigrar para outro *meta-espaço* quando desejar.

A migração tem efeito tanto sobre o *meta-espaço* fonte, onde o *objeto* deixa de existir, como sobre *meta-espaço* de destino onde o *objeto* passa a existir. Deste modo a migração é executada tanto pelo *meta-espaço* fonte como pelo *meta-espaço* destino. Deve-se notar que o *objeto* pode ele próprio decidir sua própria migração, sendo esta automigração ativada da seguinte forma:

```
EsteObjeto. migrar ( memória_secundária );
```

Enquanto o pseudocódigo do método que executa a auto-migração no *meta-espaço* fonte tem o seguinte formato:

```
método migrar ( destino ) {
        Próximometa := Atualmeta.acha ( destino );
        = ok {Próximometa.transfere ( objeto_descritor )
        } retorna ( falha )
}
```

## 6.3 Visão Geral de Aurora

A utilização do modelo de estrutura conceitual apresentado, na construção de sistemas operacionais permite o surgimento de modelos onde todos os níveis e aspectos do ambiente estão voltados para a mesma abstração conceitual. Aurora foi projetado sob este enfoque e buscando propagar os benefícios do modelo para todos os níveis do ambiente computacional.

A conseqüência direta foi o surgimento de novos modelos de implementação nos diferentes aspectos do ambiente computacional apresentados na figura 5.1. Estes novos modelos estão detalhados durante a descrição de Aurora nesta seção.

A arquitetura resultante da utilização do modelo de estrutura reflexiva sobre Aurora pode ser vista como definida dentro de uma hierarquia de *meta-espaços*, onde, no topo da hierarquia, está implementado o suporte ao modelo estrutural, a partir do qual, o sistema operacional, utilitários e aplicações do usuário são construídos. A implementação deste suporte é realizado através de *meta-espaços* que implementam basicamente mecanismos para: gerenciamentos de *meta-espaços,* suporte a ativações, a migração e multiprocessamento.

A figura 6.3 apresenta uma visão simplificada deste modelo onde o suporte ao modelo estrutural constitui o topo da hierarquia. Deve-se observar que alguns *meta-espaços* não possuem objetos associados. Esta particularidade ocorre principalmente nos *meta-espaços* que implementam basicamente serviços do sistema operacional.

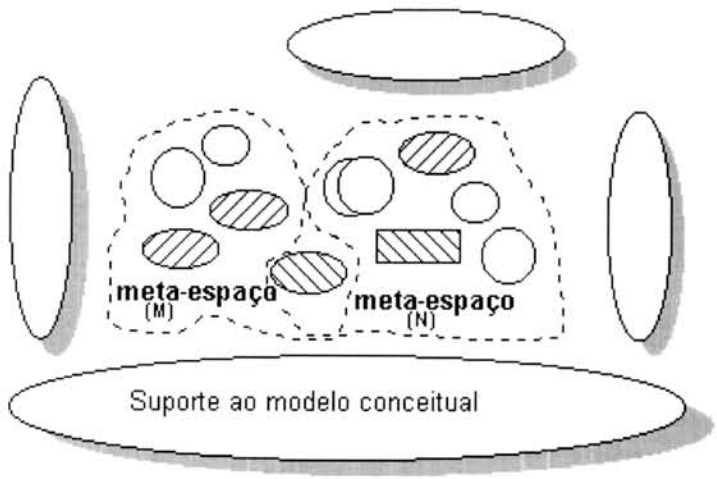


Figura 6.3. Visão simplificada da arquitetura de Aurora

### 6.3.1 Aspectos dos serviços do sistema operacional

O mecanismo básico para construção do sistema operacional Aurora é a *migração de objetos*. Migração de objetos é definida de tal

maneira que um objeto troca de *meta-espaços* quando ele necessitar de algum serviço suportado em outro *meta-espaço*. Na figura 6.3, dizer que o objeto (a) migrou do *meta-espaço* (m) para o *meta-espaço* (n), significa que o objeto (a) trocou seu *meta-espaço* de (m) para (n).

O efeito da migração sobre um objeto é que o mesmo passa a assumir as características semânticas e de comportamento do *meta-espaço* destino. Este conceito representa na verdade, segundo a orientação a objetos, o conceito de herança de escopo [NIE89], isto é, um objeto possui o comportamento que lhe é imposto pelo escopo ao qual pertence.

A adoção deste modelo permite construir o sistema operacional a partir de uma estrutura muito simples, por exemplo, um objeto pode migrar para um *meta-espaço* que representa memória secundária quando é para ser armazenado em disco, da mesma forma um objeto pode migrar para um *meta-espaço* com facilidades de depuração quando é para ser depurado.

Deve-se notar que, diferentemente de uma implementação, baseada por exemplo em *system call*, onde a interação com o sistema operacional é realizada através de um conjunto pré-definido de chamadas, a utilização do conceito de migração permite que todo o sistema operacional, ou mesmo a aplicação do usuário, seja desenvolvido baseado simplesmente na presença do objeto.

### 6.3.2 Aspectos do modelo de código

Uma característica de Aurora é a total integração entre o ambiente de programação e o sistema operacional, trazendo consigo um novo conceito de compilação e execução [ZAN93c]. Em Aurora a função básica do compilador é gerar *meta-objetos*. Estes *meta-objetos* são inseridos numa biblioteca a partir da qual os mesmos podem ser referenciados por objetos do usuário, ou do sistema, e são utilizados pelo sistema operacional quando da instanciação de objetos, ou quando da ativação de métodos externos ao objeto instanciado.

Este contexto está respaldado no parecer de Goldberg [GOL84] de que no modelo de objetos, executar um sistema é algo tão simples como criar objetos e disparar uma mensagem.

A implicação direta da adoção desta filosofia sobre o conceito de compilação é que a função básica do compilador, torna-se compilar classes separadamente (*meta-objetos* em Aurora). Estes *meta-objetos* serão conhecidos pelo sistema operacional através de bibliotecas. Uma vez conhecidos pelo sistema, os *meta-objetos* podem ser referenciados em tempo de execução para instanciação de objetos, ou para ativação de seus métodos.

Por outro lado, a implicação direta do modelo sobre o conceito de execução é que o sistema operacional não recebe mais programas prontos e montados através de processos de *link* edição, mas sim, instanciações e ativações de objetos, oriundas de outros objetos ou diretamente do usuário que interagem através da interface.

### 6.3.3 Aspectos do ambiente de programação e interface

O paradigma de objetos, semelhantemente a outros modelos, engloba dois aspectos distintos: o modelo computacional e o modelo de projeto como filosofia de desenvolvimento. O modelo de projeto é centrado na identificação e organização de conceitos no domínio da aplicação. Superficialmente podemos dizer que modelar um problema, segundo o enfoque de objetos, implica na decomposição deste problema em termos de objetos e na construção de uma hierarquia de classes de objetos, que por sua vez compõem propriedades comuns às subclasses de objetos. Por outro lado, o modelo computacional é representado por um conjunto de objetos cooperando, através de mensagens, na realização de uma determinada tarefa.

Dentro deste contexto e respaldado pelo parecer de Goldberg acima descrito, Aurora propõe que o ambiente de desenvolvimento e a interface sejam baseadas exclusivamente no modelo de objetos, de modo que as únicas entidades visíveis ao usuário sejam os objetos (e classes de objetos).

Deve-se notar também que a adoção desta filosofia vai de encontro ao modelo de projeto onde o resultado do processo de modelagem de um sistema orientado a objetos é um conjunto de classes (*meta-objetos* quando compiladas), que representam os conceitos (entidades reais e/ou abstratas) existentes no domínio do problema.

Entretanto, a disponibilidade deste universo de *meta-objetos* não é suficiente para representar o comportamento de um sistema, sendo necessário estabelecer-se um modelo computacional, isto é, definir uma lógica para o modelo executável. No modelo de objetos este processo é realizado tipicamente através de instanciações e/ou ativações de objetos.

A idéia de Aurora é permitir que este processo seja realizado diretamente pela interface do sistema através de uma "linguagem de comandos" projetada especialmente para manipulação de objetos, ou em tempo de execução através de outros objetos. É importante notar que a criação de objetos, no modelo tradicional, somente ocorre através de programas executáveis e que o processo de executar um sistema em Aurora tornou-se algo tão simples como criar objetos e disparar mensagens, tal qual preconizado por Goldberg.

Esta filosofia conduz Aurora a um enfoque onde os processos de compilação e execução estão mais próximos do paradigma de objetos, ou seja, onde as classes são compiladas e existem isoladamente para serem executadas. Deve-se notar que tradicionalmente os compiladores destroem a estrutura hierárquica das classe em tempo de compilação, enquanto, para geração de código executável, é exigida a presença de um corpo principal do sistema.

Para exemplificar a forma como um sistema é implementado em Aurora, vamos supor a necessidade de se construir um sistema para representar figuras geométricas entre o seguinte conjunto de figuras: { círculo, triângulo, retângulo, trapézio, ...}. O primeiro passo seria determinar os componentes do sistema, isto é, que classe de objetos

podem ser identificadas. Este processo certamente resultará em classes do tipo:

**class** Círculo {/* ... */}

**class** Trapézio {/* ... */}

**class** Triângulo {/* ... */}

**class** Retângulo {/* ... */}

Identificadas as classes do problema, o passo seguinte é construir e implementar as classes através de uma linguagem de programação. Por exemplo, a classe círculo caso escrita em C++ poderia ser implementada da forma abaixo. As demais classes seriam implementadas de forma semelhante, cada qual, com o comportamento que lhe é pertinente:

```
01:   class Círculo {
02:       int X, Y;
03:       int Raio;
04:     public:
05:       Círculo ( int InitX, int InitY, int InitRaio ) {/* ... */};
06:       void Mostra (void) { /* ... */};
07:       void Oculta (void) { /* ... */};
08:       void Expande (int ExpandePara) {/* ... */};
09:       void Contrai (int ContraiPara) {/* ... */};
10:   };
```

A etapa seguinte a ser realizada é o processo de compilação. Em Aurora cada uma das classes é compilada separadamente, exatamente como no exemplo acima, sendo o resultado produzido pelo processo de compilação, o *meta-objeto,* inserido em uma biblioteca passando a partir deste instante a fazer parte do universo de *meta-espaços* disponíveis no sistema.

A partir da existência de um *meta-objeto,* que representa a figura geométrica, objetos podem ser instanciados associados ao *meta-objeto* e seus métodos ativados. Note que este processo de execução pode ser realizado diretamente através da interface do

sistema. Por exemplo, para construir um círculo bastaria ser executado:

```
CC objeto ( círculo(10,12,7) ) // cria o objeto círculo
CC.Expande ( 15 )              // aumenta o raio do círculo
```

"Objeto" representa o mecanismo de Aurora para instanciação de objetos, a partir de *meta-objetos* conhecidos pelo sistema. Supondo um objeto de controle através do qual por algum processo de escolha, o algoritmo decide qual das figuras geométricas deve ser construída, isto é, deseja-se utilizar as classes no interior de outro objeto. Para ter acesso aos *meta-objetos* basta o usuário incluir no objeto de controle a diretiva #include (para C++). A instanciação e a ativação do objeto mantém a mesma sistemática, ou seja:

```
#include <ObjetoAurora.h>

CC objeto ( círculo (10,12,7) )
CC.Expande ( 15 )
```

Deve-se observar que a implementação do problema através do processo tradicional implica na necessidade do sistema conter o código que representa todas as classes previstas, mesmo que, somente uma das figuras geométricas seja utilizada ao longo da execução do sistema.

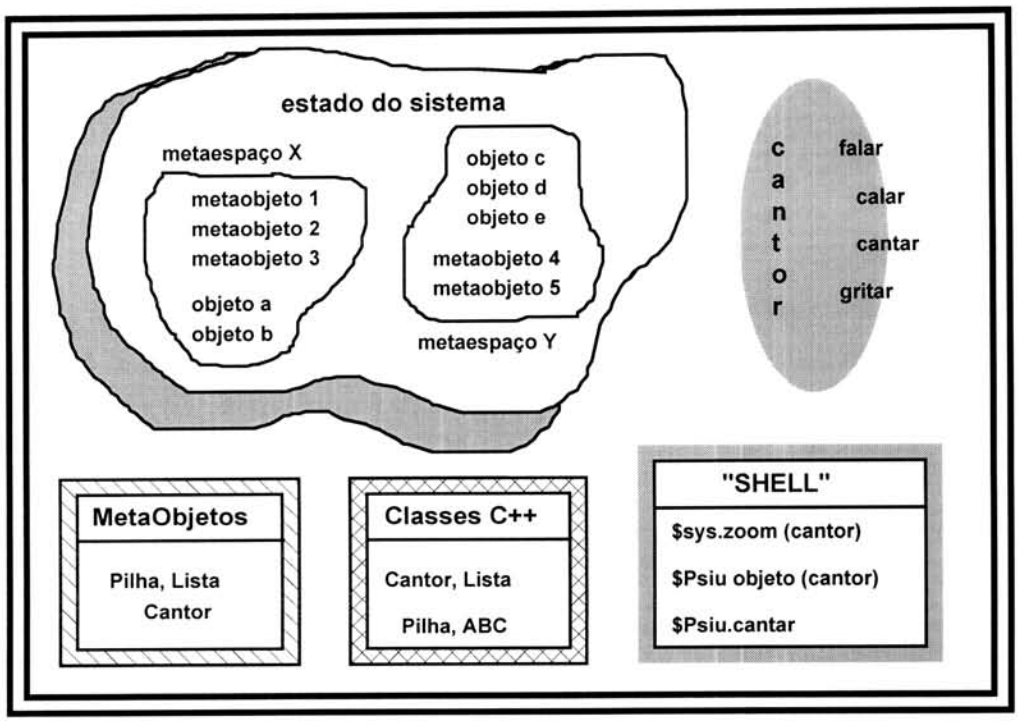


Figura 6.4. Possível interface com o usuário

A figura 6.4 apresenta uma visão simplificada de uma possível interface do usuário, onde o usuário interage diretamente com o sistema através da manipulação de objetos. Neste aspecto Aurora difere da maioria dos sistemas, primeiro porque estes estão apoiados no modelo de programas compilados e link-editados, segundo por que para a maioria dos sistemas uma classe é um padrão estático e imutável para os objetos. Em Aurora a construção hierárquica das classes pode ser transferida para o tempo de execução, permitindo que uma classe herde efetivamente novas propriedades de forma dinâmica. Neste particular até onde vai nosso conhecimento, Aurora é o primeiro sistema multilinguagem a permitir a transferência desta tarefa para o tempo de execução.

Outra característica é que a interface alcançada por Aurora reflete um ambiente exatamente como preconizado pelo paradigma de orientação a objetos, de que o processamento deve ser representado por um conjunto de mensagens, fluindo entre objetos executando de forma paralela, enquanto a execução de sistemas restringe-se a criação de objetos e ao envio de mensagens,

representando verdadeiramente uma aproximação entre o modelo de projeto e computacional.

## 6.4 Avaliação de resultados

O enfoque adotado nesta seção está baseado em dois aspectos distintos: na tradicional avaliação de desempenho realizada sobre a implementação de Aurora e que visa avaliar o resultado obtido em termos de eficiência de implementação e segundo em uma análise das propriedades e características apresentadas pela proposta em relação a outros modelos. Esta análise de propriedades, ainda que não quantitativas é significativa, visto ser conceitual o universo da proposta aqui apresentada.

### 6.4.1 Avaliação de desempenho

O fato do *Meta-Core* prover somente um conjunto mínimo de funcionalidades para suporte ao modelo *objeto/ meta-objeto*, reduziu seu conjunto de funcionalidades a somente duas primitivas, M e R, conforme detalhadas no capítulo anterior. Todavia, podemos comparar o papel destas primitivas com as funções básicas de outros modelos, como por exemplo baseados em *kernel*, sem descaracterizar suas funcionalidades e propriedades.

As medidas de desempenho aqui apresentadas, foram obtidas usando-se um microcomputador com processador Pentium de 100 Mhz, processador de ponto flutuante e 32 Mb de memória física. A tabela 6.1, apresenta primeiramente os resultados obtidos através de medições realizadas sobre a implementação em C++ de Aurora:

| | |
|---|---|
| Primitiva **M** | 14,5 micros segundos |
| Primitiva **R** | 13,7 micros segundos |

Tabela 6.2: Medidas de Tempo de Aurora

De modo a quantificar os resultados acima, tomou-se como referência os resultados apresentados em funções equivalentes na mais recente proposta de implementação de sistema operacional multiprocessado, o sistema Cosy[BUT94] que visa suportar arquiteturas massivamente paralelas.

| | |
|---|---|
| Kernel call (M ou R) | 28 micros segundos |
| Comunicação (M+R) | 118 micros segundos |

Tabela 6.3: Medidas de Tempo de Cosy

As medições realizadas pelo sistema operacional Cosy, e apresentadas na tabela 6.2, ocorreram sobre um processador Transputer 30Mhz-T805 e igualmente através da linguagem C++.

Para compararmos o desempenho entre os dois ambientes, tomamos como referência a dissertação de mestrado sobre avaliação de desempenho em máquinas paralelas [MEN93]. Neste trabalho foi utilizado o algoritmo de Andy Rabagliati para teste da memória RAM do processador Transpoter 30Mhz-T805. Implementamos este mesmo algoritmo sobre o processador Pentium 100, para testar a mesma quantidade de memória (4Kbytes) e através da mesma linguagem de programação (C++). Os resultados obtidos foram os seguintes:

| | |
|---|---|
| Transputer T805-30Mhz | 46 micros segundos |
| Pentium 100Mhz | 9 micros segundos |

Tabela 6.4: Teste de RAM, Algoritmo de Andy Rabagliati

Se considerarmos o Pentium da ordem de 5 vezes a velocidade do Transputer, observamos uma perda na ordem 20% no resultado relativo a interação entre objetos, cuja relevância para os ambientes orientados a objetos foi ressaltada nos capítulos anteriores.

Outra medida importante, ainda que não inserida no escopo do *Meta-Core*, todavia parte integrante do *kernel* em sistemas nele baseados e afeitos aos aspectos de gerenciamento dos objetos, é o custo de criação dos objetos. A tabela 6.3 compara o tempo de criação de objeto apresentado por Aurora com o tempo de criação de processos apresentado por Cosy.

| | | |
|---|---|---|
| Criação de *meta-espaço* | 178 micros segundos | Aurora |
| Criação de objeto | 154 micros segundos | Aurora |
| Criação de processos | 400 micros segundos | Cosy |

Tabela 6.5. Medidas de Tempo

É importante salientar que os resultados apresentados na tabela 6.5, não eram esperados, visto que, conceitualmente, objetos são entidades de granularidade mais leve que processos e, portanto, deveriam apresentar naturalmente menor custo de gerenciamento.

### 6.4.2 Análise de propriedades e características

Conforme ressaltado, esta análise, ainda que não quantitativa nos permite avaliar aspectos importantes (visíveis a nós) e inerentes do próprio modelo conceitual, cuja percepção às vezes foge a uma análise superficial do observador. Todavia, algumas propriedades e características podem ser fatores determinantes e que inviabilizam o suporte eficiente ao modelo de objetos. Abaixo analisamos algumas destas características nos principais modelos para finalmente ressaltar as principais propriedades do nosso modelo.

*Sistemas em Camadas:*

Tornam-se inerentemente difíceis para construção de sistemas complexos, porque nem sempre é possível garantir a existência de um processo restrito a uma camada. Por exemplo, muitos sistemas operacionais contam com a habilidade das funções de baixo nível (tais

como sinais em Unix) para invocar processos a nível do usuário, sendo esta um tarefa impossível de ser realizada usando-se estritamente técnicas de estruturação em camadas.

*Kernel Coletivos:*

Como benefícios deste esquema pode-se ressaltar: modularização, reusabilidade, manutabilidade, extensibilidade e assim por diante. Entretanto, tais benefícios são exclusivamente internos ao sistema operacional, o que torna-se uma limitação, visto que seria desejável uma visão uniforme dos objetos a serem aplicados tanto fora como internamente ao sistema.

*Máquinas Virtuais:*

Uma das vantagens das estruturas de máquinas virtuais é a possibilidade de implementar vários tipos de sistema operacionais, entretanto os sistemas são disjuntos e mesmo com um suporte para comunicação através dos diferentes sistemas operacionais, cada um deles é uma entidade distinta, o que dificulta uma forma rica de interação, compartilhamento e comunicação.

*Independência entre Políticas e Mecanismos:*

Apesar desta separação aumentar a flexibilidade do sistema para o usuário, uma das dificuldades é a determinação de que mecanismos deverão ser fornecidos de modo a permitir a implementação de qualquer política do usuário. Entretanto esta filosofia tem se mostrado um importante conceito, aparecendo repetidamente em vários contextos, inclusive no aqui proposto.

*Modelo Cliente-Servidor:*

Uma das características desta estrutura é que ela pode ocultar diferenças entre linguagens de programação pelo suporte de objetos implementados em linguagens diferentes, de uma maneira uniforme através por exemplo do procurador.

*Estruturas baseadas em objetos:*

Uma característica da estrutura baseada em objetos, similarmente a estrutura de *kernel* coletivo, é que ela requer uma disciplina para organizar serviços do sistema operacional como uma coleção de objetos. Outra possível característica é a existência de uma *thread* de controle de modo que, objetos, coletivamente, possam ser tratados como uma unidade única.

*Estruturas Reflexivas:*

Apertos apresenta um modelo reflexivo como disciplina de implementação e não como modelo conceitual do ambiente computacional.

*Modelo de Estrutura Reflexiva Proposto:*

Entre as propriedades e características do modelo aqui proposto, e não encontradas entre os modelos apresentados, salientam-se:

- O fato do modelo prover naturalmente recursos ao modelo *meta-objeto*, permitindo a construção de sistemas reflexivos em todos os níveis do ambiente.
- Prover uma clara separação entre objetos e os mecanismos de controle dos objetos, permitindo a construção de modelos flexíveis de sistemas operacionais multiprocessados, onde o ambiente pode ser modelado de acordo com as características da aplicação paralela.
- Mecanismos de baixo nível para gerenciamento da interação entre objetos, minimizando o tempo de latência das comunicações.

- Uniformidade no tratamento de recursos, permitindo a inexistência de distinção entre as abstrações providas pelo sistema operacional e aquelas vistas pelo usuário.

**Capítulo 7**

# Conclusões e Perspectivas

*Sintetiza as características, benefícios e delimitações do resultado alcançado e aponta perspectivas futuras.*

Este trabalho apresentou uma nova estrutura conceitual para implementação de sistemas operacionais multiprocessados, onde são exploradas as idéias de computação reflexiva não somente como disciplina de implementação, mas principalmente como modelo de estrutura conceitual.

O modelo proposto torna a exploração de arquiteturas multiprocessadoras mais flexível, visto que conceitos como suporte a migração de objetos fazem parte do modelo conceitual. Esta mobilidade natural atribuída aos objetos, permite que o sistema gerencie a execução paralela entre os mesmos com um baixo custo, visto que os objetos são entidades de granulosidade mais leve que processos.

Outro aspecto saliente da vantagem da utilização de uma arquitetura multiprocessada na implementação de sistemas orientados a objetos é a possibilidade do sistema tratar em paralelo com a aplicação, problemas críticos e característicos da execução orientada a objetos, tais como: criação dinâmica de objetos, acoplamento dinâmico, gerenciamento de mensagens, pesquisas a objetos e coleta de lixo.

Uma característica singular do modelo é permitir a transferência da construção da hierarquia das classes para tempo de execução, possibilitando que uma classe herde efetivamente novas propriedades de forma dinâmica. Neste particular, até onde onde vai nosso conhecimento, Aurora é o primeiro sistema multilinguagem a permitir a transferência desta tarefa para tempo de execução.

Outra propriedade que distingue o modelo é ser puro, isto é, suportar exclusivamente linguagens orientadas a objetos. Além de permitir um tratamento uniforme tanto para o sistemas como para as aplicações do usuário, a adoção deste modelo facilita a integração entre o sistema operacional e as linguagens de programação, apesar de originalmente desenvolvidos de forma independente. Tal integração representa uma base fundamental na busca de ambientes de programação e interfaces qualificados.

A utilização do modelo proposto facilita a implementação de modelos uniformes, ou seja, sistema e aplicações são construídos baseados na mesma abstração. Tal facilidade não é encontrada na maioria dos outros modelos estruturais, por exemplo a definição da abstração básica dos sistema baseados em *kernel* é definida por ele. Exemplos desta restrição podem ser encontradas em sistemas como Amoeba [MUL90], Chorus [HER88] e Choices [CAM87], onde o *kernel* define níveis de abstração distintos, isto é, o *kernel* define a interface para o objeto da aplicação.

Outro aspecto característico é permitir a construções de interfaces capazes de refletir um ambiente exatamente como preconizado pelo paradigma de orientação a objetos, de que o processamento deve ser representado por um conjunto de mensagens fluíndo entre objetos, executando de forma paralela, enquanto a execução de sistemas restringe-se a criação de objetos e ao envio de mensagens.

Os resultados obtidos tornam-se mais relevantes se enfocados sob o ponto de vista de que até recentemente, *hardware* e algoritmos (numéricos) dominavam o processamento paralelo, a tecnologia dos compiladores enfatizava o particionamento dos dados e a paralelização automática dos algoritmos, enquanto o papel do sistema operacional era relegado a um segundo plano, através do uso de *bypass*. Todavia, o surgimento das arquiteturas (massivamente) paralelas com memória distribuída introduziu transformações a partir das quais a demanda por softwares de suporte vai além dos compiladores e dos ambientes de execução (*runtimes*).

A presença de dezenas, ou centenas, de processadores introduziu desafios de desempenho, tais como latência das comunicações, aos quais somente proposições de soluções completas serão capazes de superá-los. Em conseqüência, o principal desafio a nível de sistema operacional tornou-se, primeiramente, um problema conceitual e centrado no modelo de estrutura sobre a qual o sistema operacional é construído, obter uma estrutura que minimize o tempo de carga das aplicações, evite competição excessiva aos serviços do sistema operacional, tolerante a falhas, dinamicamente configurável e orientado para a aplicação, representa um desafio ao atual estado da arte da pesquisa em sistemas operacionais (massivamente) paralelos.

O modelo reflexivo proposto possui a habilidade de configurar-se segundo as necessidades da aplicação, ou seja, prover somente os serviços que a aplicação necessitar de modo a evitar *overhead* desnecessários, como por exemplo, suporte de multitarefa para aplicações que necessitam somente suporte a uma única tarefa. Neste aspecto, o uso do modelo proposto permite que a estrutura básica do sistema operacional vá em direção a um modelo de *kernel* flexível, ou seja, possa ser modelado segundo as características e necessidades da aplicação nos diferentes nódos do sistema.

A aparente complexidade adicional introduzida pela flexibilidade, foi claramente contornada no modelo proposto através do uso de orientação a objetos, do suporte ao modelo meta-objetos[FER89] e da exploração das idéias da computação reflexiva [MAE87], **não somente como disciplina de implementação, mas como modelo conceitual da estrutura do sistema operacional multiprocessado.**

# Bibliografia


[AGH 86] AGHA, Gul. ACTORS. A Model of Concurrent Computation in Distributed Systems. **The MIT Press**, 1986. (The MIT Press series in artificial intelligence).

[AGH 90] AGHA, Gul. Concurrent Object-Oriented Programming. **Communications of the ACM**, New York, v.33, n.9, p.125-141, 1990.

[ALM 85] ALMES, G. T. et.al. The Eden System: A Technical Review. **IEEE Transactions on Software Enginnering**, v.11, n.1, p.43-58, 1985.

[AME 87] AMERICA, P. **POOL-T**: A Parallel Object-Oriented Language. MIT Press, 1987.

[ANC 95] ANCOMA, M.; DODERO, G.; GIANUZZI, V. et al. Reflective Architecture for Reusable Fault-Tolerant Software. In: Latin American Conference in Informatics. **Proceedings**... Porto Alegre:SBC, 1995. p.87-98.

[BAN 85] BANINO,J.S. Distributed Couple Actors: A CHORUS Proposal for Reliability, In: International Conference on Distributed Computing Systems,3.,1985. **Proceedings**...[S.l], IEEE,1985.

[BLA 86] BLACK, A. et al., Mach and Matchmaker - Kernel and Language Support for Object-Oriented Distributed Systems. OOPSLA'86. **Proceedings**...[S.l], v.21, 1986.

[BOD 93] BODHISATTWA, M. et al. A survey of Multiprocessor Operating System Kernels. [S.l]: Georgia Institute of Technology, 1993. (Technical Report).

[BOO 94] BOOCH, G. **Object-Oriented Analysis and Design,** Califormia: Benjamin Cummings, 1994.

[BUT 94] BUTENUTH,R. COSY-KERNEL as na Example for Efficient Call Mechanism on Transputers. In: Transputer/Occam International Conference,6.,1994, Tokyo. **Proceedings**...Tokyo, IEEE,1994.

[CAM 87] CAMPBELL,R. et al. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). **Operating Systems Review**, v.21, n.3, 1987.

[CHA 92] CHAMBERS, C. **The Design and Implementation of the SELF Compiler. An Object-Oriented Compiler for Object-Oriented Programming Languages**. Ph.D. Thesis. Stanford: Stanford University, 1992.

[CHE 84] CHERITON,F.R. et al. "The V-Kernel: A software Base for Distributed Systems", **IEEE Software**, v.1, n.2, p.19-43, 1984.

[CHE 88] CHERITON,D. The V Distributed System. **Comunications of the ACM**. New York, v.31, n.3, p.314-333, 1988.

[CHI 91] CHIN, R.S. CHANSON, S.T. Distributed. Object-Based Programming Systems. **ACM Computing Surveys**. New York, v.23, n.1, 1991.

[CRE 81] CREASY,R.J. The Origin of the VM/370 Time-Sharing Systems, **IBM Journal of Research and Development**, [S.l], v.25, n.5, p.483-490, 1981.

[COH 75] COHEN,E. and JEFFERSON,D.: Protection in the Hydra Operating System. In: Symposium on Operating System Principles,5.,1975. **Proceedings**...[S.l]:ACM, 1975.

[DAL 80] DALAL, Y.K. et al. Pilot: An Operating System for a Personal Computer, **Communication of the ACM**, New York, v.23, n.2, 1980.

[DAS 89] DASGUPTA, P. et al. The Clouds distributed operating system. In: International Conference on Distributed Computing Systems,8.,1989. **Procedings**... [S.l.]: IEEE, 1989.

[DIJ 68] DIJKSTRA,E.W.: The Structure of THE Multiprogramming System, **Communication of the ACM**, New York, v.11, p. 341-346, 1968.

[FER 88] FERNANDES,E.S.T.; AMORIN,C.B.V. Uma Introdução a Computação Paralela e Distribuída, In: Escola de Computação,4.,1988, Campinas, SP. **Anais**..., Campinas,1988.

[FER 89] FERBER, J. Computational Reflection in Class Based Object-Oriented Languages. Sigplan **Notices**, New York, v.24, n.10, p.317-326, 1989.

[FUJ 91] FUJINAMI, N. et al. The Muse Object Architecture: a New Operating System Structuring Concept. **Operating System ACM Press**, New York, v.25, n.2, p.22-46, 1991.

[FOO 93] FOOTE, B. Architectural Balkanization in the Post-Linguistic. Workshop on OO Reflection And Meta-Level Architectures, OOPSLA, 1993. **Proceedings**..., [S.l], p.1-9, 1993.

[GRA 89] GRAUBE, N. Metaclass Compatibility. **OOPSLA,** Luisiana, v.24, n.10, p.305-315, 1989.

[GOL 84] GOLDBERG, A. Smalltalk-80. The Interactive Programming Environment. **Addison Wesley**, 1984.

[HAB 90] HABERT, S. et al. COOL: Kernel Support for Object-Oriented Environments. ECOOP/OOPSLA Conference, 1990, **Proceedings**... [S.l], 1980.

[HAM 92] HAMMER, D. C.; VAN ROOSMALEN, O.S. Distributed Systems. In: International Workshop on Object Orientation in Operating Systems, 2., 1992, Dourdan, France, **Proceedings**... Dourdan, p.301-310, 1992.

[HWA 84] HWANG, K.; BRIGGS, F. A. Computer Architecture and Parallel Processing. New York: **McGraw Hill Book Company,** 1987.

[HER 88] HERRMANN, F. et al. Chorus distributed operating system. **Computing Systems**, v.1, n.4, p.305-367, 1988.

[INM 89] INMOS. The Transputer Databook. New York: **INMOS**, 1989.

[INM 90] INMOS. Module Motherboard Architecture: IMS B008. Reference Manual. **INMOS**, 1990.

[JAL 94] JALOTE,P. **Fault-tolerance in Distributed Systems**, Prentice-Hall, 1994.

[ISH 91] ISHIKAW, A,Y. Reflection Facilities and Realistic Programming, **ACM Sigplan Notices**, New York, v.26, n.8, 1991.

[KIC 92] KICZALES, G. et al. A New Model of Abstraction for Operating System Design. In: Intenational Workshop on Object Orientation in Operating Systems,3., 1993, **Proceedings**... [S.l], p.346-350, 1992.

[KIC 93] KICZALES,G.; LAMPING,J. Operating System: Why Object-Oriented?. In: International Workshop on Object Orientation in Operating Systems, 3., 1993, **Proceedings**... [S.l], p.25-30, 1993.

[KOT 84] KOTOV, V. E. **Algorithms, Software and Hardware of Parallel Computers**. Berlin: Spring-Verlag, 1984.

[LEA 93] LEA R. et al. COOL: System Support for Distributed Programming. **Communication of the ACM**, New York, v.36, n.9, 1993.

[LEB 89] LEBLANC, T. J. et al. The Elmwood Multiprocessor Operating System. **Software Practice and Expience**. London, v.19, n.11, p.1029-1055, 1989.

[LEV 75] LEVIN, R. et al. Policy/Mechanism Separation in Hydra, In: Symposium on Operating System Principles, 5., 1975, **Proceedings**... [S.l]: ACM Press, p.132-140, 1975.

[LIS 87] LISKOV, V.B. et al. Implementation of Argus. ACM Proccedings 12th Symposium of Operating System Principles, 12., 1987, **Proceedings**... [S.l]: ACM Press, p.111-122, 1987.

[MAD 95] MADSEN, O. L. Open Inssues in Object-Oriented Programming. **Software Pratice and Experience**. New York. v.25, n.S4, p.3-43, 1995.

[MAE 87] MAES,P. Concepts and Experiens in Computational Reflection. **Sigplan Notices**, New York, v.22, n.12, p.147-169, 1987.

[MAE 88] MAES, P. Issues in Computacional Reflection. In: MAES,P.; NARDI, D. (Ed). Meta-level Architectures and Reflection. Amsterdam: **Elsevier Science**. p.21-35, 1988.

[MAL 89] MALIK, J. et al. Cosmos: na architecture for a distributed programming environment. **Computer Communications**, Evildford, v.12, n.3, p.147-157, 1989.

[MAL 92] MALENFANT, J.; DONY, C., COINTE, P. Behavioral Reflection in a Prototype-Based Language. In: International Workshop on New Models for Software Architeture/Reflection and Meta-Level Architectures, 1992. **Proceedings**..., Japan: Elsevier Science, p.143-153, 1992.

[MUL 90] MULLENDER, S. J. et al. Amoeba - A Distributed Operating System for the 1990s. **IEEE Computers**. Los Alamitos, v.23, p.44-53, 1990.

[NAK92] NAKAJIMA, S. What Makes a Language Reflective and How? In: Iinternational Workshop on New Models for Software Architecture/ Reflection and Meta-Level Architetures, 1992. **Proceedings**..., Japan: Elsevier Science, p.125-136, 1992.

[NEL 91] NELSON, M.L. Concurrency & Object-Oriented Programming. **ACM Sigplan Notices**. New York, v.26, n.10, p.63-72, 1991.

[NIC 87] NICOL, J.R. Operating System Design: Towards a Holistic Approach, **Operating System Review**. New York, v.21, n.1, 1987.

[NIC 89] NICOL, J.R, et al. Cosmos: An Architecture For a Distributed Programming Environment. **Computer Communications**, Evildford, v.12, n.3, p.147-157, 1989.

[NIE 87] NIERTRASZ, OSCAR. Active Objects in Hibrid. OOPSLA Conference 1987, Orlando USA, **Proceedings**... p.243-253, 1987.

[NIE 89] NIERSTRASZ, OSCAR. **A Survey of Object-Oriented Concepts. Object-Oriented Concepts. Databases. and Applications**. Ed. Won Kim, Frederick H.L.p.79-124, 1989.

[ORG 72] ORGANICK,E.I. **The Multics System, Cambrige**: MIT Press, 1972

[OUT 80] OUTERHOUT,J.K. et al. Medusa: An Experiment in Distributed Operating System Structure, **Communications of the ACM**, New York, v.23, n.2, 1980.

[PAR 92] PARDYAK, P. Group Commnication in na Object-Based Environment. In:International Workshop on Object Orientation in Operating Systems, 2., **Proceedings**...Dourdan: p.106-116. 1992.

[RAS 81] RASHID,R.. Accent: A Communication Oriented Network Operating Systems. In: Symposium on Operating Systems Principles. 8., 1981, Pacific Grove, US. **Proceedings**..., Pacific Grove, ACM, p.64-75, 1981.

[POW 83] POWELL,M.L and PRESOTTO,D.L. A Reliable Broadcast Communication Mechanism, **ACM Operating System Review**, New York, v.17, n.5, 1983.

[SHA 86] SHAPIRO,M,: Structure and Encapsulation in Distributed Systems: The Proxy Principle, In: International Conference on Distributed Computing Systems, 6., 1986, **Proceedings**...[S.l]:[S,n],1986.

[SCH 89] SCHWAN, K. , and GOPINATH,P. . CHAOS: Why One Cannot Have Only An Operating System for Real-Time Applications. **Operating Systems Review**. New York, v.23, n.3, 1989.

[SCH 94] SCHRÖDER-PREIKSCHAT, WOLFGANG. **The Logical Design of Parallel Operating Systems**. Englewwood Cliffs, New Jersey, Prentice Hall, 1994.

[SHA 89] SHAPIRO, M. et al. SOS: An Object-Oriented Operating System - Assessment and Perspectives. **Computing Systems**. v.2(4), 1989.

[STE 94] STEEL, L. Beyond objects. European Conference on Object-Oriented Programming, Italy, 1994. **Springer-Verlag,** Berlin, p.1-11, 1994.

[TOM 89] TOMLINSON, CHRIS; SCHEEVEL, MARK**.** Concurrent Object-Oriented Programming Languages. Object-Oriented Concepts. Databases. and Applications. ed. Won Kim and Frederick H. Lochovsky. p.79-124, 1989.

[TRI 89] TRIPATHI, A.; BERGE, E. An Implementation of the Object-oriented Concurrent Programming Language SINA. **Software-Practice and Experience**, v.19, n.3, p.235-256, 1989.

[WIN 95] WINDOWS 95. Feature Review. Microsoft Corporation, 1995.

[WUL 74] WULF, W.A .et al.: HYDRA: The Kernel of a Multiprocessor Operating System, **Communications of the ACM**, v.17, p.337-345, 1974.

[YON 87] YANEZAWA, AKINORI; et al.. **Modelling and programming in na Object-Oriented Concurrent language ABCL/1**. Cambridge: MIT Press, 1987.

[YON 88] YANEZAWA, AKINORI; TOKORO, MARIO. **Object-Oriented Concurrent Programming: An Introduction**. Cambridge: MIT Press, 1988.

[YOK 92] YOKOTE, YASUHITO. **The Apertos Reflective Operating System: The Concept and Its Implementation**. [S.l]:Sony Computer Science laboratory Inc., 1992. (Technical Report).

[ZAN 93a] ZANCANELLA,L.C.; NAVAUX, P.O.A. AURORA: Um Sistema Operacional Orientado a Objetos para Arquiteturas Multiprocessadoras. In: SBAC-PAD,20, Congresso da SBC,13, Florianópolis,SC. **Anais**... Florianópolis:SBC, p.502-514, 1993.

[ZAN 93b] ZANCANELLA,L.C.; NAVAUX, P.O.A. Herança Dinâmica em Aurora. In: SBAC-PAD,20, Congresso da SBC,13, Florianópolis,SC, **Anais**... Florianópolis:SBC, p.286-296, 1993.

[ZAN 93c] ZANCANELLA,L.C.; NAVAUX,P.O.A. Os processos de compilação e execução em AURORA. In: SBES, 7, 1993, Rio de Janeiro, RJ. **Anais**... Rio de Janeiro:SBC, 1993, p.196-207, 1993.

[ZAN 94] ZANCANELLA,L.C.; NAVAUX,P.O.A. Object-Oriented Operating Systems: The Aurora Approach, In: Conferência Chilena, 1994, Concepción, Chile, **Anales**...Concepción:[S,l], p.271-280, 1994.

[ZAN 95a] ZANCANELLA,L.C.; NAVAUX,P.O.A. A Reflective Multiprocessor Object-Oriented Operating Systems, In: Nordic Transputer Conference on Parallel Computing and Transputers, 4., 1995, Linkopings, Sweden, **Proceedings**... Linkopings, p.421-427, 1995.

[ZAN 95b] ZANCANELLA,L.C.; NAVAUX,P.O.A. Flexible Kernel: The AURORA approach for Multiprocessor Operating System, In: International Conference on Parallel and Distributed Processing, Techniques and Applications, 1995, Georgia, US., **Proceedings**... Georgia:[S,l], 1995.

# Estrutura Reflexiva para Sistemas Operacionais Multiprocessados

**Anexo 1**

*Contém os resultados da validação do modelo implementadas em C++*

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#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;

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif  // _Types_h_DEFINED
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef _CPU_h_DEFINED
#define _CPU_h_DEFINED

#include  <HardWare\H\Types.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Estrutura dependente do Hardware

enum CPUMode  {mSYSTEM, mUSER};

struct CPUContext {

   CPUMode mode;

   struct CPURegisters {
      // Definicao dos Registradores
   };

   struct CPUInterrupters {
      // Controle de Interrupcoes
   };

};

// Definir macros para acessos aos Registradores
#define GET_CPUREG()
#define SET_CPUREG()

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif  // _CPU_h_DEFINED
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#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>


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// A classe Activity  e' utilizada  para  representar  o aspecto
// dinamico do objetos. Seus atributos incluem: identificacao do
// objeto, link para meta-nivel,  lista de mensagens disponiveis
// e o estado da execucao (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
```

```
   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 );

};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif   // Activity_h_DEFINED
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef GlobalError_h_DEFINED
#define GlobalError_h_DEFINED


enum mError {
     mSUCCESS          =  0, // OK
     mCFAIL            = -1, // Construtores.
     mDFAIL            = -2, // Destrutores.
     mALLOC            = -3  // Alocacao de memoria.
     mCTXTDEL          = -4  // Delecao de Contexto
     mCTXTNEW          = -5  // Criacao de Contexto
     mMFAIL            = -6  // Primitiva R
     mNOTACTIVITY      = -7  // Nova Activity nao pode ser criada
     mNOTCONT          = -8  // continuacao erro
     mNOTAID           = -9  // AID nao pode ser criado
     mRFAIL            = -10 // Primitiva R erro.
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef _Params_h_DEFINED
#define _Params_h_DEFINED

#define NAMELEN     64
#define PATHLEN     128

// Size of Disk Block

#define DISK_BLOCK_SIZE 512

#endif  // _Params_h_DEFINED
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef Structures_h_DEFINED
#define Structures_h_DEFINED

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Estrutura das mensagens para as primitivas de MetaCore

struct Message {
    Entry object;
    Entry method;
    void* message;
};

struct MessageM {
    pActivity source;
    pActivity target;
    Message* message;
};

struct MessageR {
    pActivity source;
    Message* message;
};

struct MetaActivity {
    pActivity meta;   // Activity do metaobjeto no metanivel
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif   // Structures_h_DEFINED
```

```
// AURORA Operating System - (C) Copyright UFSC/UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef AID_h_DEFINED
#define AID_h_DEFINED

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Identificar objetos (activity) univocamente

class AID {

  private:
     magicword  ObjectID;

  public:
     AID();
     magicword GetID();
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Activity::Activity( )
{
    AuroraMetaCore = LocalMetaCore;

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

    // incluir inicializacoes dependentes de hardware
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Activity::Activity( size_t size, CPUMode mod )
{
    AuroraMetaCore = LocalMetaCore;

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

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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 );
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void Activity::ExecuteM ( Message* pMsg )
{
    if ( Context.mode == Meta->Context.mode ) {
       MessageM ThisMessageM = new MessageM;
       ThisMessageM->message = pMsg;
       ThisMessageM->source  = this;
       AuroraMetaCore->M ( Meta, ThisMessageM );
    }
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void Activity::ExecuteR ( MessageR* pMsg )
{
    if ( Context.mode == pMsg->pActivity->Context.mode ) {
       AuroraMetaCore->R ( pMsg );
    }
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include <dos.h>   // para acesso a gettime()

#include  <\aurora\common\h\System_ID.h>


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

AID::AID()
{
   struct time t;
   gettime(&t);
   ObjectID = t.ti_hund; // dependente de implementacao
};


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

magicword AID::GetID()
{
    return ( ObjectID );
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include <\aurora\common\h\EntryTable.h>


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EntryTable::DefineSize ( size_t siz )
{
    entry = new Entry[siz];
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EntryTable::EntryTable ( )
{
    size  = 1;
    DefineSize ( size );
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EntryTable::EntryTable ( size_t siz )
{
    size  = siz;
    DefineSize ( size );
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EntryTable::~EntryTable ( )
{
    delete entry;
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EntryTable::SetSize ( size_t siz )
{
    delete entry
    size  = siz;
    DefineSize ( size );
}
```

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
size_t EntryTable::GetSize ( )
{
    return ( size );
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entry EntryTable::GetEntry ( size_t index )
{
    if (index < size) {
       return ( entry[index] );
    };
    return ( NULL );
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void EntryTable::SetEntry (size_t index, Entry n)
{
    if (index < size) {
       entry [index] = n;
    };
}


// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef MetaCore_h_DEFINED
#define MetaCore_h_DEFINED

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// MetaCore: Metaobjeto terminal que prove as funcionalidades
// básicas para suporte ao modelo objeto/metaobjeto:
//
// M : realiza a meta-computação, isto é, suspende a execução
//     do objeto e transfere o controle para o metaobjeto.
//
// R : retorna da meta-computação, isto é, transfere o controle
//     da execução do meta-nível para o objeto.


class MetaCore {

   private:

      // usado pelas primitiva M e R
      mcError  Bind   (register MessageM* pMsg);
      mcError  Unbind (MessageR* pMsg);

      // Tratamento de excecao
      void  ExceptionHandler (MetaActivity* pActive, longword error);

   public:

      MetaCore ();

      // primitivas publicas de MetaCore
      void  M  ( MetaActivity* mObject, MessageM* pMsg );
      void  R  ( MessageR* pMsg );

};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
extern MetaCore *LocalMetaCore;
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\metacore\h\MetaCore.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
MetaCore::MetaCore ( )
{
    //  ShowMessage (" Welcome to the Aurora world ");
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mcError MetaCore::Bind (register MessageM* pMsg)
{
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mcError MetaCore::Unbind (MessageR* pMsg)
{
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\metacore\h\MetaCore.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void MetaCore::M ( MetaActivity* mObject, MessageM* pMsg )
{
    Bind ( pMsg );

    pMsg->source->SetInterruptEnable( TRUE );
    pMsg->source->GetContextoCPU();
    pMsg->source->state = sDORMINDO;

    mObject->SetEntry ( pMsg->message->object );
    mObject->SetContextoCPU ( mObject->Context );
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\metacore\h\MetaCore.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void MetaCore::R ( MessageR* pMsg )
{
    UnBind ( pMsg );

    pMsg->source->SetInterruptEnable( TRUE );
    pMsg->source->GetContextoCPU();
    pMsg->source->state = sDORMINDO;

    if ( MultiNodePrimitive() ) {
       pMsg->SetEntry ( pMsg->message->method );
       pMsg->SetContextoCPU ( mObject->Context );
       return;
    }

    // Send ( GetRemoteMachine(), pMsg )

}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef Memory_h_DEFINED
#define Memory_h_DEFINED

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

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#define M_PAGESIZE  1024     // bytes por pagina
#define M_PAGEBITS  10       // bits a ser usados por pagina
#define M_NBITS     32

union overhead {
    overhead *oh_next;
    struct {
        longword : 31,
             bit :  1;
    } free;
    byte    oh_index;
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

class Memory {
  private:
    void*       here;
    void*       eheap;
    overhead*   next [M_NBITS];
    void        more (size_t nbits);

  public:
    // initializacao
    Memory ();
    Memory (void* pBase = NULL, size_t size = 0);

    void    Initialize (register void* pBase, size_t size);

    // metodos publicos
    void*   Alloc (size_t size);
    void    Free  (void* pFree);
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mmemory\h\mmemory.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void* Memory::Alloc (size_t size)
{
    register int        nbits = 1;
    register int        nbytes = (int) size;
    register overhead*  p;
    register overhead** op;

    if (! nbytes) return (NULL);
    nbytes += sizeof (overhead);
    while (nbytes >>= 1) nbits ++;
    op = next + nbits;
    if (! *op) {
       (void) more (nbits);
    }
    if (! (p = *op)) {
        return (NULL);
    }
    *op = p -> oh_next;
    p -> free.bit = 1;  // em uso
    p -> oh_index = (byte) nbits;
    return ((void*) (p + 1));
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mmemory\h\mmemory.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void Memory::Free (void* pFree)
{
    register overhead*  op = (overhead*) pFree;
    register overhead** p;

    if (! op --)
        return;
    if (! op -> free.bit)
        return;      // memoria ja liberada
    p = next + op -> oh_index;
    op -> oh_next = *p;
    *p = op;
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mmemory\h\mmemory.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void Memory::Initialize (register void* pBase, longword size)
{
    register overhead** pOH;
    register overhead** pEnd;

    if (pBase && size) {
        here = pBase;
        (void) LongZero ((longword*) pBase, size >> 2);
        eheap = (void*) (((byte*) pBase) + size);
        pOH = next;
        pEnd = pOH + M_NBITS;
        while (pOH < pEnd) {
           *pOH ++ = NULL;
        }
    }
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Memory::Memory (void* pBase, longword size)
{
    (void) Initialize (pBase, size);
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mmemory\h\mmemory.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void Memory::more (register longword nbits)
{
    register overhead*  op;
    register longword   nblocks;
    register longword   size;

    op = (overhead*) here;

    if (nbits < M_PAGEBITS) {
        here = ((byte*) op) + M_PAGESIZE;
        nblocks = 1 << (M_PAGEBITS - nbits);
    } else {
        here = ((byte*) op) + (1 << nbits);
        nblocks = 1;
    }

    if (here >= eheap)
        return;

    *(next + nbits) = op;
    size = 1 << nbits;
    while (-- nblocks > 0) {
        op -> oh_next = (overhead*) (((byte*) op) + size);
        op = op -> oh_next;
    }
    op -> oh_next = NULL;
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef MSpaces_h_DEFINED
#define MSpaces_h_DEFINED

#include <\aurora\hardware\h\Types.h>
#include <\aurora\common\h\activity.h>
#include <\aurora\common\h\Structures.h>
#include <\aurora\common\h\GlobalError.h>
#include "\aurora\metacore\h\MetaCore.h"
#include "\aurora\mmemory\h\mmemory.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

class MSpaces {

   private:
      Memory* MemoryMachine;

   public:

      mError NewMetaSpace   ( MetaAtivity* mSpaces,
                                          size_t size, CPUMode mod );
      mError KillMetaSpace  ( MetaAtivity* mSpaces,
                                          size_t size, CPUMode mod );
      mError NewMetaObject  ( MetaAtivity* mObject,
                                          size_t size, CPUMode mod );
      mError NewObject      ( Ativity* object );

      boolean CheckActivity ( pAtivity activity, Entry n );
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mspaces\h\mspaces.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
boolean MSpaces::CheckActivity ( pAtivity activity, Entry n )
{
   return ( n < activity->ExecQueue->GetSize() );
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mspaces\h\mspaces.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mError MSpaces::KillActivity ( pAtivity activity )
{
    MemoryMachine->Free ( activity->GetAddress() );
    delete activity;
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mspaces\h\mspaces.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mError MSpaces::NewMetaObject ( MetaAtivity* mObject,
                                        size_t size, CPUMode mod  )
{
    MemoryMachine->Alloc ( size );
    MetaAtivity* Tmp;
    Tmp->mObject;
    mObject = new  Activity ( size, mod );
    mObject->SetMeta ( Tmp );
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mspaces\h\mspaces.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mError MSpaces::NewMetaSpace ( MetaAtivity* mObject,
                                        size_t size, CPUMode mod  )
{
    MemoryMachine->Alloc ( size );
    MetaAtivity* Tmp;
    Tmp->mSpaces;
    mSpaces = new  Activity ( size, mod );
    mSpaces->SetMeta ( Tmp );
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "\aurora\mspaces\h\mspaces.h"

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mError MSpaces::NewObject ( Ativity* object,
                                        size_t size, CPUMode mod  )
{
    pAtivity Tmp;
    Tmp->object;
    MemoryMachine->Alloc ( size );
    object = new  Activity ( size, mod );
    object->SetMeta ( Tmp );
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

# Estrutura Reflexiva para Sistemas Operacionais Multiprocessados

**Anexo 2**

*Contém resultados obtidos na implementação de Aurora.*

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef MCoreError_h_DEFINED
#define MCoreError_h_DEFINED

enum cError {
    cSUCCESS    = 0,
    cNOACTIVITY = -1,
    cNOTFOUND   = -2,
    cNOTINSTALL = -3,
    cNOTAID     = -3,
    cNOTRESUME  = -4,
    cNOTRUN     = -5,
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#endif
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef MCore_h_DEFINED
#define MCore_h_DEFINED

#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\common\h\activity.h>

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
class MCore {

   private:
      // Contexto da propria execucao
      pActivity myContext;

      void  localReply (register MCoreDescriptor* pReply);
      void  externalReply (register MCoreDescriptor* pReply);

   public:
      inline MCoreDescriptor*   GetDescriptor (AID id);

      // usados por objetos
      void  Reply ();
      cError    Call (AID receiver, longword selector, void* pMsg);
      cError    Send (AID receiver, longword selector, void* pMsg);
      cError    Exit ();

      // usados por metaobjetos
      mError    Deliver (AID receiver, longword selector, void* pMsg,
                AID continuation);
      mError    GetContext (AID object, pActivity pContext);
};

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
inline MCoreDescriptor*
MCore::GetDescriptor (AID id)
{
    return ((MCoreDescriptor*)
        (((byte*) ((void*) id)) - sizeof (headerDescriptor)));
}

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
endif // MCore_h_DEFINED
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#ifndef _MapEntry_h_DEFINED
#define _MapEntry_h_DEFINED

#include  <\aurora\hardware\h\Types.h>
#include  <\aurora\common\h\System_ID.h>
#include  <\aurora\common\h\Params.h>

struct MapEntry {
    SID     name;
    char        symbol [NAMELEN];
    Descriptor* metaobject;

    // initializing
    MapEntry (SID nam, char* pSymbol, Descriptor* pMeta);
};

#endif  /* _MapEntry_h_DEFINED */
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#ifndef _Descriptor_h_DEFINED
#define _Descriptor_h_DEFINED

#include  <\aurora\hardware\h\CPU.h>
#include  <\aurora\hardware\h\Types.h>
#include  <\aurora\common\h\System_ID.h>

enum MZeroState { DORMANT, RUNNING, CALLING, SENDING, REPLYING, WAIT };

class Descriptor : public Link {
private:
    SID         name;
    MZeroState  state;
    SID         activity;
    longword    reflect;
    Descriptor* sender;
    union {     MZeroMsg*   message;
                Continuation*   continuation;
    } m;

private:
    // CPU architecture dependent
    CPUMode mode;
    Stack*  stack;

private:
    // private routines
    void    execMessage (register MZeroMsg* pMsg);
    void    execContinuation (register Continuation* pCont);
    void    senderContinue ();

public:
    // initializing & cleaning up
    Descriptor (SID nam, MZeroState stat, SID act, longword ref,
           CPUMode mo, Stack* ps);
    inline  ~Descriptor ();
    // accessing instances
    inline SID          Name ();
    inline void         SetState (MZeroState stat);
    inline MZeroState   GetState ();
    inline void         SetActivity (SID act);
    inline SID          GetActivity ();
    inline void         SetSender (Descriptor* pSender);
    inline Descriptor*  GetSender ();
    inline void         GetMZeroDescriptor
                                (register MZeroDescriptor* pDescrip);

    // subroutines for Call(), Send(), and Reply()
    mError  Call (MZeroMsg* pMsg, Descriptor* pSender);
    mError  Send (MZeroMsg* pMsg);
    mError  Reply (Continuation* pCont);

    // executing the medthod specified in the message
    mError  Execute ();
};
```

```
inline Descriptor::~Descriptor ()
{
    if (state == REPLYING && m.continuation)
        delete m.continuation;
}

inline SID Descriptor::Name ()
{
    return (name);
}

inline void Descriptor::SetState (MZeroState stat)
{
    state = stat;
}

inline MZeroState Descriptor::GetState ()
{
    return (state);
}

inline void Descriptor::SetActivity (SID act)
{
    activity = act;
}

inline SID Descriptor::GetActivity ()
{
    return (activity);
}

inline void Descriptor::SetSender (Descriptor* pSender)
{
    sender = pSender;
}

inline Descriptor* Descriptor::GetSender ()
{
    return (sender);
}

inline void Descriptor::GetMZeroDescriptor (register MZeroDescriptor* pDescrip)
{
    pDescrip -> name = name;
    pDescrip -> reflect = reflect;
}

#endif  /* _Descriptor_h_DEFINED */
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#ifndef _MCoreDescriptor_h_DEFINED
#define _MCoreDescriptor_h_DEFINED

#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\common\h\activity.h>
#include  <\aurora\mcore\h\MetaCore.h>

#include  <\aurora\mcore\h\MCoreError.h>

class MCoreDescriptor;

   struct headerDescriptor {
        AID                 name;
        Activity            context;
        Activity            senderContext;
        MCoreDescriptor*    senderDescriptor;
        Boolean             external;
        AID                 continuation;
   };

class MCoreDescriptor {
   private:
     headerDescriptor   header;
     byte           body [1];

   public:
     // creating & destroying
     inline void*   operator new (size_t);
     inline void*   operator new (size_t, void* pBase);
     inline void    operator delete (void*);
     // initializing
     MCoreDescriptor ();
     inline ~MCoreDescriptor ();
     // accessing instances
     inline void        SetName (AID nam);
     inline AID         GetName ();
     inline void        SetContext (Activity ctxt);
     inline Activity    GetContext ();
     inline void        SetSender (Activity ctxt,MCoreDescriptor* pSender);
     inline void        SetSender (MCoreDescriptor* pSender, AID cont);
     inline Activity    GetSenderContext ();
     inline MCoreDescriptor*    GetSenderDescriptor ();
     inline AID         GetContinuation ();
     inline byte*       GetBody ();
     // testing
     inline Boolean IsExternal ();
     // control execution
     inline mcError Run (longword selector, void* pMsg);
};
```

```
inline void* MCoreDescriptor::operator new (size_t)
{
    return (NULL);
}

inline void* MCoreDescriptor::operator new (size_t, void* pBase)
{
    return (pBase);
}

inline void MCoreDescriptor::SetName (AID nam)
{
    header.name = nam;
}

inline AID MCoreDescriptor::GetName ()
{
    return (header.name);
}

inline void MCoreDescriptor::SetContext (Activity ctxt)
{
    header.context = ctxt;
}

inline void MCoreDescriptor::SetSender (Activity ctxt, MCoreDescriptor* pSender)
{
    header.senderContext = ctxt;
    header.senderDescriptor = pSender;
    header.external = FALSE;
    (void) header.continuation.MakeInvalid ();
}

inline void MCoreDescriptor::SetSender (MCoreDescriptor* pSender, AID cont)
{
    header.senderContext = NULL;
    header.senderDescriptor = pSender;
    header.external = TRUE;
    header.continuation = cont;
}

inline Activity MCoreDescriptor::GetSenderContext ()
{
    return (header.senderContext);
}

inline MCoreDescriptor* MCoreDescriptor::GetSenderDescriptor ()
{
    return (header.senderDescriptor);
}

inline AID MCoreDescriptor::GetContinuation ()
{
    return (header.continuation);
}

inline byte* MCoreDescriptor::GetBody ()
{
    return (body);
}
```

```
inline Boolean MCoreDescriptor::IsExternal ()
{
    return (header.external);
}

inline mcError MCoreDescriptor::Run (longword selector, void* pMsg)
{
    MessageR msgR (header.context, selector, pMsg);
    return (msgR.Act ().status);
}

#endif  /* _MCoreDescriptor_h_DEFINED */
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Boolean IsMatched (register MCoreCache* pEntry, MCoreCallMsg* pMsg)
{
    if (pEntry && pMsg &&
        pEntry -> receiver == pMsg -> receiver &&
        pEntry -> selector == pMsg -> selector) {
        if (pEntry -> address) {
            (void) pEntry -> address (pMsg -> message);
            return (TRUE);
        }
    }
    return (FALSE);
}

// Should design nice hash function and hash size,
// based on some experience.
longword hash (AID receiver, longword selector)
{
    return (receiver.Hash () ^ selector) % CACHESIZE;
}

longword HashFunc (register MCoreCache*, MCoreCallMsg* pMsg)
{
    return hash(pMsg -> receiver, pMsg -> selector) * sizeof(MCoreCache);
}

cError MCore::Call (AID receiver, longword selector, void* pMsg)
{
    Activity                sender;
    register MCoreDescriptor*   pRec;

    sender = ActiveContext () -> GetLast ();
    pRec = GetDescriptor (receiver);
    if (receiver == pRec -> GetName ()) {
        (void) mcacheEntry[hash(receiver, selector)].Initialize (
            pRec -> GetContext () -> GetCachedEntry (selector),
            receiver, selector);
        (void) mcache.Initialize (mcacheEntry,
            (Matched) IsMatched, (Hash) HashFunc);
        (void) sender -> SetCache (&mcache);
        (void) pRec -> SetSender (sender, active);
        active = pRec;
        if (pRec -> Run (selector, pMsg) == mcSUCCESS) {
            // never reach here upon success
            return (cSUCCESS);
        } else {
            active = pRec -> GetSenderDescriptor ();
            return (cNOTRUN);
        }
    } else {
        return (cNOTFOUND);
    }
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mError MCore::Deliver (AID receiver, longword selector,
                                        void* pMsg,AID continuation)
{
    register MCoreDescriptor*   pRec;

    pRec = GetDescriptor (receiver);
    if (receiver == pRec -> GetName ()) {
        (void) pRec -> SetSender (active, continuation);
        active = pRec;
        if (pRec -> Run (selector, pMsg) == mcSUCCESS) {
            // never reach here upon success
            return (mSUCCESS);
        } else {
            active = pRec -> GetSenderDescriptor ();
            return (mRFAIL);
        }
    } else {
        return (mNOTFOUNDOBJ);
    }
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\MCore\h\desc.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
inline Descriptor::~Descriptor ()
{
    if (state == REPLYING && m.continuation)
        delete m.continuation;
}

inline SID Descriptor::Name ()
{
    return (name);
}

inline void Descriptor::SetState (MZeroState stat)
{
    state = stat;
}

inline MZeroState Descriptor::GetState ()
{
    return (state);
}

inline void Descriptor::SetActivity (SID act)
{
    activity = act;
}

inline SID Descriptor::GetActivity ()
{
    return (activity);
}

inline void Descriptor::SetSender (Descriptor* pSender)
{
    sender = pSender;
}

inline Descriptor* Descriptor::GetSender ()
{
    return (sender);
}

inline void Descriptor::GetMZeroDescriptor (register MZeroDescriptor* pDescrip)
{
    pDescrip -> name = name;
    pDescrip -> reflect = reflect;
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

static void epilogue ()
{
    MessageM msgM (MCORE_REPLY, NULL, FALSE);
    (void) msgM.Act ();
    // never reach here upon success
}

extern "C" void
_ExecNew (ExecNewMsg* pMsg)
{
    pMsg -> status = PExec -> New (pMsg -> attribute, pMsg -> newActivity);
    (void) epilogue ();
}

extern "C" void
_ExecDelete (ExecDeleteMsg* pMsg)
{
    pMsg -> status = PExec -> Delete (pMsg -> activity);
    (void) epilogue ();
}

extern "C" void
_ExecRun (ExecRunMsg* pMsg)
{
    pMsg -> status = PExec -> Run (pMsg -> activity, pMsg -> priority,
        pMsg -> selector, pMsg -> message);
    (void) epilogue ();
}

extern "C" void
_ExecStop (ExecStopMsg* pMsg)
{
    pMsg -> status = PExec -> Stop (pMsg -> activity);
    (void) epilogue ();
}

extern "C" void
_ExecTop (ExecTopMsg* pMsg)
{
    pMsg -> status = PExec -> Top (pMsg -> activity, pMsg -> priority);
    (void) epilogue ();
}

extern "C" void
_ExecChangeAttribute (ExecChangeAttributeMsg* pMsg)
{
    pMsg -> status = PExec -> ChangeAttribute (pMsg -> activity,
        pMsg -> attribute);
    (void) epilogue ();
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\common\h\System_ID.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
mError MZero::reschedule ()
{
    Descriptor* pMeta;
    mError      error;

    if (pMeta = (Descriptor*) ready -> RemoveFirst ()) {
        error = pMeta -> Execute ();
        // this code will never execute when success
        (void) ready -> AddLast (pMeta);
        return (error);
    }

    // There is no activities being ready to run.

    MessageM  msgM (MCORE_EXIT, NULL, FALSE);
    do
        (void) msgM.Act ();
    while (TRUE);
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

extern "C" void
_ExecNew_C (ExecNewMsg* pMsg)
{
    pMsg -> status = PExec -> New (pMsg -> attribute, pMsg -> newActivity);
}


extern "C" void
_ExecDelete_C (ExecDeleteMsg* pMsg)
{
    pMsg -> status = PExec -> Delete (pMsg -> activity);
}


extern "C" void
_ExecRun_C (ExecRunMsg* pMsg)
{
    pMsg -> status = PExec -> Run (pMsg -> activity, pMsg -> priority,
        pMsg -> selector, pMsg -> message);
}


extern "C" void
_ExecStop_C (ExecStopMsg* pMsg)
{
    pMsg -> status = PExec -> Stop (pMsg -> activity);
}


extern "C" void
_ExecTop_C (ExecTopMsg* pMsg)
{
    pMsg -> status = PExec -> Top (pMsg -> activity, pMsg -> priority);
}


extern "C" void
_ExecChangeAttribute_C (ExecChangeAttributeMsg* pMsg)
{
    pMsg -> status = PExec -> ChangeAttribute (pMsg -> activity,
        pMsg -> attribute);
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\common\h\System_ID.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
xError
MZero::newActivity (Attribute* pAttribute, AID* pNewActivity)
{
    static ExecNewMsg   msgExec;
    MCoreCallMsg        msgCall (MCExec, EXEC_NEW, &msgExec);
    MessageM            msgM (MCORE_CALL, &msgCall, TRUE);

    msgExec.attribute = pAttribute;
    msgExec.newActivity = pNewActivity;
    if (msgM.Act ().status == mcSUCCESS) {
        return (msgExec.status);
    } else {
        return (xMFAILED);
    }
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Global definitions for MCore

class           Exec;
class           MZero;
class           Namer;
class           Idle;
class           Promise;

Exec*           PExec;
MZero*          PMZero;
Namer*          PNamer;
Idle*           PIdle;
Promise*        PPromise;

AID         MCMCore;
AID         MCExec;
AID         MCMZero;
AID         MActivityer;
AID         MCPager;
AID         MCIdle;
AID         MCIdleActivity;
AID         MCPromise;
AID         MCPromiseActivity;
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
extern "C" void
_IdleRun (IdleMsg* pMsg)
{
    PIdle -> Run ();
    // never reach here
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

MCoreDescriptor::MCoreDescriptor ()
{
    (void) header.name.MakeInvalid ();
    header.context = NULL;
    header.senderContext = NULL;
    header.senderDescriptor = NULL;
    (void) header.continuation.MakeInvalid ();
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

MCore   Self;

#include <MetaLib.h>
#include <MetaNews.h>
static void
metaEpilogue (MZeroMsg* pMsg)
{
    //MZeroExitMsg msgExit;
    //(void) msgExit.Exit ();
    pMsg -> Reply ();
    // never reach here upon success
}

static void
objEpilogue ()
{
    MessageR msgR (NULL, UNDEF, NULL);
    (void) msgR.Act ();
    // never reach here upon success
}

extern "C" void
Prologue (void* pMsg, longword selector)
{
#if 1
extern void InitDebug (void);
InitDebug ();
#endif
    register Activity   context;

    // initializing EntryTable
    context = ActiveContext ();
//  SET CPUREG (context, rGP, GetGlobal ());
    (void) context -> SetEntryTable (new (context + 1)
        EntryTable (MCORE_NSELECTOR));
    (void) context -> SetEntry (MCORE_REPLY, (Entry) _MCoreReply);
    (void) context -> SetEntry (MCORE_CALL, (Entry) _MCoreCall);
    (void) context -> SetEntry (MCORE_SEND, (Entry) _MCoreSend);
    (void) context -> SetEntry (MCORE_EXIT, (Entry) _MCoreExit);
    (void) context -> SetEntry (MCORE_DELIVER, (Entry) _MCoreDeliver);
    (void) context -> SetEntry (MCORE_GETCTXT, (Entry) _MCoreGetContext);

    // invoking MCore::Prologue ()
    (void) Self.Prologue ((MCoreRestoreMsg*) pMsg);

#if 0
    // dispatching the method specified in the argument
    (void) (*context -> GetEntry (selector)) (pMsg);
#endif
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

extern "C" void
_MCoreReply ()
{
    (void) Self.Reply ();
    (void) objEpilogue ();
}

extern "C" void
_MCoreCall (MCoreCallMsg* pMsg)
{
    pMsg -> status = Self.Call (pMsg -> receiver, pMsg -> selector,
        pMsg -> message);
    (void) objEpilogue ();
}

extern "C" void
_MCoreSend (MCoreSendMsg* pMsg)
{
    pMsg -> status = Self.Send (pMsg -> receiver, pMsg -> selector,
        pMsg -> message);
    (void) objEpilogue ();
}

extern "C" void
_MCoreExit ()
{
    (void) Self.Exit ();
    (void) objEpilogue ();
}

extern "C" void
_MCoreDeliver (register MZeroMsg* pMsg)
{
    pMsg -> status = Self.Deliver (pMsg -> receiver, pMsg -> selector,
        pMsg -> message, pMsg -> continuation);
    (void) metaEpilogue (pMsg);
}

extern "C" void
_MCoreGetContext (register MCoreGetContextMsg* pMsg)
{
    pMsg -> status = Self.GetContext (pMsg -> object, &pMsg -> context);
    (void) objEpilogue ();
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

extern "C" void
_NamerSetOAD (NamerMessage* pMsg)
{
    (void) PNamer -> SetOAD (pMsg -> AID, pMsg -> oad);
    (void) epilogue ();
}

extern "C" void
_NamerOIDof (NamerMessage* pMsg)
{
    pMsg -> oid = PNamer -> OIDof (pMsg -> AID);
    (void) epilogue ();
}

extern "C" void
_NamerOADof (NamerMessage* pMsg)
{
    pMsg -> oad = PNamer -> OADof (pMsg -> AID);
    (void) epilogue ();
}

extern "C" void
_NamerAIDof (NamerMessage* pMsg)
{
    pMsg -> AID = PNamer -> AIDof (pMsg -> oid);
    (void) epilogue ();
}

extern "C" void
_NamerAIDofLAD (NamerMessage* pMsg)
{
    pMsg -> AID = PNamer -> AIDofLAD (pMsg -> lad);
    (void) epilogue ();
}

extern "C" void
_NamerOADofOID (NamerMessage* pMsg)
{
    pMsg -> oad = PNamer -> OADofOID (pMsg -> oid);
    (void) epilogue ();
}

extern "C" void
_NamerInitAID (NamerMessage* pMsg)
{
    pMsg -> status = PNamer
        -> InitAID (pMsg -> AID, pMsg -> oid, pMsg -> oad);
    (void) epilogue ();
}
```

```
extern "C" void _NamerNewID_C (NamerMessage* pMsg)
{
    pMsg -> AID = PNamer -> NewID ();
}

extern "C" void _NamerNewIDAddr_C (NamerMessage* pMsg)
{
    pMsg -> status = PNamer -> NewIDAddr (pMsg -> AID);
}

extern "C" void _NamerDeleteID_C (NamerMessage* pMsg)
{
    (void) PNamer -> DeleteID (pMsg -> AID);
}

extern "C" void _NamerAddID_C (NamerMessage* pMsg)
{
    pMsg -> AID = PNamer -> AddID (pMsg -> oid, pMsg -> time);
}

extern "C" void _NamerSubID_C (NamerMessage* pMsg)
{
    (void) PNamer -> SubID (pMsg -> AID);
}

extern "C" void _NamerResetOAD_C (NamerMessage* pMsg)
{
    (void) PNamer -> ResetOAD (pMsg -> AID);
}

extern "C" void _NamerSetOAD_C (NamerMessage* pMsg)
{
    (void) PNamer -> SetOAD (pMsg -> AID, pMsg -> oad);
}

extern "C" void _NamerOIDof_C (NamerMessage* pMsg)
{
    pMsg -> oid = PNamer -> OIDof (pMsg -> AID);
}

extern "C" void _NamerOADof_C (NamerMessage* pMsg)
{
    pMsg -> oad = PNamer -> OADof (pMsg -> AID);
}

extern "C" void _NamerAIDof_C (NamerMessage* pMsg)
{
    pMsg -> AID = PNamer -> AIDof (pMsg -> oid);
}

extern "C" void _NamerAIDofLAD_C (NamerMessage* pMsg)
{
    pMsg -> AID = PNamer -> AIDofLAD (pMsg -> lad);
}

extern "C" void _NamerOADofOID_C (NamerMessage* pMsg)
{
    pMsg -> oad = PNamer -> OADofOID (pMsg -> oid);
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\common\h\System_ID.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

xError
MZero::deleteActivity (AID activity)
{
    static ExecDeleteMsg    msgExec;
    MCoreCallMsg        msgCall (MCExec, EXEC_DELETE, &msgExec);
    MessageM        msgM (MCORE_CALL, &msgCall, TRUE);

    msgExec.activity = activity;
    if (msgM.Act ().status == mcSUCCESS) {
        return (msgExec.status);
    } else {
        return (xMFAILED);
    }
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void MCore::Reply ()
{
    register MCoreDescriptor*   pOldActive;

    if (pOldActive = active) {
        if (pOldActive -> IsExternal ()) {
            (void) externalReply (pOldActive);
        } else {
            (void) localReply (pOldActive);
        }
        // never reach here under success
    }

    MZeroExitMsg msgExit;
    (void) msgExit.Exit ();
    // never reach here upon success
    Activity* pActivity;
    if (PExec -> Next (&pActivity) == xSUCCESS) {
        (void) pActivity -> Run ();
    }
}

void MCore::localReply (register MCoreDescriptor* pReply)
{
    if (pReply -> GetSenderContext ()) {
        MCoreDescriptor*    pOldDescriptor = pReply
                        -> GetSenderDescriptor ();
        MessageR        msgR (pReply -> GetSenderContext (),
                        UNDEF, NULL);

        (void) pReply -> SetSender (NULL, NULL);
        active = pOldDescriptor;
        (void) msgR.Act ();
        // never reach here upon success
        active = pReply;
        (void) pReply -> SetSender (msgR.context, pOldDescriptor);
    }
}

void  MCore::externalReply (register MCoreDescriptor* pReply)
{
    static AID  invalid;
    MZeroMsg    msgReply (invalid, UNDEF, NULL,
                pReply -> GetContinuation ());

    (void) msgReply.Reply ();
}
```

```
// AURORA Operating System - (C) Copyright INE-UFSC/II-UFRGS
//
// Author  Do. Luiz Carlos Zancanella
// Advisor Dr. Philippe Olivier Alexandre Navaux
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include  <\aurora\mcore\h\MCoreError.h>

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
cError MCore::Send (AID receiver, longword selector, void* pMsg)
{
    Activity                sender;
    register MCoreDescriptor*   pRec;
    extern Boolean          IsMatched (MCoreCache*, MCoreCallMsg*);
    extern longword         HashFunc (MCoreCache*, MCoreCallMsg*);
    extern longword         hash (AID, longword);

    sender = ActiveContext () -> GetLast ();
    pRec = GetDescriptor (receiver);
    if (receiver == pRec -> GetName ()) {
        /*
         * dumb overwriting cache
         */
        (void) mcacheEntry[hash(receiver, selector)].Initialize (
            pRec -> GetContext () -> GetCachedEntry (selector),
            receiver, selector);
        (void) mcache.Initialize (mcacheEntry, (Matched) IsMatched,
                        (Hash) HashFunc);
        (void) sender -> SetCache (&mcache);
        (void) pRec -> SetSender (NULL, active);
        active = pRec;
        if (pRec -> Run (selector, pMsg) == mcSUCCESS) {
            // never reach here upon success
            return (cSUCCESS);
        } else {
            active = pRec -> GetSenderDescriptor ();
            return (cNOTRUN);
        }
    } else {
        return (cNOTFOUND);
    }
}
```

**Informática**  UFRGS

**CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

*"Estrutura Reflexiva para Sistemas Operacionais Multiprocessados"*

por

Luiz Carlos Zancanella

Tese apresentada aos Senhores:

Prof. Dr. Edil Severiano Tavares Fernandes (COPPE/UFRJ)

Prof. Dr. José Lucas Mourão Rangel Netto (PUC-RJ)

Profa. Dra. Maria Lúcia Blanck Lisbôa

Prof. Dr. Cláudio Fernando Resin Geyer

Vista e permitida a impressão.
Porto Alegre, 28/09/98.

Prof. Dr. Philippe Olivier Alexandre Navaux,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
Coordenadora do Curso de Pós-Graduação
em Ciência da Computação
Instituto de Informática - UFRGS