UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Roger Kreutz Immich

Modelo de um Núcleo de Sistema Operacional Extensível utilizando Reflexão Computacional

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

Prof. Dr. Luiz Carlos Zancanella

Florianópolis, Agosto de 2006

Modelo de um Núcleo de Sistema Operacional Extensível utilizando Reflexão Computacional

Roger Kreutz Immich

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

Prof. Dr. Raul Sidnei Wazlawick

Banca Examinadora

Prof. Dr. Luiz Carlos Zancanella

Prof. Dr. Carlos Barros Montez

Prof. Dr. Antônio Augusto Medeiros Fröhlich

Prof. Dr. Mário Antonio Ribeiro Dantas

“Perfection is not achieved when there is nothing left to add, but when there is nothing left to take away”
Antoine de St. Exupery

“Two things are infinite: the universe and human stupidity; and I’m not sure about the universe.”
Albert Einstein

“Aos meus familiares e amigos...”

Agradecimentos

Primeiramente, gostaria de agradecer a minha família, em especial ao meu pai **Aloísio Immich**, a minha mãe **Jacinta Kreutz Immich** e a minha irmã, **Larissa Kreutz Immich**, que sempre me apoiaram nos estudos e na vida e da qual eu recebi e continuo recebendo muita força, esperança, carinho e tudo mais que além pudesse desejar.

Gostaria de agradecer também ao LaCPaD e ao **Prof. Luís Fernando Friedrich** que me "abrigaram" no meu primeiro ano de mestrado e da mesma forma ao LISHA e ao **Prof. Antônio Augusto Medeiros Fröhlich**, aos quais só tenho a agradecer pela força e incentivo, principalmente na reta final do trabalho.

Por último, mas não menos importante, gostaria de agradecer ao meu orientador, o **Prof. Luiz Carlos Zancanella** que me ofereceu a oportunidade de ingressar no mestrado, e pelas longas conversas e ensinamentos sobre sistemas operacionais e reflexão computacional, sem as quais não seria possível o desenvolvimento deste trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

A concepção de computadores cada vez mais poderosos, com mais recursos e funcionalidades impulsionou uma significativa evolução no desenvolvimento de sistemas operacionais. Estes sistemas, com o objetivo de prover acesso aos dispositivos, implementam uma complexa abstração do *hardware*, permitindo que as aplicações sejam projetadas em uma camada de alto nível, facilitando o desenvolvimento e aumentando a portabilidade. Esta abordagem é eficiente nos casos citados acima, porém ela produz um gerenciador de recursos fortemente centralizado, que pode entrar em conflito com as necessidades específicas das aplicações, limitando-as tanto em performance quanto em flexibilidade, devido ao fato de que a aplicação precisa se adaptar ao ambiente de execução. De acordo com autores conceituados, a necessidade da adaptação do sistema operacional em relação a aplicação é cada vez mais evidente e somente desta forma será possível oferecer um ambiente especializado de acordo com as necessidades específicas de cada uma delas. O modelo proposto neste trabalho, visa suprir estas necessidades, oferecendo a possibilidade da modificação do ambiente de execução através de meta-informações passadas pelas aplicações no momento da sua inicialização ou dinamicamente durante a sua execução. Através das simulações realizadas, foi provado que é possível a concepção de tal arquitetura, contudo ainda é muito dependente de recursos que estão sendo desenvolvidos e aprimorados, como a máquina virtual Java.

Palavras-chave: sistemas operacionais extensíveis, reflexão computacional, núcleo extensível

Abstract

The increase conception of more powerful computers, with better resources and functionalities, began to stimulate a significant evolution in the operation system development. These systems provide devices access by implementing one complex hardware abstraction layer, allowing that the applications can be developed in a high level layer, which help to quick development and portability increase. This strategy is efficient in the cited cases above, however it produces a strong centered resources management, which can conflict with the application specific necessities, limiting them in performance and flexibility, because the application has to adapt yourself to an execution environment. Some authors with appraised, has been talk about the necessity of the operational system adaptation in relation to the application and it have been more evident, maybe this could be the only way to offer a specialized environment in agreement with the application specific necessities. The approach present in this thesis, aims at to supply dynamically these necessities, offering the modification possibility in the environment through meta-information passed by the applications at the load-time or run-time. Through simulations, was demonstrated that the conception of such architecture is possible, however this approach still dependent of the resources that are being in development and improvement, as the Java virtual machine.

Keywords: extensible operational systems, computation reflection, extensible kernel

Capítulo 1

Introdução

Avanços na tecnologia de desenvolvimento do hardware estão permitindo a concepção de computadores cada vez mais poderosos, utilizando processadores mais rápidos, redes de altíssima velocidade, disponibilizando grande espaço de armazenamento e novos recursos têm sido adicionados constantemente. Conforme [AND 93], a cada década é possível prever um crescimento de pelo menos duas ordens de magnitude em cada uma dessas áreas.

Toda essa tecnologia impulsionou uma significativa evolução e aprimoramento dos sistemas operacionais existentes. Assim como o número e a capacidade de dispositivos físicos estão em constante crescimento, a complexidade do sistema, que tem por obrigação gerenciá-lo, deve acompanhar este desenvolvimento. Com o intuito de não repassar essa complexidade para as aplicações é utilizada uma abrangente abstração do *hardware* possibilitando que as aplicações sejam desenvolvidas em uma camada de alto nível. Desta forma é possível aumentar a portabilidade e a facilidade de diferentes implementações.

Esta abordagem é eficiente na redução da complexidade do sistema, porém ocasiona um grande problema, ela produz um gerenciador de recursos fortemente centralizado. Esta centralização pode entrar em conflito com as necessidades específicas das aplicações, limitando-as tanto em desempenho quanto em flexibilidade, pois não está sendo oferecido o ambiente ideal para sua execução e também não existe a possibilidade de que a aplicação adapte esse ambiente as suas necessidades, devendo ela se adaptar ao ambiente.

De acordo com Kiczales [KIC 93] a necessidade da adaptação do sistema operacional em relação a aplicação que esta executando é cada vez mais evidente, pois somente com políticas e mecanismos de manipulação adequados esta poderá utilizar os recursos físicos da melhor maneira possível, desta forma, ao invés da aplicação se adaptar ao sistema, o sistema se adaptaria

as aplicações em execução oferecendo um ambiente especializado de acordo com às necessidades específicas de cada uma delas.

Um sistema operacional adaptável é aquele que consegue modificar a si mesmo em uma determinada circunstância com o objetivo de ganhar algumas funcionalidades ou melhorar seu desempenho. Dentro da categoria dos sistemas adaptáveis, estão os sistemas operacionais extensíveis, que permitem além da modificação das suas estruturas e funcionalidades a adição de novos serviços que não estavam presentes durante o processo de desenvolvimento. Um dos grandes problemas é definir quais aspectos devem ser adaptados e com que finalidade[COW 96].

Por exemplo, em sistemas embarcados que necessitam de grande mobilidade, uma das alternativas possíveis para a melhor utilização dos recursos físicos poderia ser alcançada através da economia de energia ou do gerenciamento eficiente da mesma, permitindo assim, uma maior autonomia aos dispositivos. Diferentemente de um sistema que está constantemente ligado a uma fonte de energia, onde a melhor utilização dos recursos físicos poderia ser obtida através do aumento do desempenho, mesmo que para isto, gastasse mais energia.

O contexto onde a aplicação será executada é um dos fatores determinantes para o tipo de adaptação que o sistema irá realizar. Como citado anteriormente, estes fatores poderiam ser a economia de energia, a disponibilidade no caso de servidores importantes, o aumento do desempenho, entre outros. Isto representa uma grande quantidade de informações que o sistema operacional teria que processar para decidir qual adaptação realizaria, impondo uma sobrecarga desnecessária. Desta forma, uma das soluções encontradas foi passar este nível de conhecimento para aplicação, sendo que estas devem informar ao sistema as suas necessidades.

Na proposta apresentada nesta dissertação, foi realizada uma utilização conjunta dos paradigmas de orientação a objetos e reflexão computacional, bem como os conceitos relativos ao sistema operacional Aurora, para desenvolver um modelo de um núcleo de sistema operacional extensível. A orientação a objetos e a reflexão computacional serviram não somente como base para o desenvolvimento, mas também com a função de separação conceitual, pois todo o sistema foi projetado pensando-se no conceito de objetos e meta-objetos.

A reflexão computacional também foi utilizada para prover um ambiente onde fosse possível obter ou receber informações sobre o sistema e traduzi-las em ações, que refletem no comportamento do mesmo, possibilitando que adaptações sejam realizadas de forma dinâmica. A estrutura organizacional do Aurora foi utilizada para modelagem, utilizando-se conceitos como, meta-objetos e meta-espaços que realizam papéis fundamentais neste modelo.

Dentro da arquitetura proposta, foi disponibilizada uma forma de comunicação

entre as aplicações e o sistema operacional. Esta comunicação possibilita que as aplicações em execução adaptem o sistema as suas necessidades específicas alocando espaço somente para o que será utilizado. O modelo também prove a capacidade de adição de serviços que não estavam presentes no projeto do sistema, especializando ainda mais o ambiente de execução.

Ainda no capítulo de introdução é descrita uma das forma atuais de desenvolvimento de sistemas operacionais, bem como os problemas encontrados nela e o porque da necessidade de sistemas que adaptem-se aos requisitos das aplicações. O restante desta dissertação está organizado da seguinte forma. No Capítulo 2 são apresentadas as principais arquiteturas de organização de núcleos de sistemas operacionais, com suas respectivas vantagens e desvantagens em relação a sistemas de uso geral. No Capítulo 3 são descritas as técnicas utilizadas para alcançar a extensibilidade, como o paradigma de orientação a objetos, a reflexão computacional, a separação de conceitos, máquinas virtuais e a linguagem Java. Ainda no mesmo capítulo é apresentada a taxonomia e um resumo sobre alguns sistemas existentes e suas principais características. No Capítulo 4 é apresentado o modelo desenvolvido, com uma descrição detalhada da sua implementação e com os resultados obtidos da sua simulação. E por último no Capítulo 5 é realizada a conclusão do trabalho.

1.1 Necessidade de Sistemas Operacionais extensíveis

A necessidade da adaptação do sistema operacional em relação a aplicação que esta executando é cada vez mais evidente, pois somente com políticas e mecanismos de manipulação adequados esta poderá utilizar os recursos físicos da melhor forma possível [KIC 93].

1.1.1 Desing de Sistemas Operacionais de propósito geral

O desenvolvimento da maioria dos sistemas operacionais é baseado em duas premissas da engenharia de software que são: decomposição e abstração[BER 98]. Com objetivo de diminuir a complexidade e prover uma camada de abstração para os desenvolvedores, os componentes dos sistemas são divididos em módulos bem definidos que interagem uns com os outros através de interfaces que escondem os detalhes de suas implementações.

Com a união destas duas técnicas, são produzidas grandes “caixas pretas” (Figura 1.1) que facilitam o desenvolvimento de sistemas complexos através da união de pequenos componentes sem precisar saber de que forma eles forma implementados, apenas realizando cha-

madas as suas interfaces de abstrações sendo que toda a implementação de interação com o hardware é escondida dentro da caixa preta.

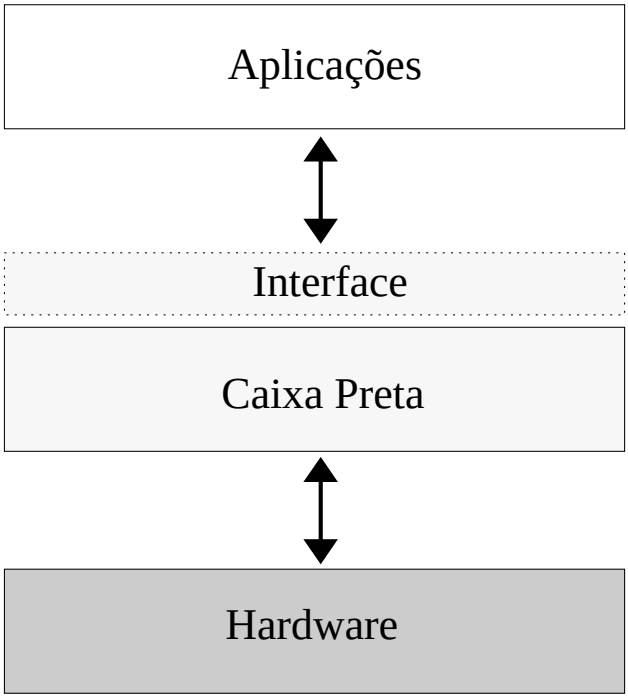


Figura 1.1: Implementação através de decomposição e abstração

Como já foi mencionada esta é uma técnica bastante empregada no desenvolvimento de sistemas complexos, inclusive: sistemas operacionais, sistemas de tempo real, máquinas virtuais, entre outros. Desta forma são escondidas as características intrínsecas da manipulação do hardware e apenas disponibilizada uma interface com as funcionalidades necessárias. Com essa abordagem, é possível aplicar as técnicas de reuso de software, reduzindo o tempo de desenvolvimento e a complexidade empregada.

Apesar dos benefícios oferecidos por esta técnica, existem problemas. Um destes problemas é referente às políticas de gerenciamento de *buffers*. Em sistemas de tempo real especializados, a aplicação reserva a quantia de buffers que ela precisará, no entanto, em um sistema de uso geral, isso não é possível, pois existem outras aplicações que estarão concorrendo pelos mesmos recursos então será preciso utilizar uma política de gerenciamento para eles[BER 98]. O grande desafio é definir qual política de gerenciamento agradará a todas as aplicações, tendo em vista que cada aplicação possuí características próprias e muitas vezes exclusivas, não se adequando a uma política de gerenciamento comum.

Com o intuito de amenizar este problema, existem algumas aplicações que es-

crevem suas próprias políticas de gerenciamento. Estas políticas são executadas concorrentemente com as políticas implementadas pelo sistema operacional e tem como objetivo prover um melhor gerenciamento dos recursos que são indispensáveis para sua execução. Porém estas novas políticas estão sujeitas as limitações impostas pela política global do sistema operacional. A sessão 1.1.2 realiza uma melhor definição do problema que esta sendo tratado.

1.1.2 Definição do problema

Os sistemas operacionais de propósito geral provêem uma complexa e abrangente abstração, do hardware permitindo que as aplicações sejam desenvolvidas em uma camada de alto nível com o objetivo de prover maior portabilidade e quantidade de características que são diretamente implementadas pelo hardware[ENG 95].

Como resultado dessa abordagem temos um gerenciador de recursos fortemente centralizado, o que pode conflitar com as necessidades das aplicações limitando-as tanto em performance quanto em flexibilidade, não oferecendo o nível de abstração adequado para a melhor otimização e utilização dos recursos físicos disponíveis, principalmente no caso de programas especializados, que executam tarefas ou rotinas que envolvem acesso direto aos dispositivos [AND 93].

Por exemplo, podemos citar o caso do sistema operacional FreeBSD 4.4, muito utilizado em servidores de Internet que precisam de alto nível de segurança. Este sistema operacional utiliza somente um algoritmo de substituição de páginas na memória cache. Estas políticas são baseadas na combinação de algoritmos da década de 70 e início de 80 que possuem uma boa referencia de localidade[BAB 81], embora sejam pouco eficientes para as novas aplicações como recuperação de informações em banco de dados onde o padrão de acesso a páginas é praticamente aleatório, e multimídia onde o acesso acontece de forma estritamente seqüencial e com variação constante através do tempo[KEA 89].

Quando o gerenciamento da memória cache não é executado eficientemente[1] pelo sistema operacional, umas das alternativas encontradas por aplicações que possuem necessidades específicas e bem definidas é ela própria realizar o gerenciamento desta memória[GIV 06]. Este procedimento aumentará consideravelmente a complexidade da aplicação e seu efeito não será reproduzido no sistema como um todo, pois somente a aplicação que implementou a função de gerenciamento se beneficiará dela.

[1]Eficientemente no sentido de suprir as necessidades específicas de uma aplicação ou de um conjunto de aplicações[STO 81].

Outra característica que pode ser mencionada, devido ao fato de estar presente na forma de implementação dos sistemas operacionais atualmente, é o sistema de arquivos. Estes assumem que a maioria dos acessos a arquivos será realizada de forma seqüencial[KRI 93, OUS 85], sendo otimizado para esta finalidade. Entretanto, importantes classes de aplicações possuem características de acesso a arquivos completamente não seqüências, como já citado acima, nas aplicações de recuperação de informações em banco de dados. Outra vez a aplicação que desejar outra política, deverá implementá-la por conta própria, incorrendo nos mesmo problemas já citados.

Ao invés de delegar as funções do gerenciamento destes recursos físicos para as aplicações, fazendo com que o sistema possua múltiplas implementações para o mesmo fim, e desta forma estando sujeito a erros e aumentando consideravelmente a complexidade da aplicação e do sistema, a melhor solução seria permitir que o próprio sistema operacional tivesse a capacidade de adaptar o seu ambiente e as suas funcionalidades em tempo de execução. Provendo assim a melhor utilização dos recursos disponíveis, seja por uma adaptação automática através da analise da utilização dos seus componentes ou uma adaptação sugerida pela aplicação de acordo com as suas necessidades.

Capítulo 2

Arquitetura de Sistemas Operacionais

De acordo com a já conhecida definição de Tanenbaum[TAN 92] um sistema operacional é “uma camada de software que de forma segura, permite a abstração e multiplexação dos recursos físicos”. A partir desta premissa podemos inferir que a função do sistema operacional é prover uma maior facilidade para o gerenciamento eficiente do hardware, visando uma redução de erros e possibilitando que desenvolvedores implementem suas aplicações sem a necessidade de conhecer todos os detalhes dos recursos físicos que ela utilizará.

Devido a grande variedade e complexidade dos recursos físicos e das possíveis funções apresentadas por eles, uma abordagem para facilitar a utilização e compreensão destes é desejada. Possivelmente a melhor forma de fazer isto é decompor o sistema em partes menores, criando abstrações para cada uma delas[RUM 97]. A escolha das funcionalidades e recursos apresentados por cada abstração é de essencial importância, pois esta será a única forma de acessar os recursos físicos posteriormente.

A Figura 2.1 mostra um modelo simplificado da relação entre hardware, sistema operacional e aplicações. É possível observar que as aplicações não têm acesso direto aos recursos físicos do sistema. Este acesso é obtido através comunicação com uma API(*Application Program Interface*) do sistema operacional, que deve apresentar de forma fácil e organizada as abstrações do sistema, possibilitando a interação entre as aplicações e os recursos físicos disponíveis.

A responsabilidade do sistema operacional é a de gerenciar as funcionalidades apresentadas pelo hardware como: escalonamento de processos, sistemas de arquivos, memória, interfaces de rede, comunicação entre processos, entre outras. Todas estas características devem ser multiplexadas de forma segura, permitindo o acesso simultâneo por diversas aplicações e/ou usuários, sem apresentar inconsistências.

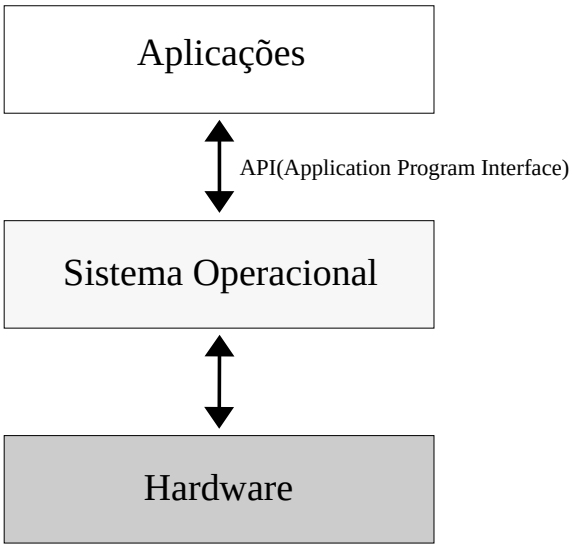


Figura 2.1: Relação entre o hardware, sistema operacional e as aplicações

Uma das partes mais importantes dos sistemas operacionais é o seu *kernel* ou núcleo. O núcleo dos sistemas operacionais é composto por um conjunto de procedimentos responsáveis pela comunicação das aplicações com o *hardware*. Através das APIs, as aplicações utilizam "chamadas de sistemas" para ter acesso a estes procedimentos e realizar as operações desejadas.

Além de servir de interface para as aplicações acessarem os recursos físicos, o núcleo do sistema geralmente controla todas as outras funções do *hardware*, em muitos casos, sendo ele o único elo de ligação entre as funções escritas em software com as funções implementadas fisicamente nos dispositivos.

A forma como o núcleo dos sistemas operacionais são projetados e implementados podem seguir conceitos diferentes, proporcionando vantagens e também possivelmente desvantagens para determinados nichos de aplicações que possuem necessidades diferenciadas. A evolução na forma de implementação e utilização do *kernel* é importante para possibilitar o desenvolvimento de sistemas operacionais com núcleos extensíveis.

As sessões seguintes apresentam um breve relato sobre possíveis formas de organização dos núcleos de sistemas operacionais, descrevendo suas arquiteturas e apresentando vantagens e desvantagens para cada uma delas.

2.1 Arquitetura baseada em Núcleo Monolítico

O núcleo monolítico de um sistema operacional agrega todos os serviços e abstrações do sistema em uma imagem única e inseparável[DEA 00]. Em tempo de projeto este pode ser estruturado e logicamente separado em módulos e rotinas, porém após compilação e geração da imagem todas as funções são alocadas no mesmo espaço de endereçamento e a principio, uma rotina pode chamar qualquer outra(Figura 2.2).

A figura 2.2 apresenta a arquitetura simplificada de um núcleo monolítico de um sistema operacional. Como pode ser observado, todos os procedimentos e funções encontram-se no mesmo espaço de endereçamento (que é carregado na memória principal na inicialização do sistema e permanece lá até a finalização), permitindo que as chamadas sejam executadas muito rapidamente, tornado o *kernel* muito estável, eficiente e rápido.

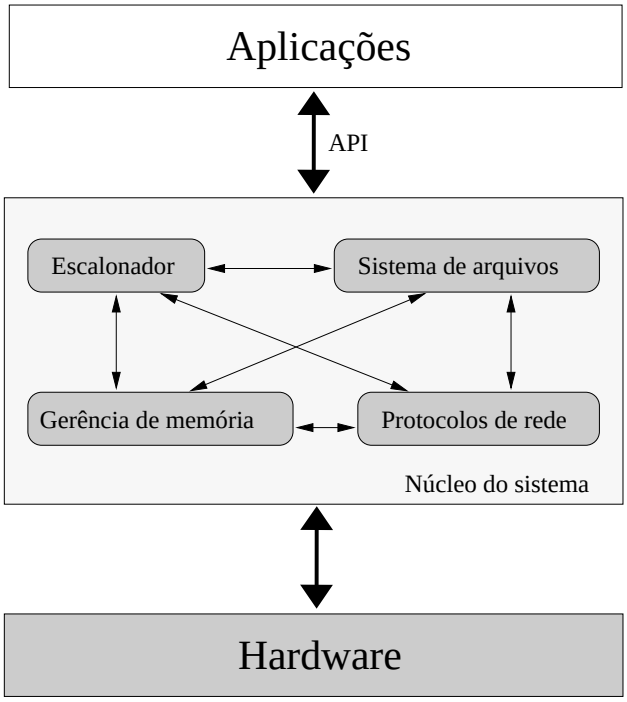


Figura 2.2: Núcleo monolítico de um sistema operacional

Os núcleos monolíticos podem ser interrompíveis ou não-interrompíveis [dO 01]. O conceito de processo não existe dentro do *kernel* monolítico, apenas fluxos de execução, desta forma, um kernel monolítico não-interrompível não recebe nem trata nenhuma interrupção durante o tempo em que esta executando. A vantagem de um sistema como este é a facilidade de implementação, porém existe uma perda de desempenho devido ao não tratamento das interrupções de dispositivos e temporizadores quando estas acontecem.

Um *kernel* monolítico interrompível apresenta um melhor desempenho em relação a um não-interrompível porém apresenta uma maior dificuldade de desenvolvimento e implementação. Quando o núcleo do sistema interrompe sua execução temporariamente para atender outras chamadas podem ser geradas inconsistências no sistema, para evitar este acontecimento é necessária a definição de sessões criticas que somente poderão ser alteradas no momento correto.

A abordagem de núcleos monolíticos prove algumas desvantagens. Uma delas é a falta de proteção entre os componentes, pelo fato deles povoarem o mesmo espaço de endereçamento, uma falha em algum deles poderia comprometer a integridade do sistema inteiro[MAC 02]. Outro problema são as atualizações, uma vez que este tipo de *kernel* não oferece a possibilidade de carregamento de módulos e funções em tempo de execução sendo necessário finalizar o sistema e recompilar novamente com a nova funcionalidade.

2.2 Arquitetura baseada em Microkernel

O conceito por traz do microkernel é implementar somente as características essenciais a nível de núcleo do sistema (no espaço de endereçamento do *kernel*) e todos os outros serviços, abstrações e funcionalidades devem ser implementados no espaço de endereçamento do usuário [CHE 95]. O microkernel executa da mesma fora que um núcleo monolítico, porém as desvantagens do *kernel* monolítico não são tão aparente devido ao tamanho reduzido do microkernel.

A figura 2.3 apresenta a arquitetura simplificada de um sistema operacional baseado em microkernel. Podemos observar que somente os serviços necessários estão alocados no espaço de endereçamento do núcleo do sistema e executam em modo protegido, os outros serviços e abstrações ficam alocados no espaço de endereçamento do usuário. A comunicação entre os serviços alocados a nível de kernel e os alocados a nível de usuário acontece através da SPI(*System Programming Interface*).

A interface SPI é semelhante a API, porém ao invés de intermediar a comunicação entre aplicações e serviços ela intermedeia a comunicação entre os serviços e/ou funcionalidades alocadas no espaço de endereçamento do núcleo do sistema e os serviços e/ou funcionalidades alocadas no espaço de endereçamento do usuário.

Sistemas operacionais baseados na arquitetura de microkernel não serão fisicamente menores (em termos número de linhas de código) que sistemas operacionais baseados em núcleos monolíticos[CLA 00]. O termo “micro” refere-se a um trecho de código que será exe-

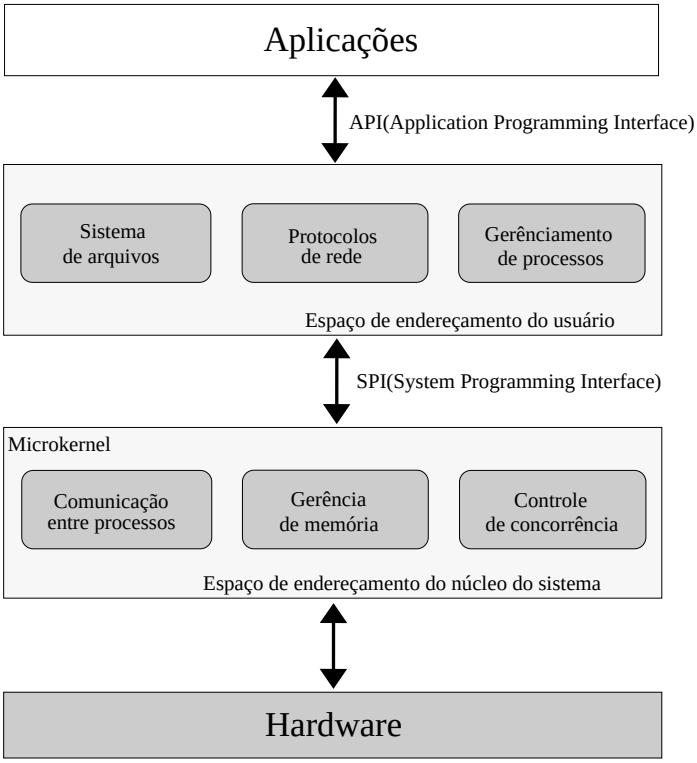


Figura 2.3: Arquitetura de um sistema operacional baseado em microkernel

cutado em modo protegido e alocado no espaço de endereçamento do núcleo do sistema que é consideravelmente menor do que o código que sistemas monolíticos executam em modo protegido.

A abordagem de comunicação adotada por arquiteturas baseadas em microkernels é a troca de mensagens. Os processos passam a ser clientes e servidores, e a comunicação entre eles é gerenciada pelo microkernel[TAN 87]. Desta forma o microkernel encarrega-se de entregar as mensagens para o servidor que deverá atendê-las e posteriormente encaminha os resultados para os clientes que estão esperando.

Levando-se em consideração estas características, a arquitetura baseada em microkernel mostrou-se mais flexível e modular do que núcleos monolíticos. Esta permite que sejam realizadas atualizações e modificações durante a execução do sistema, sem a necessidade de reiniciá-lo[BAC 00]. Assim é possível inicializar o sistema com um conjunto reduzido de serviços, e conforme houver necessidade carregar os serviços adicionais.

Outra vantagem é em relação a segurança e estabilidade do sistema. Nesta arquitetura, as abstrações não ocupam obrigatoriamente o mesmo espaço de alocação e são implementadas independentemente, desta forma, se um processo servidor vir falhar, o restante do sistema pode

continuar a execução normalmente, e ainda o processo servidor que falhou pode ser recarregado e voltar a executar[CAH 96].

A maior desvantagem desta arquitetura é o alto custo das chamadas de sistema, que acabaram tornando-se o gargalo do sistema, pois todas as comunicações são realizadas através do microkernel exigindo um grande poder de processamento para isto e muitas trocas de contexto. Sendo assim, quanto menor o tamanho do *kernel* mais onerosa será sua execução, pois mais mensagens deverão ser gerenciadas por ele.

2.3 Arquitetura baseada em Núcleo Orientado a Objetos

Em uma arquitetura de sistema operacional baseada em um núcleo orientado a objetos o serviços do sistema são implementados como uma coleção de objetos e um conjunto de operações associadas[ZAN 97]. A coleção de objetos é alocada em um espaço protegido, e tem como objetivo encapsular estados privados de informações ou dados. O conjunto de operações associadas tem como objetivo fornecer uma interface para acesso e alteração destes dados.

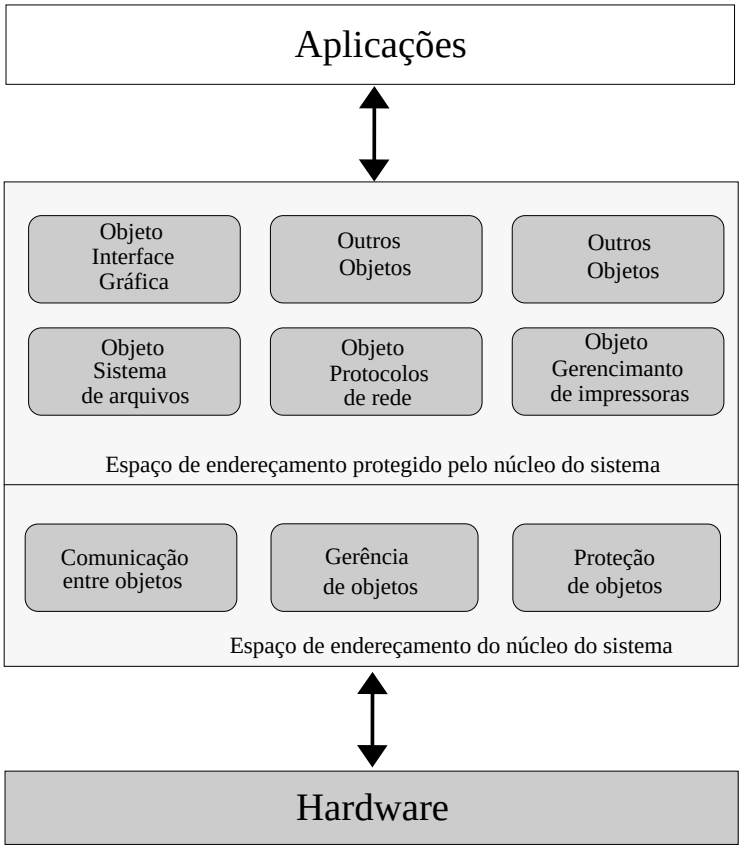


Figura 2.4: Arquitetura de um sistema operacional baseado em núcleo orientado a objetos

A figura 2.4 demonstra a arquitetura simplificada de um sistema operacional com o núcleo orientado a objetos. Como pode ser observado na figura, o núcleo de um sistema baseado nesta arquitetura tem como premissas básicas o gerenciamento de objetos, prover a comunicação entre objetos e proteger os objetos do sistema contra acessos não autorizados[ZAN 94].

Devido a grande quantidade de objetos que este núcleo possivelmente precisará gerenciar é necessária uma grande disciplina para organizar os serviços disponibilizados pelo o sistema operacional. Assim como na arquitetura baseada em microkernel uma das partes mais sensíveis do sistema é a comunicação entre objetos, a implementação desta comunicação pode trazer sucesso ou o fracasso do sistema.

Por utilizar um sistema de comunicação entre objetos na forma de cliente-servidor torna-se mais fácil as atualizações do sistema e também a distribuição destes objetos em vários equipamentos, podendo facilmente este sistema evoluir para um sistema operacional distribuído, oferecendo a possibilidade de migrar serviços do sistema a fim de fazer um balanceamento de carga entre processadores ou equipamentos fisicamente independentes[ZAN 93].

Apesar de desenvolver um conceito interessante e muitas facilidades, esse tipo de sistema operacional apresenta geralmente uma baixa eficiência [BEU 97]. Mais desvantagens começam a aparecer com a crescimento do sistema, pois fica cada vez mais trabalhoso gerenciar os objetos do sistema, deixando os objetos responsáveis pela comunicação com uma sobrecarga de mensagens.

2.4 Arquitetura baseada em Núcleo Reflexívo

Um sistema operacional que utiliza um núcleo reflexivo é aquele capaz de acessar sua própria descrição[ZAN 97], realizar uma análise sobre ela, e se necessário alterá-la com o objetivo de modificar uma funcionalidade já existente ou ainda adicionar uma totalmente nova, mesmo que esta tenha sido concebida após o projeto, implementação e inicialização do sistema.

A figura 2.5 apresenta de forma simplificada uma arquitetura baseada em um núcleo reflexivo. Pode ser identificada uma separação lógica de níveis de execução. No nível base ficam os objetos do sistema que implementam características funcionais e no meta-nível ficam os meta-objetos que implementam as características não funcionais da aplicação porém necessárias para o manutenção dos objetos.

Os objetos do nível base não precisam saber a existência dos meta-objetos, também não é obrigatório que cada objeto possua um meta-objeto associado a ele, porém cada objeto

deve estar inserido em um ou mais meta-espaços. Todas as chamadas aos objetos base são desviadas para o meta-nível para serem analisadas e se necessário serão executadas operações sobre elas, e somente depois disso as mensagens são devolvidas ao devido objeto no nível base. Mais informações sobre reflexão computacional são apresentadas na sessão 3.1.

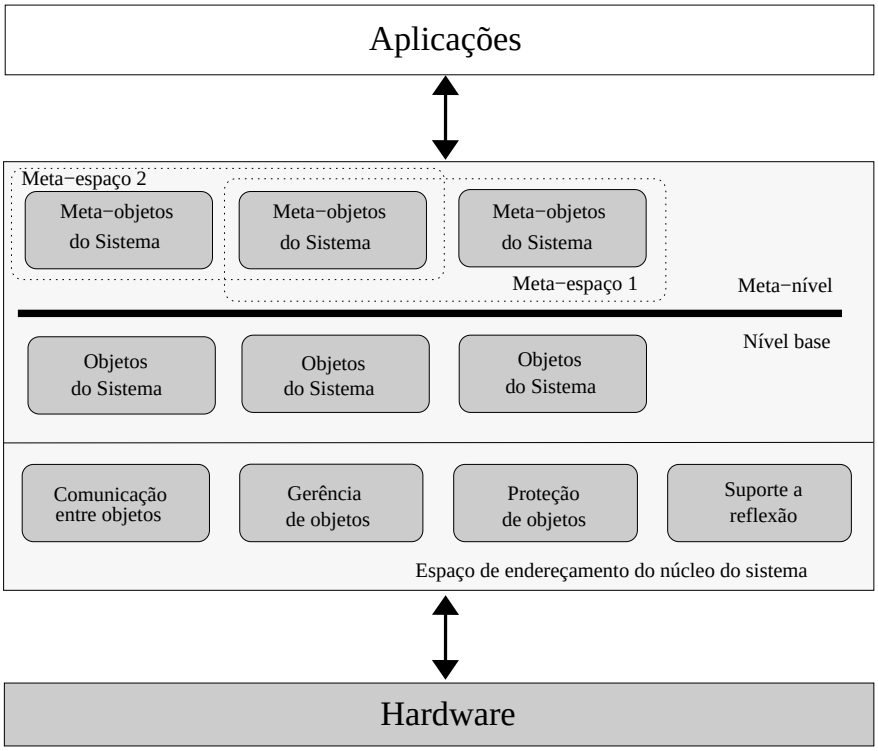


Figura 2.5: Arquitetura de um sistema operacional baseado em núcleo reflexivo

Uma característica nova desta arquitetura, é a disponibilização de meta-espaços. A função dos meta-espaços é de prover um ambiente de execução agrupando meta-objetos que disponibilizam suporte ao nível base[YOK 93]. Desta forma os objetos no nível base têm disponível somente as funções implementadas pelos meta-objetos do seu meta-espaço. Se houver necessidade de uma função implementada por outro meta-objeto pode-se tanto adicionar o meta-objeto ao meta-espaço do objeto, como adicionar o objeto ao meta-espaço onde encontra-se a funcionalidade.

Outro detalhe importante do sistema pode ser observado na figura 2.5 no que diz respeito aos serviços alocados no espaço de endereçamento do kernel. O número de serviços disponibilizados nesse espaço pode variar de implementação para implementação, mas basicamente é necessário disponibilizar o serviço de gerenciamento de objetos e o suporte a reflexão[ZAN 94] já que não estão nativamente implementados nos dispositivos físicos.

Na figura 2.5 as aplicações aparecem separadamente do sistema, porém é muito

comum que estas também tirem proveito da reflexão computacional, através da utilização deste paradigma obter o melhor resultado possível na sua execução.

As meta-informações armazenadas e manipuladas pelos meta-objetos podem dizer a respeito das propriedades do sistema operacional, sobre questões de desempenho e muitas outras. A certo nível, todos os sistemas operacionais são reflexivos[STA 98], entretanto a reflexão computacional nesta arquitetura esta sendo utilizada como principio central do sistema, permitindo uma maior flexibilidade para atender as necessidades específicas de cada aplicação.

Todas estas características que possibilitam a flexibilização do sistema operacional, proporcionando um ambiente modular e facilmente extensível tem um custo relacionando ao desempenho do sistema. Quando todas estas opções estão em execução ou disponíveis para serem executadas o sistema apresenta uma sobrecarga, devido ao grande número de objetos que precisa gerenciar, organizar, interceptar mensagens e todas as outras operações relacionadas aos objetos e meta-objetos. Desta forma existe uma perda de desempenho em relação a sistemas tradicionais.

2.5 Arquitetura baseada em Núcleo Extensível

Uma arquitetura baseada em núcleo extensível pode ser considerada como aquela que possui a habilidade ser modificada dinamicamente, com o intuito de oferecer os melhores recursos e funcionalidades para as aplicações que estão em execução em um determinado momento.

Existem divergências em relação ao nome dado a sistemas deste tipo. Alguns autores chamam de reconfiguráveis, outros de adaptáveis, e também os que classificam como extensíveis. De acordo com a taxonomia adotada para este trabalho, e apresentada na sessão 3.2, sistemas com estas características serão denominados extensíveis.

A idéia de núcleos extensíveis pode ser considerada como a evolução lógica dos microkernels[SEL 95], pois eles implementam um número mínimo de serviços, que na maioria das vezes servem apenas para dar o suporte necessário ao funcionamento dos recursos físicos e todas as outras funcionalidades são implementadas para uma utilização de forma dinâmica, ou seja, somente estarão presentes no sistema se forem requisitadas.

Uma das maiores diferenças entre um sistema operacional baseado na arquitetura de microkernel e um sistema operacional baseado na arquitetura de núcleo extensível é a possibilidade de adicionar código no espaço de endereçamento do núcleo no caso dos sistemas extensíveis. Os servidores de funcionalidades em sistemas baseados em microkernel até podem executar em modo protegido, mas seus códigos ficam armazenados no espaço de endereçamento

do usuário[BAC 98].

Como pode ser observado na figura 2.6 existem diversas abordagens adotadas por sistemas com núcleos extensíveis. Existem aqueles que permitem a alocação de códigos no espaço de endereçamento do usuário, no espaço de endereçamento do núcleo e ainda os que permitem que ambos sejam utilizados. Cada uma destas abordagens pode propiciar vantagens e desvantagens, uma avaliação mais completa pode ser encontrada na sessão 3.3

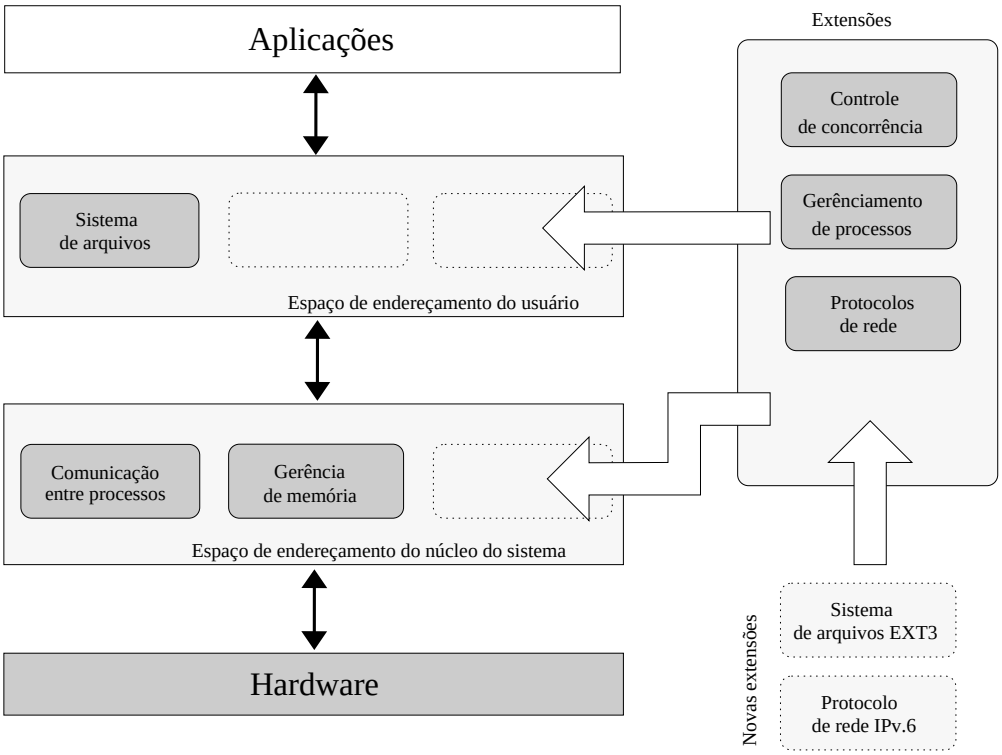


Figura 2.6: Arquitetura de um sistema operacional baseado em núcleo extensível

Além de permitir a adição no sistema operacional de serviços previamente implementados, os sistemas extensíveis permitem que códigos não previstos em tempo de projeto e desenvolvimento do sistema sejam adicionados e executados em tempo de execução, sem a necessidade de modificar e nem mesmo reinicializar o sistema.

A figura 2.6 demonstra esta possibilidade através da adição de “Novas extensões”. No caso do exemplo seriam a extensão “Sistemas de arquivos EXT3” e “Protocolo de rede IPv.6”, que hipoteticamente não existiam durante o desenvolvimento do sistema e serão carregados como extensões no espaço de endereçamento do usuário ou do núcleo do sistema.

Para possibilitar o desenvolvimento de extensões os sistemas operacionais que utilizam este tipo de arquitetura precisam ter uma relação dos seus serviços que podem ser externalizados bem definida. Outra informação importante seria uma completa definição das suas

interfaces para possibilitar o desenvolvimento de extensões por terceiros. Sem estes recursos disponíveis, a extensibilidade do sistema ficaria restrita somente aos projetistas originais[CLA 00].

Mais informações sobre sistemas extensíveis podem ser encontradas na sessão 3.3, bem como os principais sistemas existentes e o seu funcionamento.

Capítulo 3

Sistemas Operacionais Extensíveis

Sistemas operacionais extensíveis são aqueles que conseguem se adaptar a uma determinada circunstância com o objetivo de ganhar funcionalidades ou melhorar seu desempenho [COW 96].

3.1 Técnicas para prover extensibilidade

Esta sessão tem como objetivo apresentar uma breve descrição sobre os paradigmas de programação e as técnicas disponíveis atualmente que auxiliam no desenvolvimento de sistemas operacionais. Estes paradigmas e técnicas provêem facilidades como modularização do código, reuso, e especialmente para o nosso caso, a extensibilidade do sistema em tempo de execução.

3.1.1 O Paradigma de Orientação a objetos

Com a introdução de conceitos e abstrações que encapsulam dados e mecanismos naturalmente dentro do próprio modelo, o paradigma de linguagem de programação orientado a objetos mostrou-se eficiente na resolução de muitos dos problemas envolvidos no projeto de sistemas operacionais, tais como: identificação, proteção, atomicidade e sincronização.

Além disso, as linguagens de programação orientadas a objetos difundiram-se e tiveram suas funcionalidades e recursos bem documentadas, apresentando um alto grau de reuso de código, aumentando a portabilidade, facilitando as manutenções e permitindo a extensibilidade do código através de técnicas modernas de engenharia de software[BOO 04].

Outro aspecto favorável é modularização que possibilita a existência de siste-

mas operacionais orientados a objetos altamente customizáveis [FRO 01, ZAN 95b, HER 88]. Estes sistemas baseiam-se em *frameworks*, que facilitam a reusabilidade do código e customizações através do uso de herança. Este sistemas ainda permitem que as aplicações obtenham serviços especializados de acordo com as suas necessidades através da utilização de herança e polimorfismo[CAH 96].

E ainda, este paradigma permite que recursos do sistema e aplicações do usuário sejam modelados em termos da mesma abstração. Permitindo assim que o sistema operacional tenha o poder de manipular o comportamento transparente e dinâmico dos sistemas, pelo fato de que recursos, serviços e o próprio sistema podem ser modelados de forma abstrata [ZAN 97].

3.1.2 O Paradigma de Reflexão computacional

A reflexão computacional, de acordo com Steel[STE 94] é a capacidade de um sistema interromper o processo de execução, por algum motivo interno ou externo ao sistema, realizar computações no meta-nível e retornar ao nível de execução traduzindo o impacto das decisões, para então retomar o processo de execução.

Segundo Maes [MAE 87], a reflexão computacional no modelo de orientação a objetos acontece quando uma atividade executada por um sistema realiza computações sobre, e possivelmente afetando, suas próprias computações. No âmbito dos sistemas operacionais, a reflexão computacional pode ser considerada como a habilidade do sistema de realizar processamento sobre si mesmo e em particular de estender, em tempo de execução, a própria linguagem, impactando nas computações subseqüentes. Para alcançar essas propriedades, as linguagens reflexivas fazem uso de meta-classes. Cada objeto no sistema tem um meta-objeto que descreve a própria classe, através dos quais as aplicações ou o sistema podem alterar seu comportamento.

A figura 3.1 apresenta um sistema com dois objetos “A” e “B” no nível base e seus respectivos meta-objetos[1] no meta-nível. Quanto um deles tenta comunicar-se com o outro, a sua chamada é desviada e redirecionada ao seu meta-objeto, o qual se comunicará com o meta-objeto destino e este sim com o objeto propriamente dito. Se houver a necessidade de resposta nessa comunicação ela será realizada através do caminho inverso, ou seja, saindo do objeto origem no nível base, para o meta-objeto origem que repassará ao meta-objeto destino, no meta-nível e retornando ao objeto destino nível base.

[1]Meta-objetos podem ser considerados como objetos que definem, implementam, dão suporte ou participam de alguma maneira da execução da aplicação[FOO 93]

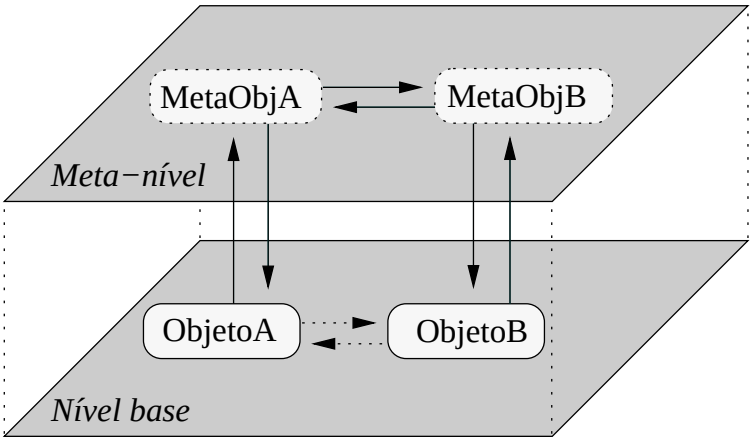


Figura 3.1: Nível base e meta-nível

A consolidação deste processo ocorre em três estágios distintos [MAE 87]:

- **A reificação**. É o primeiro estágio, o qual consiste em obter uma descrição abstrata do sistema tornando-a suficientemente concreta para permitir operações sobre ela.
- **A reflexão**. O segundo estágio, utiliza a descrição produzida no primeiro estágio com o objetivo de realizar alguma manipulação sobre, com ou através dela.
- **Modificação do sistema**. No terceiro e último estágio a descrição reificada será modificada conforme os resultados da reflexão computacional, retornando a nova descrição ao sistema. Desta forma, as operações subseqüentes refletirão as alterações efetuadas sobre a descrição reificada do sistema.

De acordo com Ferber[FER 89] a 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 serão realizadas as computações reflexivas.

Ainda no modelo de reflexão computacional temos o conceito de níveis de computação, levando em consideração que a ocorrência de um evento no processo de execução reflete-se em um nível superior de computação(meta-nível), e assim sucessivamente, gerando um hierarquia de níveis chamada torre de reflexão(Figura 3.2). O número de meta-níveis na torre de reflexão é teoricamente infinito, como pode ser observado nesta figura temos 3 níveis, no nível base, estão os objetos, e nos meta-níveis os meta-objetos.

Durante a execução de um objeto no nível N_i, se houver necessidade de utilizar a reflexão computacional, esse processamento ocorrerá em um nível superior N_{i+1}. Se um meta-objeto do meta-nível 1(N_{i+1}) reificar sobre ele mesmo, o processamento dessas informações será

um nível acima, no meta-nível 2(N_{i+2}), e assim sucessivamente. Porém existe uma divisão entre os meta-níveis e cada meta-objeto enxerga somente dois níveis, um acima N_{i+1} onde estão os meta-objetos e outro abaixo N_{i-1} onde estão os objetos, desta forma, um meta-objeto no meta-nível 2 reconheceria como objeto e não meta-objeto os ocupantes do meta-nível 1. A exceção é para o nível base N_0 que enxerga somente um nível acima(N_{i+1}).

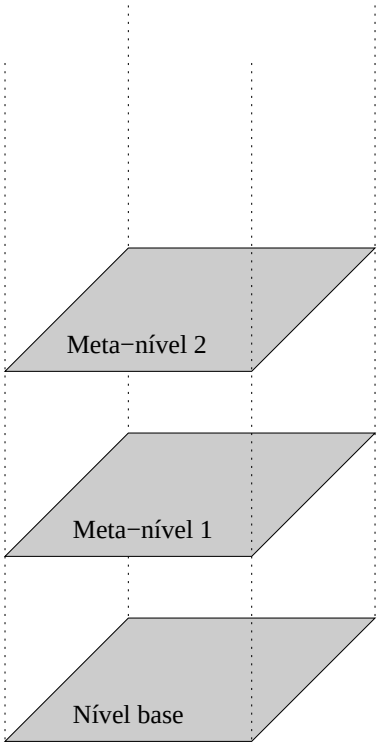


Figura 3.2: Torre de reflexão

Cada objeto no sistema representa uma parte da funcionalidade do ambiente e possui um meta-objeto associado que descreve seu comportamento. Podemos 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 [ZAN 95a].

A comunicação entre os objetos e meta-objetos e os meta-objetos entre si é realizada através de protocolos de comunicação conhecidos como MOP(*MetaObject Protocol*). De acordo com [ZIM 96] estes protocolos podem ser divididos em três categorias:

- **Protocolo Explícito**. Este protocolo permite a comunicação entre objetos do nível base e os meta-objetos no meta-nível.

- **Protocolo Implícito**. Protocolo responsável pela interceptação das mensagens no nível base e redirecionamento aos respectivos meta-objetos.

- **Protocolo Inter-meta-objetos**. A função deste protocolo é prover a comunicação entre os meta-objetos, não sendo visíveis aos objetos do nível base.

A reflexão computacional é utilizada para diminuir algumas limitações existentes no modelo clássico de orientação a objetos [SOL 95] como o espalhamento de determinados trechos de códigos pertencentes a diversas classes no sistema. Este paradigma permite ao desenvolvedor tratar de forma independente as funcionalidades ortogonais do sistema, produzindo um código mais modular incentivando a reutilização e facilitando as correções e atualizações de código, bem como a flexibilidade na utilização.

Um exemplo de funcionalidade ortogonal ao sistema é o controle de concorrência. Sem a utilização da reflexão computacional esses controles precisam estar presentes em várias partes do código e provavelmente em diferentes classes. Assim toda vez que for necessária alguma modificação ela será feita diretamente através da alteração das linhas de código espalhadas. O número de alterações será proporcional ao tamanho da aplicação, se esta for pequena não traria maiores dificuldades, mas em uma aplicação com, por exemplo, 300 mil linhas de código isso representaria um grande problema.

As principais vantagens do paradigma de reflexão computacional são[BOU 02]: a redução da complexidade – deixando a cargo do sistema operacional os requisitos não funcionais permitindo que o desenvolvedor se atenha à lógica do negócio – a separação conceitual – as aplicações ficam em um nível base e os serviços do sistema em um meta-nível, permitindo tanto a modificação de um como de outro independentemente – a reutilização evidenciada – devido a implementação de classes com objetivos bem definidos e de uso comum, e que de uma maneira geral podem ser utilizadas por vários objetos no sistema – e a capacidade de adaptação dinâmica – permitindo que o sistema seja modificado em tempo de execução sem afetar o funcionamento das aplicações.

O tempo em que será realizada a reflexão é variável, sendo que cada um deles apresenta vantagens e desvantagens. Para este trabalho vamos considerar apenas a reflexão em tempo de compilação e execução. A primeira oferece uma menor sobrecarga do sistema durante a execução e facilidade de implementação do compilador porém tem sua flexibilidade reduzida e maior tempo compilação, já a reflexão em tempo de execução possui uma grande flexibilidade, mas apresenta sobrecarga do sistema e grande dificuldade de implementação de um compilador e

um ambiente que suporte modificações durante a execução.

3.1.3 Separação de interesses

A idéia de separação de interesses(*separation of concerns*) foi introduzida por Dijkstra[DIJ 76] e baseia-se na divisão do domínio do sistema em partes menores que agrupam características comuns. Tratando isoladamente estas partes menores é possível resolver problemas mais complexos em um menor tempo proporcionando ainda uma maior facilidade nas adaptações e modificações do sistema [LAD 03].

Uma das melhores maneiras de se projetar um sistema é através da separação de suas responsabilidades em módulos distintos seguindo uma forma ordenada, permitindo assim, que seja possível a alteração e adaptação de cada uma delas sem que isto afete as características do sistema como um todo[SCH 01]. Quanto melhor realizada esta separação, maior será a eficiência no desenvolvimento, porém é necessário manter a clareza do código, para permitir o fácil entendimento, manutenção e evolução do mesmo.

As responsabilidades ou interesses podem ser considerados como uma determinada parte do domínio que será tratada, sendo que estes interesses podem ter requisitos funcionais ou não funcionais. Os requisitos funcionais dizem respeito às ações que o sistema deve realizar, ou seja, a lógica do negócio. Por exemplo, em um sistema bancário as ações seriam em relação a manipulação da conta, basicamente crédito e débito de valores(depósitos, cobrança de juros, crédito de investimentos, entre outros).

Os requisitos não funcionais do sistema – também conhecidos como funcionalidades ortogonais – dizem respeito à implementação de funções que não estão diretamente ligadas à lógica do negócio, mas sim a características que são consideradas necessárias para a execução da aplicação[KIC 05]. Voltando ao exemplo do sistema bancário, os requisitos não funcionais poderiam ser a segurança do sistema, controle de acessos, autenticação de usuários, persistência dos dados, verificação de erros, entre outros.

Quanto maior a complexidade do sistema, maior a probabilidade de existirem interesses comuns que deverão ser utilizados por vários módulos. Os requisitos não funcionais, como citado acima, geralmente estão incluídos neste grupo e apresentam um desafio aos desenvolvedores que devem isolá-los de uma forma que sejam facilmente acessados pelos módulos e que ainda mantenham a legibilidade do código[GUR 05].

No paradigma de programação orientada a objetos, as classes disponibilizam

uma boa solução para separar a maioria das responsabilidades funcionais, contudo, são bastante limitadas no isolamento de funcionalidades ortogonais ao sistema. Estas funcionalidades ficam espalhadas por vários módulos, muitas vezes em pequenos trechos de código, que podem se tornar repetitivos dificultando as revisões de segurança e atualizações. Neste sentido a técnica de programação orientada a aspectos(AOP)[KIC 97] complementa a orientação a objetos possibilitando uma divisão dos requisitos ortogonais.

A AOP busca a separação das funcionalidades ortogonais do sistema em aspectos, implementados separadamente do código da aplicação, e a sua posterior composição com as demais classes, dando origem a um sistema único[KIC 01]. Por definição os aspectos podem ser inseridos, alterados ou removidos em tempo de compilação. Devido ao fato de estarem em um conjunto de código separado, sua manutenção é mais simples, diminuindo a complexidade do sistema e facilitando o entendimento de uma forma geral. Outra vantagem dos aspectos é a possibilidade de um deles ser utilizado em diferentes locais e diferentes contextos, provendo assim uma forma transparente de disponibilizar requisitos não funcionais[ELR 01].

3.1.4 Máquinas virtuais e Java

Uma máquina virtual provê as funções e serviços básicos esperados por um programa computacional, porém estes são prestados através de uma camada de software e não diretamente pelo hardware. Para isto, é realizada uma multiplexação dos recursos, tais como memória, conjunto de instruções e também dos outros dispositivos da camada física, apresentando para a aplicação uma interface padrão de acesso a todos estes recursos[FIG 03].

Uma das principais funções de uma máquina virtual é separar o desenvolvimento de software da diversidade de dispositivos físicos existentes e/ou da grande variedade de sistemas operacionais[HAR 99], pois o mesmo código da aplicação poderá ser executado em qualquer sistema(dispositivos físico ou sistema operacional) que possua suporte a máquina virtual específica. Esta facilidade de execução permite uma grande mobilidade, inclusive de modificar o local físico onde o programa esta executando, possibilitando por exemplo, um balanceamento de cargas entre servidores, transferindo serviços dos que estão mais ocupados para os ociosos mesmo que estes sejam equipamentos distintos.

Além da mobilidade e portabilidade, podemos citar como outra vantagem o isolamento, pois os softwares que estão executando em uma máquina virtual não influenciam no funcionamento dos que estão executando na máquina real e vice-versa. Esta é uma técnica muito

utilizada para realização de testes, tanto de sistema operacionais como de programas pelo fato de não necessitar a reinicialização do equipamento a cada teste e no caso de falha, o processo simplesmente é abortado[ROS 04].

As máquinas virtuais também apresentam desvantagens. Estas desvantagens podem ser visualizadas exatamente em uma de suas principais características, a flexibilidade[CHE 01]. Provendo uma plataforma uniforme para um ambiente de programação a ser utilizado em diversos tipos de processadores podem ser encontradas dificuldades em acessar recursos especiais implementados somente por uma parte destes. Ainda é possível destacar que diferentes processadores implementam diferentes formas de realizar suas funções, que invariavelmente não serão iguais as da máquina virtual, necessitando assim uma conversão antes de serem executadas pelo dispositivo físico[BLU 02].

No restante desta sessão serão apresentadas as principais características da máquina virtual Java, a qual foi utilizada para a prototipação do modelo e serviu de base para os testes realizados. Começando com uma introdução, em seguida uma descrição do funcionamento dos carregadores(*Class loaders*), reflexão e por último o *proxy* dinâmico de objetos(*dynamic object proxy*).

A máquina virtual Java(*Java Virtual Machine JVM*) [GOS 96] serve como uma camada intermediária, que recebe programas(classes) em *Java bytecode* [GOS 95], e realiza operações através da comunicação com o sistema operacional. O *bytecode* pode servir para execução direta através do interpretador ou pode ser compilado em código nativo para determinada plataforma[LIN 96].

Carregadores de classes

Um dos recursos oferecidos pela JVM é a carga dinâmica de classes. Durante a inicialização e execução da máquina virtual, as classes são carregadas por demanda, ou seja, somente serão alocadas quando forem necessárias, diminuindo assim o consumo de memória e aumentando a velocidade de resposta do sistema. Para agilizar o processo de alocação não são realizadas verificações de tipos, pois quando o código fonte da classe foi transformado em *bytecode* ele já passou por estas operações[BOY 02].

Os carregadores de classes(*Class loaders*) são classes comuns no sistema(`java.lang.Classloader`), permitindo que o usuário modifique suas políticas de alocação e até mesmo utilize uma classe que implemente as características que este considerar necessárias. Estas propriedades devem ser modificadas através do uso de herança, ou seja, através da criação

de uma nova classe carregador que estende o funcionamento do carregador padrão do sistema, como demonstrado na figura3.3. O objetivo desta figura é somente ilustrativo, sendo assim, não é demonstrada nenhuma modificação do *bytecode* ou qualquer outra funcionalidade, contudo, ela apresenta a estrutura básica necessária para realização de tais operações.

```
1 class meuLoader extends ClassLoader {
2         public Class loadClass(String name) {
3                 byte[] bytecode = readClassFile(name);
4                 // operacoes com o bytecode
5                 return resolveClass(defineClass(bytecode));
6         }
7 }
```

Figura 3.3: Exemplo de como personalizar um carregador de classes

As cargas de classes no sistema seguem a seguinte seqüência[HAR 01]:

1. Quando realizada a solicitação de carga de uma classe, a JVM verifica se esta classe não existe no sistema, se existir simplesmente é retornado o apontador para ela.

2. Caso a classe requisitada ainda não existir no sistema, é realizada a solicitação ao carregador, da classe que originou a chamada, para carregá-la. Se houver alguma impossibilidade neste carregador de realizar esta operação a chamada será redirecionada ao objeto do qual ele foi originado(objeto pai).

3. Se nenhum carregador localizar a determinada classe, será gerada uma exceção (`java.lang.ClassNotFoundException`).

4. Se a classe for encontrada por algum dos carregadores, o seu *bytecode* será carregado e uma nova instância será criada a partir de `java.lang.Class`, e retornado um apontador.

Respeitando a seqüência pré-definida de operações, as funções implementadas pelo novo carregador realizarão a sobrecarga das funções originais, desta forma não existe a necessidade de implementar todas as funcionalidades, pois as demais podem ser as existentes no objeto pai(o carregador do sistema), provendo ainda mais flexibilidade ao usuário[OGE 05]. Durante a carga de uma classe, todas as classes referenciadas por ela também serão preparadas para

a carga, porém a máquina virtual somente carregará estas quando no momento da sua utilização, esta operação é denominada *lazy-loading*.

Outro recurso implementado pela JVM e utilizado pelos *Class loaders* é o múltiplo espaço de nomes(*multiple namespace*). Esta característica, permite que sejam carregadas várias classes com o mesmo nome dentro da mesma máquina virtual. Cada *class loader* identificará corretamente seu objeto e realizará as devidas operações com ele. Este recurso é muito útil no desenvolvimento de navegadores de Internet, onde possivelmente poderão aparecer classes com nomes iguais[HMD 01].

Uma das vantagens de desenvolver o próprio carregador é a possibilidade de adicionar funções extras a ele – a seqüência apresentada nesta sessão serve como base para o carregador, o que não impossibilita que outras operações sejam realizadas antes, depois ou intercaladamente – de modo a torná-lo mais seguro, completo, flexível, ou seja, mais adequado as necessidades específicas da aplicação que esta em execução.

Reflexão computacional

A máquina virtual Java, suporta reflexão estrutural desde a versão 1.1 do JDK, através da API `java.lang.reflect`. Com esta é possível instanciar e utilizar objetos dinamicamente. Esta API também fornece a capacidade de uma aplicação se auto-examinar ou examinar outras classes(reificação), resultado no conhecimento das suas propriedade e estrutura. Com esta facilidade é possível, por exemplo, obter o nome de todos os membros de uma classe, desde atributos, métodos em geral até os construtores[WU 98].

O processo de reificação pode acontecer com métodos como o `getClass()` que retorna o apontador para uma classe que posteriormente pode ser acessada, o método `getName()` que retorna o nome de uma determinada classe a partir de seu apontador, o método `getDeclaredMethods()` que retorna um vetor de métodos declarados em um determinada classe, o método `getField()` que retorna o conteúdo de uma determinada variável, o método `getConstructor()` que retorna os construtores da um classe, o método `invoke()` que permite a invocação de um determinado método de uma classe, entre outros.

Na figura 3.4 é apresentado um exemplo de como utilizar a reflexão em Java para obter todos os métodos de uma classe. A função utilizada foi a `getDeclaredMethods()` que, em tempo de execução, retornou um vetor com todos o métodos públicos justamente de uma das classes que provêem a reflexão, a `java.lang.reflect.Method`. Não será necessário utilizar o método `getName()`, pois será impresso na tela o caminho completo de onde esta classe está

```
1  import java.lang.reflect.*;
2
3  public class TesteReflexao {
4          public static void main(String args[]) {
5                  try {
6                          Class c = Class.forName( "java.lang.reflect.Method" );
7                          Method m[] = c.getDeclaredMethods();
8                          for (int i = 0; i < m.length; i++) {
9                                  System.out.println( m[i].toString() );
10                         }
11                 }
12                 catch (Throwable e) {
13                         System.err.println(e);
14                 }
15         }
16 }
```

Figura 3.4: Exemplo de como utilizar reflexão em Java

sendo carregada e/ou executada, bastando unicamente ser convertido para texto. Como o resultado é um vetor, foi implementado também um simples laço de repetição para mostrar o resultado ao usuário.

Como apresentado, a reflexão em Java pode facilitar a manipulação e utilização de classes em ambientes que necessitam de dinamicidade. Porém, esta reflexão restringe-se a forma estrutural, como carregamento de classes, modificações de campos, reificação sobre métodos, etc, mas não permite a reflexão comportamental que diz respeito à forma de como as classes realmente executarão e as opções que serão tomadas. É possível através de variáveis realizar uma certa tomada de decisão em tempo de execução, contudo de forma muito reduzida e pré-definida em tempo de projeto.

A reflexão comportamental é realizada através de um MOP (*MetaObject Protocol*) como já apresentado na sessão 3.1.2, que é responsável por todo o gerenciamento de troca de mensagens e comunicação entre objetos e meta-objetos, que não são muito bem definidos nas propriedades disponibilizadas atualmente pela API de reflexão do JDK. Para tentar contornar esse problema e oferecer a possibilidade de reflexão comportamental dinâmica sem adicionar extensões ao Java, foi utilizada uma das classes reflexivas(`java.lang.reflect.Proxy`) para prover

uma aproximação deste modelo. A qual será descrita na sessão seguinte.

Proxy dinâmico de objetos

A funcionalidade de *proxy* dinâmico de objetos foi incluída no pacote `java.lang.reflect` a partir da versão 1.3 do JDK. Através da criação de um objeto *proxy* dinâmico é possível realizar modificações indiretamente no comportamento de objetos em tempo de execução. Indiretamente por que não estarão sendo modificadas as características do objeto propriamente, porém, com o uso da reflexão será possível substituir funções, executar códigos externos, entre outras facilidades[OGE 05].

A interligação entre um objeto e o seu objeto *proxy* deve ser feita no momento em que o objeto é carregado sendo que vários objetos distintos podem compartilhar um único *proxy*. O objeto *proxy* funciona como um interceptador de chamadas, sendo que todos os métodos executados pelos objetos passarão pelo objeto *proxy* referenciado. Toda vez que uma chamada for interceptada, o objeto *proxy* pode realizar computações sobre ela, inclusive decidindo se esta será executada ou não. Além disso, devido ao com o controle total sobre a chamada, podem ser executados códigos previamente e posteriormente a requisição original, provendo desta forma uma alta flexibilidade para o sistema.

Como pode ser observado na figura 3.5, para criar um objeto *proxy* são utilizadas duas classes `java.lang.reflect.Proxy` e `java.lang.reflect.InvocationHandler`. A classe `Proxy` oferece o método utilizado para fazer a ligação de um objeto com o seu *proxy*, que é o `newProxyInstance()`. Este método recebe como entrada um carregador de classes, uma interface e uma instância da classe `java.lang.reflect.InvocationHandler` com o objeto base e retorna um objeto *proxy* dinâmico. A classe `InvocationHandler` possuí somente um método, o `invoke()` que é executado cada vez que houver uma chamada desviada ao *proxy*.

Um dos poucos requisitos para a instanciação e interconexão de objetos *proxy* dinâmicos, é que o objeto referenciado deve ter sido implementado através da utilização de uma interface[2]. Existem métodos para contornar esta necessidade [REN 00, PRA 04], permitindo que a ligação entre objeto e objeto *proxy* seja realizada unicamente a partir da classe do objeto. Contudo estas funcionalidades ainda não são oferecidas nativamente pela plataforma de programação Java.

[2]Não necessita do código fonte da classe, somente a classe e a sua interface.

```
1 import java.lang.reflect.Proxy;
2 import java.lang.reflect.InvocationHandler;
3
4 public static Object newInstance(Object obj) {
5         return java.lang.reflect.Proxy.newProxyInstance(
6                         obj.getClass().getClassLoader(),
7                         obj.getClass().getInterfaces(),
8                         new ProxyDinamico(obj) );
9 }
```

Figura 3.5: Exemplo de como criar um *proxy* dinâmico de objetos

3.2 Taxonomia em sistemas extensíveis

Devido às diversas denominações atribuídas as classificações de sistemas operacionais este trabalho seguirá a taxonomia apresentada nesta sessão. Esta taxonomia tem como base a apresentada por Seltzer em [SEL 97a].

3.2.1 Mutabilidade

Levando-se em consideração sistemas operacionais extensíveis, existem características que devem ser observados. Uma delas é em relação ao código base do sistema. Após a compilação, este código base será estático ou poderá ser estendido em tempo de execução. Estas características definem a mutabilidade do sistema. Os sistemas operacionais podem ser classificados em três categorias: parametrizado, reconfigurável e extensível. De acordo com Seltzer[SEL 97a] a melhor forma para descrevê-los é começar pelo estático, considerado menos flexível e gradativamente adicionando flexibilidade a ele.

As características de um SO **parametrizado**, também conhecido como sistema operacional especializado, são definidas em tempo de projeto. Sabendo-se que domínios específicos de aplicações e determinadas arquiteturas de hardware, muitas vezes tem necessidades bem determinadas e conhecidas é possível adaptar o sistema para satisfazê-las da melhor forma possível e uma vez compilado seu código, este não sofrerá mais nenhum tipo modificação[MON 94].

Adicionando-se adaptação dinâmica a um sistema parametrizado ele passa a ser denominado **reconfigurável**. Este sistema tem a capacidade de modificar seu comportamento

de uma forma limitada, pois todas as capacidades desejadas devem estar presentes no projeto e compilação do sistema, ficando a cargo dele escolher durante a execução qual ou quais alternativas provem maior vantagem. Como exemplo podemos citar o SO Solaris da Sun, que é desenvolvido e compilado com vários algoritmos de escalonamento e em tempo de execução as aplicações podem escolher qual deles é melhor para seu caso especifico[KHA 92]

Por último, os sistemas operacionais **extensíveis**. Eles podem, além da adaptação dinâmica, adicionar novas funcionalidades em tempo de execução. Mesmo após o desenvolvimento do projeto, compilação e inicialização do sistema existe a possibilidade deste sofrer modificações, adicionando-se novas características com o objetivo de prover um maior ganho de performance ou suportar novos recursos que não estavam presentes durante o desenvolvimento do projeto, modificando o seu comportamento e se necessário seu código também.

Na figura 3.6[3] podem ser observados, de acordo com a taxonomia adotada as diferentes características de adaptações em relação os períodos em que elas podem acontecer. No eixo “Y” estão representadas as características dos sistemas em relação as suas capacidades de adaptações. No eixo “X” estão representados os períodos em que as adaptações podem acontecer. Através da analise da figura, é possível chegar a conclusão de que quanto maior a distância do ponto de origem, maior a adaptabilidade do sistema em tempo de execução.

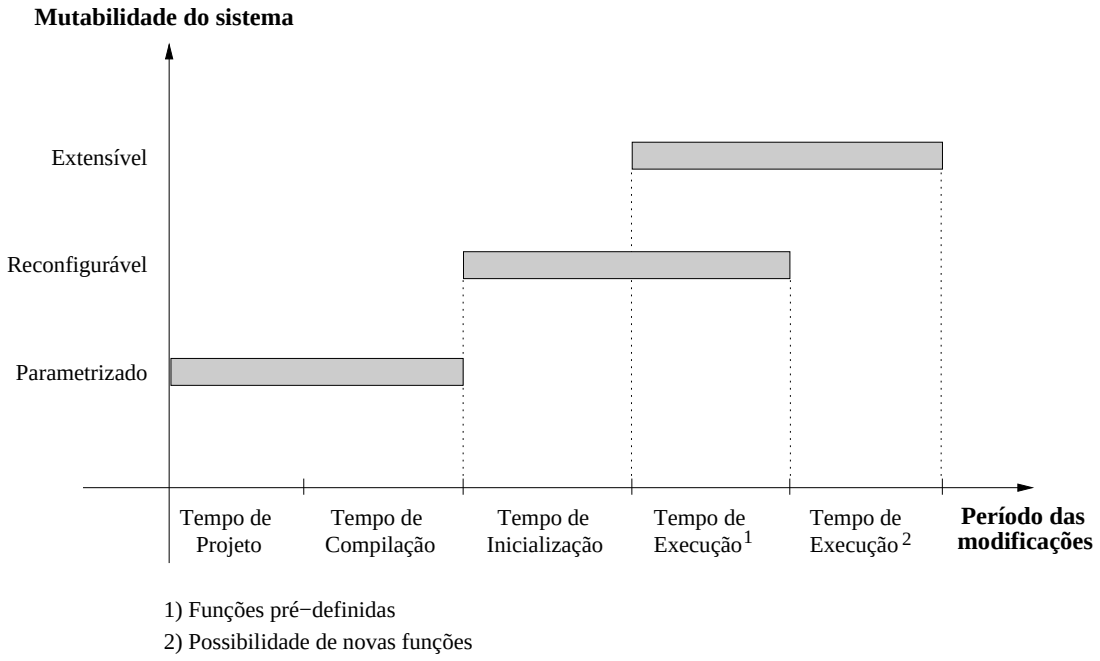


Figura 3.6: Classificação dos sistemas quanto a sua mutabilidade

[3]Figura adaptada de [MCK 04]

3.2.2 Localização

Em sistemas operacionais que permitem a adaptação do ambiente de execução de forma dinâmica, é preciso definir onde estas modificações podem ocorrer. Esta definição é conhecida como a localização das extensões. Estas podem estar a **nível de núcleo do sistema** ou a **nível de usuário**. Ambas abordagem apresentam vantagens e desvantagens. A alocação de extensões no espaço de endereçamento do usuário prove um alto grau de modularidade e flexibilidade, facilitado a sua modificação em tempo de execução, mas não oferece desempenho, que pode ser alcançado com extensões em nível de *kernel*.

Ainda, utilizando-se o espaço de endereçamento do usuário, é possível oferecer isolamento entre os componentes, não permitindo que um interfira diretamente no funcionamento de outro, porém as constantes trocas de contexto podem causar uma sobrecarga no sistema. O problema da sobrecarga pode ser resolvido adicionando as extensões no espaço de endereçamento do *kernel*, onde simplesmente será invocada a execução de uma sub-rotina dentro do mesmo espaço de endereçamento, não necessitando troca de contexto, contudo deixando o sistema vulnerável a execução de códigos arbitrários.

Existem técnicas para evitar a execução de código arbitrário no espaço de endereçamento do *kernel* ou ainda para garantir que este código seja seguro. Uma delas é a linguagem de programação Modula-3 [NEL 91], que possibilita a utilização de variáveis tipadas, fácil manuseio de exceções, programação orientada a objetos, concorrência e *garbage collection* automático, ainda provê acesso direto ao hardware, através de instruções especiais, e possui mecanismos de proteção de recursos para as demais instruções.

Outra técnica para evitar a execução de código arbitrário no espaço de endereçamento do núcleo do sistema, é o isolamento de falhas por software. O Sandboxing[WAH 93] realiza esta operação disponibilizando um conjunto reduzido de funcionalidades para as extensões, desta forma, limitando as possibilidade de ações, como escrita e leitura, no restante do núcleo do sistema.

Todas estas características não são mutuamente exclusivas, desta forma, podem ser combinadas com o objetivo de obter melhor desempenho e conseguir agregar o maior número possível de funcionalidades que necessitam estar presentes no sistema, o melhor é utilizá-las em conjunto aproveitando o que cada uma delas tem a oferece de melhor para determinado caso [DEN 02].

3.2.3 Confiabilidade da adaptação

A estabilidade de um sistema operacional é um dos pré-requisitos básicos em todos os projetos, levando em consideração que se um dos componentes falhar, por algum motivo qualquer, afete somente os componentes que dependem diretamente dele, não prejudicando o funcionamento do sistema como um todo [TAN 92].

Em um sistema operacional de comportamento estático, parametrizado por exemplo, ou ainda em um sistema reconfigurável que tem seu comportamento modificado em tempo de execução, mas com códigos definidos em tempo de projeto, é relativamente mais fácil prover estabilidade em relação a um sistema extensível, especialmente se as extensões do sistema serão alocadas no espaço de endereçamento do núcleo do sistema.

O termo confiabilidade, refere-se ao grau de liberdade, tanto de leitura como de escrita, que as extensões terão no sistema, levando-se em conta o quanto este sistema poderá ser prejudicado no caso de falhas ou erros de programação. Segundo[SEL 97a] existem três classes de proteção em um sistema: confiança, hardware e software.

No primeiro caso, não é implementada nenhuma proteção, pressupõem-se a boa vontade da aplicação, que neste caso pode adicionar da forma que preferir sua extensão e utilizá-las sem restrições. Essa abordagem é adotada, por exemplo, pelo MS-DOS[DEN 02] e no caso de falha de uma das extensões alocadas o sistema inteiro poderá falhar ou ficar com sua operação comprometida.

Com o objetivo de não comprometer o sistema inteiro, deve-se utilizar alguma técnica de proteção. Sistemas que utilizam barreiras de hardware têm uma diferenciação entre espaços de endereçamento, dessa forma o kernel do sistema será alocado em um espaço protegido e somente dele(espaço de endereçamento do kernel), onde é o único capaz de realizar leituras e/ou escrita nas estruturas de dados alocadas e os demais serviços ficam no espaço de endereçamento do usuário, acessível a todos.

Apesar de esta abordagem trazer a vantagem do isolamento dos componentes vitais do sistema ela pode apresentar sobrecarga na troca de mensagens, devido à necessidade das constantes troca de contexto realizadas pelo núcleo do sistema com o objetivo de contatar os serviços no espaço de endereçamento do usuário. Dependendo das operações realizadas pelo serviço, pode ser gasto mais tempo fazendo as chamadas de troca de contexto do que executando a operação em si[SMA 96].

Outra possibilidade é realizar proteção por software, existem várias formas de

implementar esses mecanismos, uma delas é através da utilização de linguagens seguras que impossibilitam o acesso, tanto de leitura quanto de escrita, fora dos limites da aplicação, ou seja, o programa só vai ter acesso aos endereço de memória alocados para ele, entre outras técnicas já comentadas anteriormente.

3.2.4 Tempo de vida das extensões

O período que uma determinada extensão permanecerá ativa no sistema depende da funcionalidade para qual ela foi desenvolvida. Extensões que agregam recursos a uma única aplicação, devem permanecer no sistema somente durante a execução desta. Se o recurso disponibilizado for de interesse de várias aplicações e/ou componentes do sistema, esta deverá permanecer enquanto ela for necessária.

Um das formas de classificar o tempo de vida das extensões é levar em consideração o objetivo para qual ela foi projetada. Desta forma, uma extensão que permanece no sistema durante a execução de uma aplicação tem um **tempo de vida de aplicação**. Extensões que permanecem no sistema pelo período que um recurso persiste possuem um **tempo de vida de recurso**. **Tempo de vida de kernel**, significa que a extensão permanecerá durante a execução do kernel, e para modificá-la ou retira-la será necessária uma reinicialização do kernel. **Tempo de vida permanente** significa que esta extensão permanecerá até que seja explicitamente removida[SEL 97a].

3.2.5 Granularidade

A granularidade das extensões do sistema é definida pelo tipo de adaptações que estas estão autorizadas a realizar. No caso de um sistema operacional que aceita a alteração, em tempo de execução, somente de módulos inteiros, podemos dizer que este apresenta uma granularidade grossa sendo considerado como de **extensibilidade modular** . Pois o módulo, como um recurso único deve ser substituído para dar espaço as novas alterações no sistema. Esta um dos tipos de extensibilidade mais implementados devido à relativa facilidade de implementação em relação as demais [SEL 97a].

Outra vantagem desta abordagem é a facilidade de gerenciamento destas extensões, uma simples lista de extensões é o suficiente para obter-se um estado do sistema. Podemos ainda citar a estabilidade do sistema. Devido as características citas anteriormente, estas extensões costuma ser alocadas no espaço de endereçamento do usuário, provendo assim uma segurança maior, pois em uma falha estas não afetarão os serviços que estão sendo executados no núcleo do

sistema.

Uma das desvantagens deste tipo de extensão é a flexibilidade reduzida, pois as adaptações devem ser realizadas em módulos inteiros, não deixando a opção de adaptação de apenas uma parte de um serviço, que acaba levando também a uma maior ocupação da memória. Por exemplo, se duas extensões implementam recursos distintos porém ambos interessantes para uma aplicação as duas devem ser totalmente carregadas na memória, mesmo que o recurso implementado seja apenas um de vários oferecidos por cada uma delas[DEN 02].

Com o objetivo de prover maior flexibilidade e controle do sistema, existem extensões que podem ser apenas parcialmente alocadas, utilizando-se somente o que for necessário para a aplicação. Estas extensões são caracterizadas como de granularidade fina e o sistema pode ser considerado como de **extensibilidade procedural**. Estas extensões geralmente são implementadas através da adição ou modificação de códigos dentro do espaço de endereçamento do núcleo do sistema, tendo uma execução mais rápida. As principais desvantagens deste tipo de extensões são as dificuldades de gerenciamento do sistema e a falta de segurança que estas podem trazer executando códigos arbitrários dentro do espaço de endereçamento do núcleo.

Existem ainda os sistemas que são caracterizados por ter uma granularidade média e serem considerados com de **extensibilidade procedural limitada**. Geralmente estes sistemas apresentam um controle de tolerância a falhas através de software reduzindo assim as possibilidades de atuações das extensões, não permitindo que o sistema alcance a flexibilidade oferecida pelos sistemas de granularidade fina, porém provendo maior segurança que estes[SEL 97a].

3.2.6 Agente de extensibilidade

O agente de extensibilidade refere-se ao responsável pelo início da adaptação do sistema, ou seja, o fator que produziu a informação alertando o sistema que ele deveria adaptar-se a uma determinada situação, provavelmente também indicada pelo próprio agente. De uma forma geral, existem três tipos de agentes de extensibilidade [DEN 02], que são: o humano (usuário/administrador), a aplicação e o próprio sistema.

O agente de extensibilidade **humano** pode agir durante a inicialização do sistema através da informação das características que ele deseja que estejam presentes no sistema operacional. Esta operação é muito comum em sistemas parametrizados e reconfiguráveis, mas pode acontecer em sistemas extensíveis também. Além de prover adaptações durante a inicialização do sistema, o administrador/usuário pode realizar estas operações em tempo de execução,

indicando ao SO quais módulos ou serviços devem ser carregados ou substituídos. Uma das vantagens de realizar este procedimento de forma dinâmica é que podem ser percebidas as modificações no momento em que elas estão ocorrendo provendo um retorno imediato das ações.

Outra forma de iniciar o processo de adaptação do sistema é através da ação direta das **aplicações**. As aplicações são programas em execução no sistema operacional, desta forma, elas só podem iniciar um processo de modificação dinâmica. Partindo do pressuposto de que as aplicações conhecem suas necessidades e a forma de interação com o sistema operacional, elas podem sugerir modificações que tragam vantagens para sua execução. Geralmente operações como estas podem ocasionar uma sobrecarga ao sistema, mas de uma forma geral, as vantagens alcançadas compensam[DEN 02].

Quando o agente de extensibilidade é o próprio **sistema**, acontece uma adaptação dinâmica e automática dos serviços oferecidos, objetivando prover o melhor ambiente de execução para as aplicações. Para alcançar uma adaptação deste tipo, o sistema operacional deve ser capaz de realizar uma análise das suas funcionalidades comparando-as com as funcionalidades requeridas pelas aplicações e tomar uma decisão de qual atenderá da melhor maneira possível estes requisitos. Sistemas operacionais de uso geral que apresentem estas características podem ser considerados com o último passo na evolução desta categoria[DEN 02].

3.3 Sistemas existentes

Existe um número considerável de projetos e grupos de pesquisa desenvolvendo sistemas operacionais extensíveis, alguns destes serão descritos nesta sessão. O objetivo primordial é apresentar algumas das suas principais características que possibilitam a adaptação do sistema de forma dinâmica buscando atender da melhor forma possível os requisitos das aplicações.

3.3.1 SPIN

O SPIN[BER 94] é um sistema operacional que utiliza serviços orientados a aplicações, ou seja, que é capaz de satisfazer precisamente as necessidades de performance e funcionalidades requeridas por uma determinada aplicação ou tipos de aplicações.

Esse tipo de serviço é implementado através da utilização de núcleos dinâmicos que tem a capacidade de disponibilizar recursos que podem ser gerenciados de forma eficiente(com maior rapidez de execução e menor complexidade computacional) e segurança(aplicações execu-

tam protegidas por barreiras tanto em nível de hardware quanto em nível de software) pelas aplicações.

Os serviços do sistema são divididos em três unidades principais:

- Spindles(*SPIN Dynamically Linked Extensions*) são seqüências de código carregadas dinamicamente no espaço de endereçamento do núcleo do sistema.
- Bibliotecas que estão no espaço de endereçamento das aplicações podem interagir com as aplicações sem a necessidade de trocas de contexto ou contatar os spindles do *kernel* via *system calls*.
- Servidores a nível de usuário, mantêm o estado geral sobre o serviço extensível.

A figura 3.7 apresenta uma forma simplificada da arquitetura do sistema operacional SPIN. É possível perceber que existem espaços reservados dentro do espaço de endereçamento do núcleo do sistema para receber extensões chamadas de *spindles*. Outra forma de adicionar recursos ao sistema é através da inclusão de bibliotecas no espaço de endereçamento do usuário.

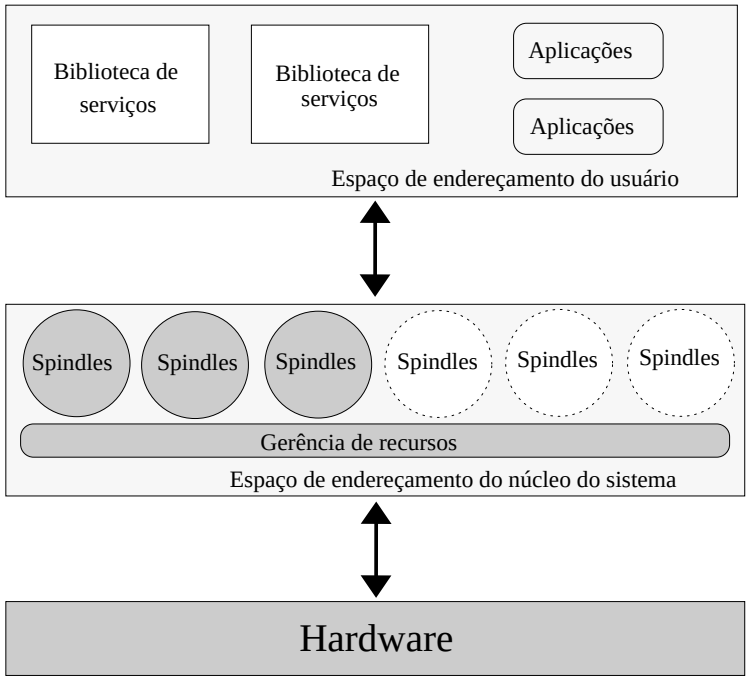


Figura 3.7: Representação do sistema operacional SPIN

Todo o sistema é estruturado na arquitetura extensível do microkernel, que exporta interfaces que oferecem controle detalhado para as aplicações sobre abstrações fundamentais do sistema, como processador, memória e I/O. Os *spindles* podem responder a eventos tanto de

hardware como de software, como exceções e trocas de contexto, produzindo a extensibilidade e customizando as interfaces do kernel especificamente de acordo com as necessidades de determinada aplicação[BER 95].

Existem duas formas básicas de utilização dos *spindles*. Na maneira mais simples de utilização ele pode servir como se fosse a implementação tradicional de um chamada de sistema(*system call*). Em uma forma mais sofisticada ele pode ser habilitado por uma aplicação para monitorar e reagir a mudanças nos recursos globais, tendo em vista que seu código compilado esta dentro do núcleo do sistema, ele pode executar evitando o custo da troca de contexto.

Na tentativa de aumentar a segurança, os *spindles* precisam ser escritos numa linguagem segura(Modula-3[NEL 91]) e compilada pelo kernel(ou por um servidor confiável), onde é realizada um verificação estática de segurança. Em tempo de execução ainda são verificados dos tipos e valores dinâmicos.

3.3.2 Exokernel

Com o intuito de prover o controle dos recursos da máquina para as aplicações, um Exokernel define uma interface de baixo nível. A arquitetura é baseada em uma simples observação: uma primitiva de mais baixo nível pode ser implementada mais eficientemente e permitir mais liberdade para os desenvolvedores das abstrações de mais alto nível[ENG 98].

A implementação do Exokernel é realizada através de bibliotecas. O maior desafio é oferecer a essas bibliotecas do sistema operacional máxima liberdade no gerenciamento dos recursos físicos enquanto os protege de cada um dos outros; um erro de programação em uma biblioteca pode afetar outra biblioteca do sistema operacional. Para alcançar esse objetivo, o Exokernel separa proteção de gerenciamento através de interfaces de baixo nível.

É possível observar na figura 3.8 a forma simplificada do sistema operacional Exokernel. A função propriamente do Exokernel é multiplexar os recursos físicos de forma segura e toda a implementação do restante do sistema deve ser realizada através das bibliotecas do sistema. Desta forma, como representado na figura 3.8 poderíamos ter três sistemas operacionais distintos executando sobre o mesmo *hardware* multiplexado pelo Exokernel.

De acordo com Engler[ENG 95], existem três razões básicas para permitir que uma aplicação acesse recursos de baixo nível. Uma das razões seria o acesso da aplicação a recursos avançados do hardware sem a necessidade de atualizar o kernel. Outra razão seria a possibilidade da adaptação das políticas do sistema operacional a suas características, podendo

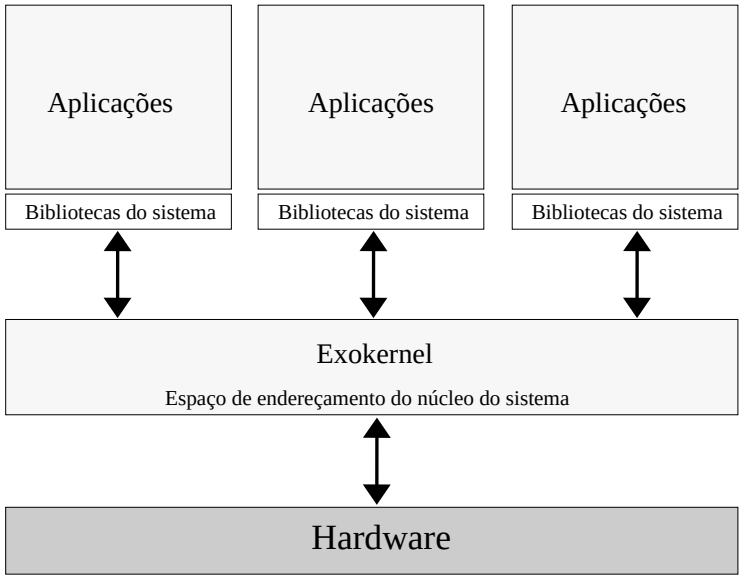


Figura 3.8: Representação do sistema operacional Exokernel

assim alcançar uma melhor performance. E por último a especialização do sistema operacional através da utilização de políticas de gerenciamento específicas para cada tipo de aplicação.

Existem duas maneiras de estender as funcionalidades do Exokernel. Uma delas é através da modificação das bibliotecas a nível de usuários, que representam abstrações do núcleo do sistema. Devido ao fato destas bibliotecas não estarem diretamente ligadas a recursos físicos estas podem ser alteradas de qualquer maneira e até assumir estados arbitrários, pois não implicarão no mau funcionamento do sistema como um todo.

Além da extensão através da modificação de bibliotecas no espaço de endereçamento de usuários, é possível estender as funcionalidades do Exokernel através de download de código para dentro do espaço de endereçamento do núcleo do sistema. Esta segunda opção é bem mais limitada, pois uma falha nesse estágio poderia comprometer a integridade do sistema inteiro. Assim os códigos candidatos a download precisam ser escritos em uma linguagem segura ou utilizando isolação de falhas através de software.

3.3.3 Choices e μChoices

O Choices[CAM 93] é um sistema operacional orientado a objetos, com um kernel monolítico e um abrangente framework, através do qual pode ser realizada a especialização do sistema para uma determinada aplicação ou um determinado conjunto de aplicações. Além disso, o framework prove maior facilidade para reutilização do código e rápida prototipação de um novo

sistema.

Este sistema pode ser considerado parametrizado e extensível. Parametrizado porque como já foi visto anteriormente, oferece a capacidade de alterar sua configuração em tempo de projeto através do framework. Extensível porque possui agentes de download de software que permitem a adição de códigos no espaço de endereçamento do núcleo do sistema.

Com o intuito de aumentar a portabilidade do código o Choices passou por um processo de reformulação, e o núcleo monolítico do seu sistema passou a ser implementado na forma de microkernel, recebendo o nome de μChoices[CAM 95]. Além do microkernel, foi desenvolvido também um nano-kernel, responsável pela abstração dos recursos físicos da maquina. O nano-kernel prove uma interface bem definida para ser utilizada pelo microkernel, que implementa as abstrações do sistema como a paginação, memória virtual, controle de concorrência, entre outros.

A extensibilidade tanto do Choices como do μChoices é alcançada através da alocação de código no espaço de endereçamento do núcleo do sistema. Com o intuito de aumentar a segurança das extensões, o código não é diretamente executando, mas sim interpretado por uma estrutura semelhante a uma máquina virtual, provendo mais controle do sistema operacional sobre as adaptações.

Após todas as modificações no projeto e na sua forma de organização, o novo μChoices mantêm sua flexibilidade e suas características originais, podendo ser parametrizado em tempo de projeto, através do seu completo framework ou estendido em tempo de execução, através de agentes de download de código[CAM 96].

3.3.4 VINO

O sistema operacional extensível VINO[SEL 94] permite modificações em tempo de execução através da inserção de *grafts* no código do núcleo do sistema. Estes *grafts* são trechos de códigos escritos em C ou C++ lincados dinamicamente. Por questões de segurança o compilador introduz isolamento de falhas através de software e adiciona uma assinatura única a cada uma das extensões, que somente será carregada após a conferência da mesma.

Os *grafts* são lincados no espaço de endereçamento núcleo do sistema através de *grafts points*, como pode ser observado na figura 3.9. Sendo assim existem locais específicos onde podem ser adicionadas extensões. Existem duas formas realizar esta adição, uma delas é através da substituição do método existente no *grafts point* por outro método a escolha do usuário. A outra

forma é através da adição gerenciadores de eventos, que podem desviar o fluxo de execução do sistema para outra região de memória onde encontram-se as funções estendidas.

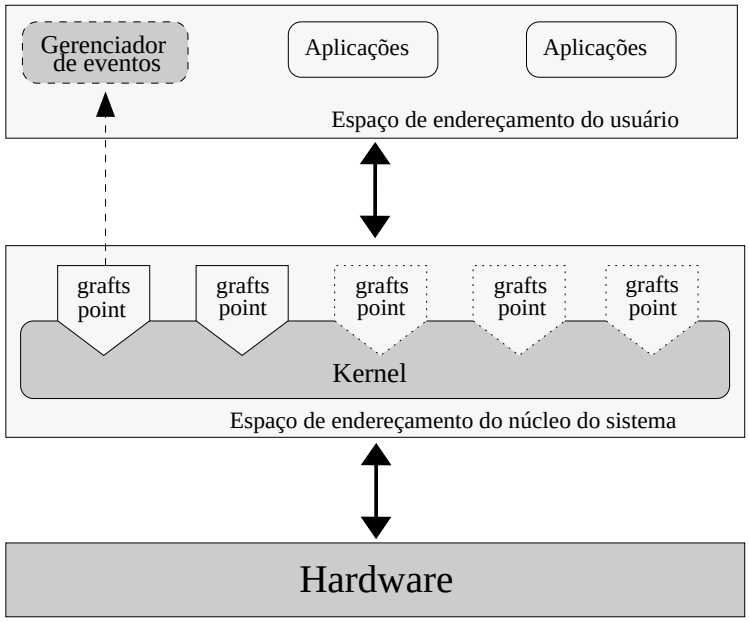


Figura 3.9: Representação do sistema operacional VINO

Na arquitetura deste sistema operacional, as extensões, geralmente, são validas somente para a aplicação que a ativou[SEL 97b]. Por exemplo, se uma aplicação desejar que a sua forma de persistência mude de um armazenamento local das informações em disco rígido para um servidor de dados conectado a Internet, somente ela será afetada por essa modificação no comportamento, permanecendo o restante do sistema com armazenamento local. Porém políticas gerais também podem ser modificadas e/ou estendidas, como no caso da adição e ativação de um novo algoritmo de escalonamento, desta forma, todas as aplicações no sistema seriam afetadas, modificações dessa natureza somente são permitidas a determinados usuários do sistema.

Todas as operações realizadas por extinções no VINO são através de transações monitoradas. Sendo assim, desde o momento da adição de uma extinção até o final da sua execução ela é monitorada e se houver a necessidade de abortar o processo em execução ou se o mesmo chegar a um ponto de inconsistência, é possível desfazer todas as suas operações e retornar ao último estado estável do sistema, continuando a partir deste ponto[SMA 98].

3.3.5 Apertos

O sistema operacional Apertos[YOK 92] é baseado no paradigma de reflexão computacional e foi o responsável pela difusão de conceitos como meta-objetos definido por Maes[MAE 87] e a introdução de novos conceitos como meta-espaços, dos quais ele é constituído.

Uma das principais características dos meta-objetos no Apertos é implementar aspectos não funcionais as aplicações, como persistência, gerenciamento de mensagens, escalonamento, etc. Esta separação conceitual permite que desenvolvedores atenham-se as funcionalidades das aplicações, sem se preocupar com as funcionalidades do sistema.

Uma inovação proposta pelo sistema operacional Apertos foi o conceito de meta-espaços. Os meta-espaços podem ser definidos como o conjunto de meta-objetos que tem por função básica oferecer as características necessárias à execução de determinada aplicação, permitindo desta forma, um atendimento personalizado para cada caso.

Em tempo de execução, uma aplicação é suportada por um dos vários meta-espaços, sendo que cada meta-espaço contém no mínimo um meta-objeto, chamado de *reflector*[YOK 92]. Um meta-objeto *reflector* age como um *gateway*, interceptando requisições que chegam ao meta-espaço e redirecionando-as ao meta-objeto apropriado. Por exemplo, uma requisição de escalonamento, será redirecionando pelo *reflector* ao meta-objeto escalonador.

A reflexão computacional prove vários níveis de reflexão, onde os objetos do nível base(N0) estão associados a meta-objetos do nível superior(N1). Os meta-objetos do nível superior(N1), por sua vez, podem estar associados a um nível superior também(N2), sendo ao mesmo tempo objetos aos meta-objetos do nível N2 e meta-objetos aos objetos no nível base(N0). A partir deste conceito, podemos tender ao infinito, para evitar estas situações, no Apertos estão definidos somente três níveis(Nível base, meta-nível e meta-meta-nível).

Os serviços básicos para todos os objetos são oferecidos por um estrutura chamada *MetaCore*. O *MetaCore* pode ser levemente associado ao conceito de microkernel[ZAN 97] e oferece serviços como identificação de objetos permitindo que estes sejam localizados posteriormente, instanciação e destruição de objetos do ambiente. Devido a arquitetura do sistema, estes objetos podem ser aplicações ou serviços e serão excluídos da mesma forma.

3.3.6 MetaOS

O sistema operacional MetaOS[HOR 97] foi desenvolvido para dar continuidade nas idéias apresentadas pelo Apertos. Ele continua utilizando a reflexão computacional como

base de desenvolvimento e organização do sistema. Com o objetivo de prover maior desempenho, a torre de reflexão do MetaOS foi reduzida para somente três níveis, de acordo com os seus desenvolvedores[HOR 99], uma torre de reflexão infinita não acrescentava muitas facilidades ao sistema e proporcionava uma sobrecarga que era considerada como desnecessária.

O motivo principal da redução da torre de reflexão foi devido ao fato de que a estruturação de um sistema em meta-níveis pode ser comparada a estrutura de um microkernel, onde cada chamada realizada pelo sistema precisa executar uma troca de contexto para passar de um nível para outro[HäR 97], consumindo recursos computacionais para isto. No caso do microkernel existem somente dois níveis, porém a reflexão pode oferecer uma estrutura de níveis infinita aumentando consideravelmente as necessidades de troca de contexto e proporcionalmente a necessidade de utilização de recursos para isso.

Depois da redução, os três níveis existentes no sistema são: o nível base, o meta-nível e o meta-meta-nível. O nível base é onde ficam alocados os objetos das aplicações em geral, uma espécie de espaço de endereçamento do usuário. No meta-nível é onde ficam os meta-objetos relacionados as aplicações que oferecem serviços e recursos, também é onde a reflexão acontece sendo que este é necessário para a customização do ambiente. O último nível, o meta-meta-nível é responsável por fornecer a possibilidade de reflexão, comunicação com os recursos físicos e gerenciamento dos meta-espaços[HOR 98].

Assim como o Apertos, o MetaOS também é composto por meta-espaços. Dentro de cada meta-espaço ficam alocados os meta-objetos que pode ser substituídos dinamicamente. Cada meta-espaço é composto por determinados meta-objetos que implementam serviços básicos, que são: o meta-objeto refletor, o correio, o nomeador, o escalonador e o gerenciador do meta-espaço[HOR 97]. Cada um destes meta-objetos tem funcionalidades bem definidas, por exemplo, o refletor é responsável por interceptar todas as mensagens que ocorrem no nível base referente ao seu meta-espaço, repassando-a ao meta-objeto destino.

O meta-objeto correio é responsável pelo gerenciamento(recebimento e entrega) de mensagens entre meta-espaços distintos. A função do escalonador é escalonador os objetos e fluxos de execução de acordo com uma determinada política específica para o meta-espaço em que ele esta alocado. Finalmente, o meta-objeto gerenciador do meta-espaço, que deve realizar o controle de autenticação e permissões dos demais objetos e meta-objetos, garantindo assim a segurança do sistema. Como estes meta-objetos possuem funcionalidades básicas para cada meta-espaço, não podem ser substituídos, mas suas políticas podem ser adaptadas dinamicamente.

3.3.7 Aurora

O sistema operacional Aurora[ZAN 97], introduziu um novo conceito de modelagem de sistemas operacionais, aplicando as idéias utilizadas pelo Apertos, em ambientes distribuídos e/ou multiprocessados. De acordo com Zancanella[ZAN 95b], o Aurora é baseado no modelo de objetos, os quais no mundo real são naturalmente concorrentes e distribuídos, então a forma mais natural de implementação de um sistema seria através do gerenciamento de um conjunto de mensagens que pudessem fluir entre os objetos executando de forma paralela.

Além da utilização do modelo de reflexão computacional e orientação a objetos para a implementação do sistema, estes conceitos funcionam como entidades fundamentais para alcançar a uniformidade do ambiente. Sendo assim, objetos e meta-objetos são as únicas estruturas existentes, em qualquer nível de execução, desde aplicações dos usuários, serviços essenciais do sistema operacional até os recursos do mesmo[ZAN 94].

Dentro da arquitetura do Aurora, os objetos mantêm uma representação de estados, ou seja, são designados para modelar e armazenar abstrações de dados ou informações referentes a sua aplicação. Já os meta-objetos, são responsáveis por armazenar as funcionalidades de cada objeto, ou seja, o seu comportamento. Desta forma, cada objeto no sistema armazenará informações e terá um meta-objeto associado a ele que contém as funções. Os meta-objetos podem ser agrupados em conjuntos, formando meta-espaços, que estão relacionados a um objeto no nível base, desta forma, este objeto pode utilizar vários meta-objetos que implementam diferentes funções. A extensibilidade do sistema pode ser alcançada adicionando-se novos meta-objetos que não estavam previstos em tempo de projeto.

As características de sistema distribuído e/ou multiprocessado são satisfeitas através da migração de objetos e meta-objetos, sejam elas serviços do sistema ou aplicações, permitindo a distribuição da carga entre os processadores, sendo que todos os objetos são tratados de maneira uniforme e independentemente da localização, provendo total transparência para as aplicações.

A figura 3.10 ilustra o modelo conceitual da arquitetura do Aurora. Nesta imagem, podemos perceber a existência de um `MetaCore`, que tem como única funcionalidade prover as características de reflexão necessárias a todos os objetos e meta-objetos, sejam eles aplicações ou serviços do sistema. Ele é implemento com duas funções básicas, uma que transfere a execução para o meta-nível e outra que devolve a execução para o nível base.

A primeira camada que pode ser observada na figura 3.10 – Suporte ao modelo

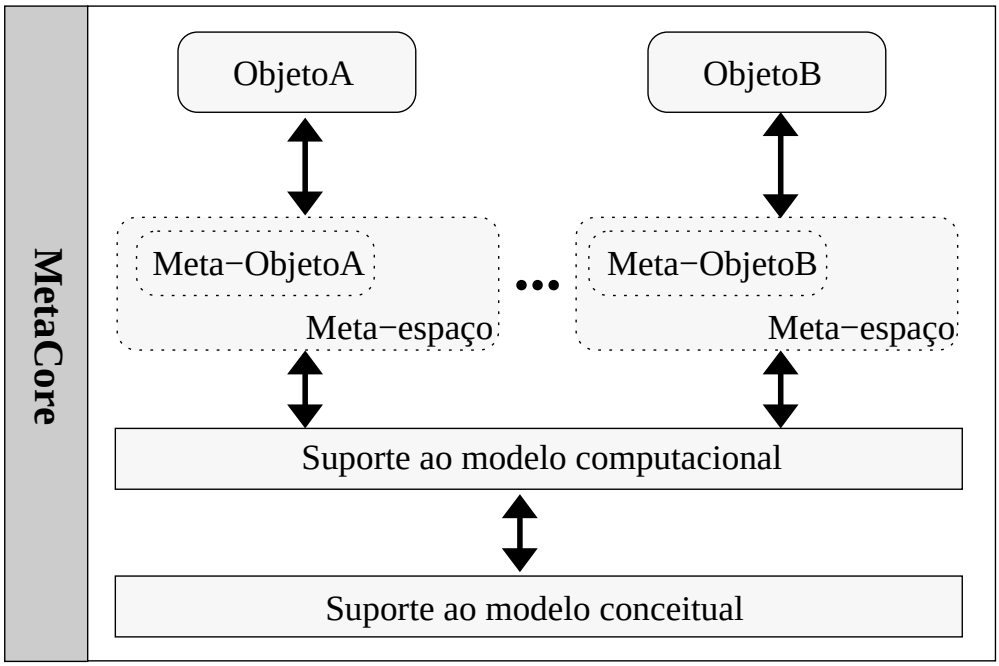


Figura 3.10: Representação do sistema operacional Aurora

conceitual – tem como finalidade oferecer os serviços de gerenciamento de objetos, meta-objetos e meta-espaços. A camada seguinte – Suporte ao modelo computacional – é responsável pelo controle de concorrência, prover a migração de objetos e a troca de mensagens entre eles. Logo acima destas duas camadas vêm as aplicações, sendo que cada objeto pode estar relacionado a um ou mais meta-espaços, dependo das funcionalidades de execução de que necessitará.

3.3.8 Comparação dos sistemas

Como apresentado nesta sessão, existem diversas formas dos sistemas operacionais proverem extensibilidade. Estas vão desde a implementação de pseudo-maquinas virtuais a nível de núcleo do sistema para interpretação segura de extensões, até a utilização de reflexão computacional que provê um ambiente onde é possível obter o estado do sistema em execução e realizar operações dinamicamente sobre ele, mudando tanto a sua estrutura como o seu comportamento.

Ainda não é possível indicar um modelo ótimo. Devido ao fato de que cada um dos modelos apresentados oferece vantagens e desvantagens, que podem se tornar mais atrativas em determinadas situações e não aconselháveis em outras, tudo depende das necessidades específicas das aplicações que estão em execução em um determinado momento.

Algumas das diferentes técnicas adotadas estão resumidas na tabela3.1[4] rela-

[4]Tabela adaptada de [SEL 97a]

cionadas as seus respectivos sistemas operacionais, e organizadas de acordo com a taxonomia utilizada no modelo proposto e apresentada na sessão 3.2.

Sistema	Localização	Confiabilidade	Tempo de vida	Granularidade
SPIN	núcleo	software	kernel	procedural limitada
Exokernel	usuário	hardware	permanente	procedural
	núcleo	software	aplicação	procedural limitada
Choices	núcleo	software	kernel	procedural
μ**Choices**	núcleo	software	kernel	modular
VINO	núcleo	software	recurso	procedural limitada
	usuário	hardware	recurso	procedural
Apertos	núcleo	software	aplicação	procedural
MetaOS	núcleo	software	aplicação	procedural
Aurora	núcleo	software	aplicação	procedural

Tabela 3.1: Comparação das características dos sistemas estudados

Capítulo 4

Alcançando extensibilidade através da reflexão computacional

No modelo proposto, baseado no paradigma de reflexão computacional em sistemas operacionais orientados a objetos, ao invés de esconder os componentes do sistema, eles são descritos em meta-objetos e agrupados em meta-espaços. Através destas estruturas o sistema operacional armazena sua própria descrição e ainda recebe meta-informações das aplicações com instruções de como deverá se comportar o ambiente de execução. Este conjunto de dados pode ser acessado e utilizado como base para a adaptação dos serviços do sistema, visando oferecer da melhor forma possível os recursos solicitados pelas aplicações.

A representação do sistema é descrita em entidades de forma abstrata. Sendo assim, esta arquitetura provê a capacidade de utilização de componentes genéricos oferecendo alto grau de reusabilidade de código, evitando redundância e reduzindo as possibilidades de erro. Estes componentes podem ser escritos para oferecer recursos a um determinado domínio de aplicações sendo incorporado a estas de forma transparente.

A flexibilidade e extensibilidade do sistema, somente foram possíveis, devido ao uso da reflexão computacional. Esta possibilitou o desenvolvimento do modelo de um núcleo extensível que é adaptado em tempo de execução. O paradigma de reflexão computacional também foi utilizado como premissa principal no desenvolvimento do sistema operacional Aurora[1][ZAN 97], que serviu de base para o modelo proposto.

Outro sistema operacional de grande valia para o desenvolvimento deste trabalho foi o Apertos[2] [YOK 92]. Este sistema abordou uma concepção revolucionária na sua época e até

[1]A descrição do sistema operacional Aurora pode ser encontrada na sessão 3.3.7.

[2]A descrição do sistema operacional Apertos pode ser encontrada na sessão 3.3.5.

hoje serve como base para desenvolvimento de novas pesquisas e modelos utilizando reflexão computacional.

4.1 Modelo de um Núcleo Extensível

A extensibilidade de um sistema operacional é definida pela possibilidade do sistema se adaptar às características solicitadas pela aplicação que está em execução. Esta adaptação deve acontecer de forma dinâmica e oferecer a oportunidade de incluir ou excluir serviços previamente existentes no sistema operacional e também serviços externos ao sistema.

O objetivo desta adaptação é prover um ambiente especializado para cada aplicação, com o intuito de alcançar maior desempenho ou disponibilizar recursos que não estavam previstos no desenvolvimento e implementação do sistema. Outra vantagem desta adaptação é a capacidade de atualizar serviços que sofreram importantes modificações.

No modelo proposto, existem duas formas de utilizar a extensibilidade do sistema. Uma delas é através de meta-informações contidas na aplicação que será executada e a outra forma é através de ações tomadas pelo próprio sistema operacional levando em consideração o seu estado atual de execução. A implementação e o funcionamento das meta-informações serão discutidos na sessão 4.1.1, e casos onde o sistema adapta-se de acordo com a sua necessidade podem ser encontrados na sessão 4.1.4.

Devido a questões de implementação e controle sobre o sistema, o modelo proposto possuí apenas o nível base e um meta-nível. No nível base ficam alocados os objetos das aplicações e no meta-nível, ficam alocados os meta-objetos do sistema, tanto os nativos do sistema como também os carregados pelas aplicações. Isso significa que os meta-objetos não podem refletir sobre si mesmos, a única reflexão que acontece é sobre os objetos do nível base.

De acordo com as estratégias adotadas, quando um meta-objeto é carregado no meta-nível, este se torna estático, podendo ser descarregado, substituído por outro, mas não pode modificar o seu próprio comportamento ou estrutura. Esta medida foi tomada para evitar uma torre de reflexão infinita, provendo desta forma o maior controle do sistema operacional sobre as aplicações e extensões. Outra facilidade alcançada foi na simplificação do processo de desenvolvimento de extensões, pois estas, uma vez carregadas no sistema não refletem sobre si mesmas, apenas utilizam a reflexão computacional para chamar métodos e obter dados de execução.

Dentro do meta-nível, podem existir diversos meta-espaços. Cada objeto do nível

base terá seu próprio meta-espaço [3], dando a impressão de que é o único objeto ativo no sistema. Os meta-espaços podem compartilhar meta-objetos, ou adicionar ao seu espaço aqueles que julgarem necessário. Estes meta-objetos representam as funcionalidades do sistema que estarão presentes para determinado grupo de objetos. O único meta-objeto compartilhado por todos os meta-espaços é o `mGerenciador`.

Todas as extensões do sistema operacional ficam alocadas no meta-nível, dentro do espaço de endereçamento do usuário, não necessitando uma execução protegida, pois no caso de uma falha esta não afetará os serviços oferecidos pelo núcleo do sistema e para a extensão voltar a funcionar depois da falha, basta realocá-la. Devido a grande variedade de operações que as extensões podem oferecer, é responsabilidade delas identificar se uma mensagem ou comando é referente a um de seus serviços, desta forma especializando ainda mais a extensão e diminuindo a complexidade do sistema operacional.

Com o objetivo de atingir um alto grau de flexibilidade e extensibilidade no sistema, o desenvolvimento das extensões deve seguir o conceito de containers[WU 98]. Através da utilização desta estratégia um meta-objeto pode ser trocado por outro meta-objeto que implemente as mesmas funcionalidades básicas, mas de diferentes formas e até mesmo utilizando diferentes políticas. Esta técnica difere-se da utilizada pela decomposição e abstração[4] do sistema, sendo que apesar de apresentar uma interface rígida permite a modificação interna dos componentes.

Para este modelo, consideramos que cada objeto do nível base deve ser “completo”, ou seja, deve possuir todos os recursos que necessitará durante a execução, não fazendo referências a classes ou objetos externos e somente será capaz de carregar extensões na forma de meta-objetos no meta-nível do sistema.

Dentro do modelo proposto existem alguns meta-objetos que desempenham papéis essenciais para o funcionamento do sistema. Um deles é o `mTerminalAcesso`. Este meta-objeto tem por função básica servir como interface com o usuário e realizar a inicialização dos serviços do sistema. Através deste meta-objeto que são recebidas as solicitações de carga de aplicações e objetos do sistema. Mais informações sobre o funcionamento deste meta-objeto podem ser encontradas na sessão 4.1.3.

Além do `mTerminalAcesso`, um dos principais meta-objetos do sistema é o `mGerenciador`. Este meta-objeto tem por objetivo gerenciar o ambiente de execução. Tal

[3] A descrição completa do funcionamento dos meta-espaços é apresentada na sessão 4.1.2

[4] A técnica de decomposição e abstração foi discutida na sessão 1.1.1 e é apontada como uma das causas pela falta de flexibilidade dos sistemas.

gerenciamento é realizado através da interceptação das chamadas acionadas pelos objetos no nível base, a partir das quais é possível fazer uma análise para obter os seus requisitos, e se necessário, repassar estas chamadas as extensões alocadas. Este meta-objeto é o único que sempre estará presente no sistema. Uma descrição detalhada do funcionamento do meta-objeto `mGerenciador` pode se encontrada na sessão 4.1.4.

Outro meta-objeto importante ao sistema é o `mCarregador`. Este meta-objeto tem como função prover a alocação de extensões no meta-nível. A solicitação de quais extensões devem ser carregadas pode ser feita através das meta-informações passadas em tempo de inicialização da aplicação ou durante a execução. A solicitação de carregamento é interceptada pelo `mGerenciador` e repassada ao `mCarregador`. Se não houver nenhum carregador de classes alocado, será utilizado o carregador padrão da máquina virtual. Mais informações sobre o meta-objeto `mCarregador` podem ser encontradas na sessão 4.1.5.

O meta-objeto `mEscalonador`, como o próprio nome diz, tem como objetivo realizar o escalonamento dentro de um determinado meta-espaço. Como cada objeto no sistema tem seu próprio meta-espaço, este meta-objeto se limita a escalonar fluxos de execução(*threads*) que um determinado objeto possa ter. A descrição completa deste meta-objeto é apresentada na sessão 4.1.6.

A persistência de um meta-espaço é definida pelo meta-objeto `mPersistência`, que vai desde a inexistência dela, no caso de um ambiente não persistente, até a utilização de servidores remotos para esta finalidade. A principio a persistência é totalmente transparente para a aplicação, ou seja, quando existir um meta-objeto `mPersistência`, o meta-espaço inteiro(objeto no nível base e meta-objetos) estão sendo gravados de alguma forma, porém nada impede o desenvolvimento de uma extensão que realizem a persistência de outras formas. Mais informações sobre o meta-objeto `mPersistência` podem ser encontradas na sessão 4.1.7.

A função de rastreamento do sistema pode ser obtida através do meta-objeto `mTrace`. Este meta-objeto provê uma funcionalidade interessante, pois ele permite que todas as chamadas executadas pelo objeto no nível base, sejam exibidas. Esta funcionalidade pode ser útil durante a realização de testes no sistema, permitindo localizar falhas rapidamente. Sem este meta-objeto a própria aplicação teria que implementar estas funções, poluindo o código fonte. Outras informações sobre o meta-objeto `mTrace` podem ser encontradas na sessão 4.1.8

E por último, porém não menos importante, o meta-objeto `mCache`. Este meta-objeto permite realizar a armazenamento de informações que são consideradas importantes e que podem vir a ser utilizadas em um período próximo. O melhor exemplo para isto, são recursos

multimídias, como vídeos. Neste caso a melhor política de armazenamento temporário, seria conforme a variação do tempo, não seria interessante armazenar todo o vídeo no momento da sua inicialização, correndo o risco de limitar a memória de outras aplicações, porém também não seria interessante ter este vídeo interrompido temporariamente por falta de dados. Outras informações sobre o meta-objeto `mCache` são apresentadas na sessão 4.1.9

4.1.1 Meta-informações

A extensibilidade do sistema pode ser acessada e/ou modificada através do uso de meta-informações que são passadas pela aplicação no momento em que esta é carregada ou a qualquer instante durante sua execução. Uma vez que estas meta-informações estiverem disponíveis ao sistema podem ser tomadas decisões levando em consideração as dados recebidos e o estado atual do sistema.

Por convenção, decidiu-se utilizar variáveis comuns e funções vazias para a passagem das meta-informações. Estas variáveis acabam agindo como palavras reservadas no sistema e não tem nenhum efeito no nível base, somente no meta-nível. Elas precisam seguir um determinado padrão(figura 4.1) para serem interpretadas adequadamente pelo `mGerenciador`.

A existência de duas formas de disponibilizar as meta-informações para o meta-espaço é devido a possibilidade da utilização de alguma ferramenta que realize a verificação estática das necessidades das aplicações e adicione ao cabeçalho do código fonte da mesma os serviços que melhor atenderiam os requisitos encontrados. Existem pesquisas nesta área que buscam os requisitos necessários para execução de uma determinada aplicação[POL 05, STE 06, WIE 05], porém apresentam somente resultados satisfatórios para alguns casos.

Como pode ser observado na figura 4.1, as variáveis de meta-informações que serão utilizadas no carregamento da aplicação precisam ser do tipo "`private static String`" e possuir os caracteres "`metainf_`" no início do nome da variável. As meta-informações para adaptar o sistema em tempo de execução, são passadas através de um método vazio – "`public void metaInf_(String mi){};`" – que é reconhecido pelo `mGerenciador` como meta-informações sendo passadas pela aplicação, a partir das quais ele tomas as providencias necessárias.

No modelo proposto existem três meta-instruções reconhecidas, que são o `load`, `set` e `unset`, que podem ser passadas por variáveis no momento da inicialização ou pela função reservada durante a execução. Logo após a meta-instrução, é necessário passar um "identificador"

```
1  class HelloWorld {
2
3          // Conjunto de meta-informacoes
4          private static String metainf_load_Scheduler="meuEscalonador";
5          private static String metainf_set_Scheduler="meuEscalonador";
6
7          private static String metainf_load_Class="extensao1,extensao2";
8          private static String metainf_set_Class="extensao1,extensao2";
9
10         public void metaInf_(String mi){};
11         // Fim ...
12
13         public static void main( String[] args ) {
14                 metaInf_("load_Class=minhaExtensao");
15                 metaInf_("set_Class=minhaExtensao");
16                 System.out.println( "Hello␣world!" );
17                 metaInf_("unset_Class=minhaExtensao");
18         }
19 }
```

Figura 4.1: Exemplo de meta-informações sendo passadas através de variáveis

que serve para mostrar o tipo da extensão que esta sendo alocada. Existem alguns tipo de extensões pré-definidas que são: `_Scheduler`,`_Persistence`, `_Trace`, `_Loader` e o tipo genérico `_Class`. Mais informações sobre cada uma delas, podem ser encontradas na sessão 4.1.4.

Através da meta-instrução `load` é possível solicitar o carregamento de uma classe externa ao sistema. Esta classe será alocada no meta-nível na forma de um meta-objeto, ou seja, uma extensão do sistema, e passará a fazer parte dos serviços disponibilizados pelo sistema operacional. É possível através desta instrução solicitar a carga de várias extensões ao mesmo tempo, basta separá-las por vírgula. Esta meta-instrução não permite o carregamento de classes no nível base, para tal operação deve ser utilizado o carregador de classes padrão do sistema, ou algum outro carregador pré-estabelecido em tempo de execução.

Se a aplicação não solicitar o carregamento de nenhum meta-objeto, não haverá meta-carregador neste meta-espaço, todavia no caso do recebimento da meta-instrução `load` será alocado o carregador padrão do sistema e utilizado. Se o objetivo desta meta-instrução for jus-

tamente alocar um carregador externo, primeiramente deve ser alocado e utilizado o carregador padrão do sistema que em seguida será substituído pelo carregador definido pelo usuário. Mais informações sobre o funcionamento do meta-objeto carregador(`mCarregador`) pode ser encontrado na sessão 4.1.5.

A instrução `set` ativa um determinado meta-objeto, que pode ter sido carregado pela aplicação ou previamente existente no sistema. No caso da existência de um meta-objeto para a função desejada este será substituído pelo novo. Por exemplo, na figura 4.1 a aplicação esta carregando e ativando um escalonador de fluxo de execução externo ao sistema denominado neste caso de "`meuEscalonador`", se houver um escalonador nesse meta-espaço ele ser substituído pelo novo, se não houver o novo será carregado e começará a sua execução.

Se a meta-classe desejada não existir no sistema, ela deve ser previamente carregada através da utilização da meta-instrução `load`, mesmo quando alocada ela ainda não estará recebendo informações do `mGerenciador`, sendo que somente passará a fazer parte do sistema, para que as aplicações possam utilizar seus recursos e serviços, depois da ativação realizada através da meta-instrução `set`.

A instrução `unset` tem como função básica promover a retirada de um meta-objeto de um determinado meta-espaço. Devido a transparência dos meta-espaços em relação às aplicações, esta instrução não precisa se preocupar com qual meta-espaço ela esta sendo referenciada, pois sempre será o meta-espaço de execução da aplicação solicitante. Se o objeto a ser retirado do sistema não estiver presente em mais nenhum meta-espaço este será retirado do sistema também.

4.1.2 Meta-espaços

O conceito de meta-espaço é transparente a aplicação, para ela existe apenas um meta-espaço, que é o qual ela pertence. De uma maneira geral, será criado um meta-espaço por objeto no nível base e somente objetos correlatos ocuparão o mesmo meta-espaço se solicitado. A idéia dos meta-espaços é agrupar as características necessárias para o melhor funcionamento da aplicação, oferecendo um ambiente em que a aplicação possa realizar modificações conforme a sua vontade sem colocar em risco a execução de outros objetos.

A figura 4.2 mostra o estado do sistema com 2 meta-espaços criados. O "`ObjetoA`" está associado ao meta-espaço 1, que por solicitação da aplicação ou decisão do sistema, disponibiliza o carregador padrão(`mCarregador`) e um meta-objeto de rastreamento(`Trace`).

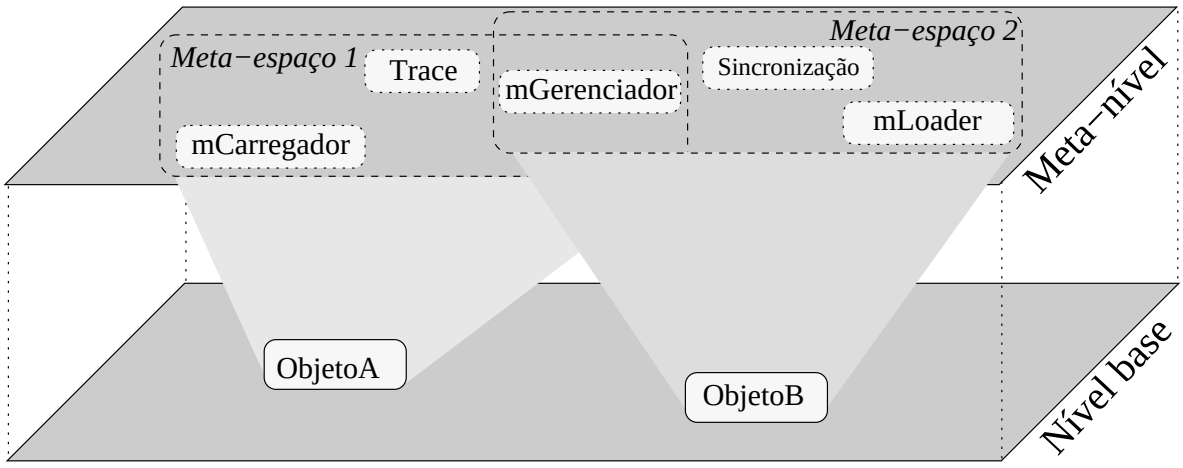


Figura 4.2: Sistema com dois meta-espaços criados

O "`ObjetoB`" esta associado ao meta-espaço 2, e possuí um carregador próprio chamado `mLoad` e um objeto de sincronização carregado e acionado pela aplicação.

É possível encontrar uma diferença em relação ao modelo proposto e o apresentado pelo Apertos. No modelo do Apertos o sistema operacional tenta agrupar objetos com características semelhantes. O tempo gasto para tal operação vai depender do número de objetos no sistema, e pode ser expressivo se este estiver com muitos objetos alocados. E ainda, toda vez que um objeto muda seus requisitos seria necessário uma busca nos meta-espaços para encontrar a função desejada e poder realocar este objeto em outro meta-espaço ou adicionar a funcionalidade ao seu meta-espaço.

Realizando estes tipos de operações, podem ser adicionadas funcionalidades que não serão úteis para todos os objetos que estão povoando aquele determinado meta-espaço, gerando assim inconsistências, pois uma das premissas do sistema é adaptar-se às necessidades específicas de cada aplicação. Outra solução seria, ao invés de buscar meta-espaços com as características solicitadas, criar um novo meta-espaço específico para este objeto, agindo desta forma pode haver um considerável aumentando no grau de complexidade de gerenciamento destes meta-espaços. Levando em consideração estes fatos o modelo apresentado cria um novo meta-espaço por objeto e o único meta-objeto obrigatoriamente compartilhado por todos é o `mGerenciador`.

4.1.3 Meta-objeto `mTerminalAcesso`

O meta-objeto `mTerminalAcesso` é responsável por implementar a interface de comunicação com o usuário. Desta forma, no modelo proposto este é o primeiro meta-objeto

a ser carregado e ficará a disposição para receber o nome de classes que devem ser alocadas no sistema. Como pode ser observado na figura 4.3, a primeira ação quando recebida uma chamada de alocação proveniente do usuário é carregar a referida classe, para isto é utilizado o carregador padrão da máquina virtual, pois este objeto será alocado no nível base do sistema e não é possível que ele escolha o seu carregador, somente o carregador dos meta-objetos subseqüentes.

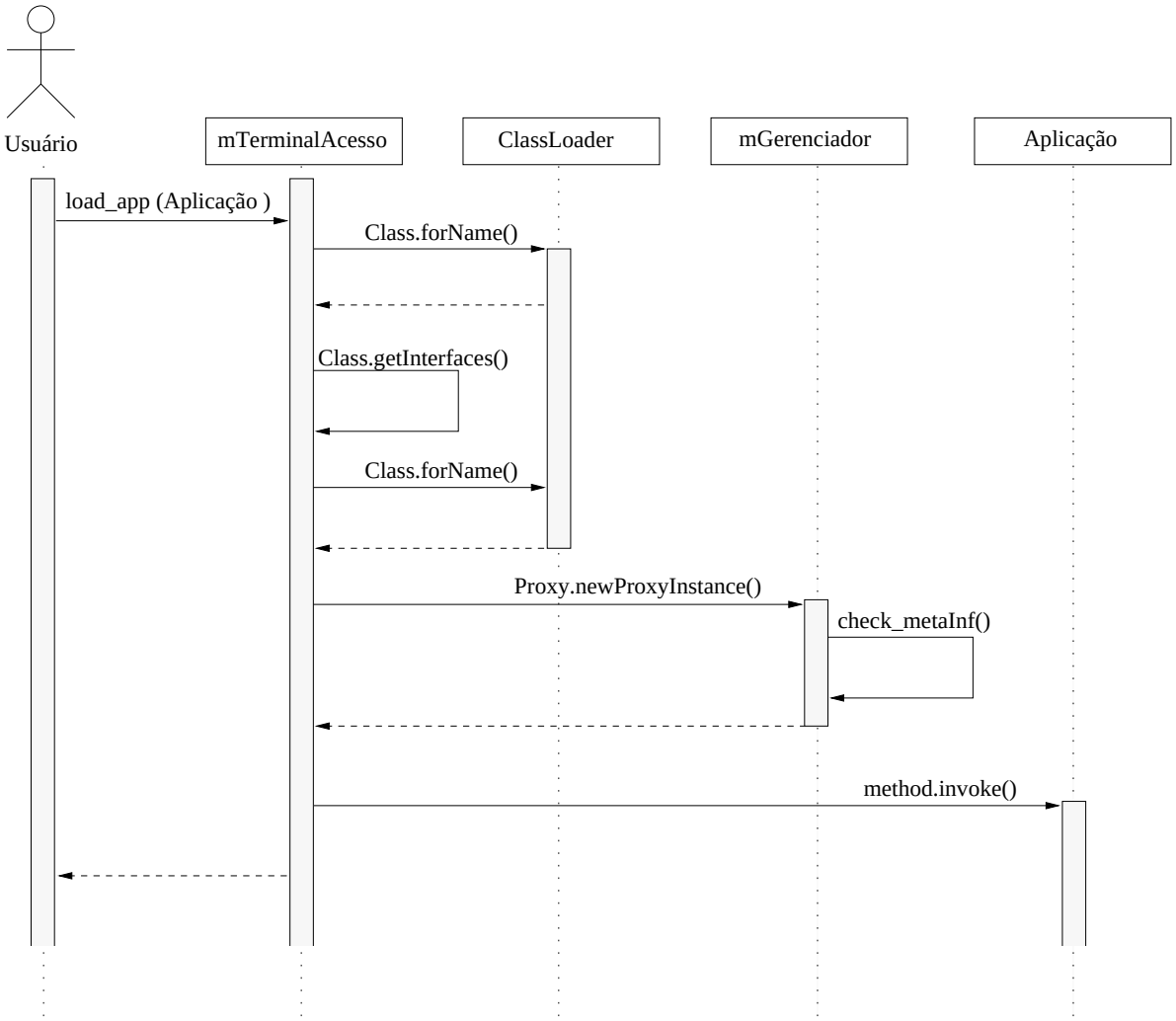


Figura 4.3: Esquema de carregamento de uma aplicação

Logo após a carga da classe deve ser obtida sua interface, esta operação é possível através da reflexão oferecida pela linguagem Java, em seguida esta é carregada no sistema, a partir disto, é criado um *proxy* de objetos relacionando-a com o `mGerenciador` [5], esta operação pode ser visualizada na figura 4.3 com a realização da chamada `Proxy.newProxyInstance`, e constituí-se como uma operação padrão do sistema, pois o meta-objeto `mGerenciador` deverá estar presente em todos os meta-espaços e interceptar as chamadas provenientes das classes

[5] O meta-objeto `mGerenciador` será detalhado na sessão 4.1.4

carregadas no nível base do sistema.

Durante o processo de criação do *proxy* de objeto, o `mGerenciador` verifica se a classe que está sendo utilizada tem alguma meta-informação. Se tiver, estas serão obtidas, analisadas e suas solicitações atendidas. Após processo de inicialização a classe torna-se um objeto que está pronto para executar. A última operação do `mTerminalAcesso`, como pode ser observado na figura 4.3, é executar o método `main()` do objeto, isto é possível através da função `method.invoke()` que faz parte do pacote `java.lang.reflect`.

4.1.4 Meta-objeto `mGerenciador`

O meta-objeto `mGerenciador` é o responsável pelo gerenciamento do sistema como um todo. Este é o único meta-objeto presente em todos os meta-espaços, e tem como função básica a criação e definição das características dos meta-espaços através de meta-informações recebidas do objeto no nível base ou a partir do seu próprio conhecimento do sistema, e o gerenciamento e ativação das extensões.

Houve a necessidade da criação deste meta-objeto de forma global, pois a linguagem de programação Java não oferece nativamente um MOP(*Meta-Object Protocol*) o qual proporcionaria a possibilidade de interceptação das mensagens entre objetos. Após estudo de soluções existentes[DOW 01, TAN 01, OLI 99, GOL 98, KLE 98], optou-se pela não adoção das mesmas, com o objetivo de manter a compatibilidade com outros sistemas e em trabalhos futuros facilitar a interconexão do modelo com um sistema operacional real.

A solução encontrada foi a implementação de um *proxy* de objetos dinâmico global(o `mGerenciador`). Assim todos os objetos do sistema, no momento da alocação na memória são ligados com o meta-objeto que atua como um *proxy*, recebendo as requisições, realizando analises, se necessário executando ações e devolvendo as requisições ao objeto de origem. Cada objeto alocado no nível base é relacionado ao seu próprio meta-espaço, que é criado dinamicamente pelo `mGerenciador`. Dentro do seu meta-espaço, o objeto tem permissão de realizar modificações e adicionar extensões, com o objetivo de criar um ambiente de execução que atenda as suas necessidades específicas.

As extensões, são alocadas em uma meta-lista que é criada juntamente com o meta-espaço e mantida pelo próprio `mGerenciador`. A meta-lista é composta por uma parte de tamanho fixo e uma parte de tamanho variável. A parte de tamanho fixo deve ser utilizada para o armazenamento de serviços pré-definidos e comumente encontrados na arquitetura de sistemas

operacionais. Já a parte variável, é um espaço de alocação genérico e é responsável por armazenar quaisquer outros meta-objetos carregados pelas aplicações.

A parte da meta-lista de tamanho fixo, como pode ser observado na figura 4.4, é responsável por armazenar os apontadores de serviços básicos e comuns na arquitetura de sistemas operacionais. Ela armazena o apontador de onde está o meta-objeto escalonador(`Scheduler`), o meta-objeto persistência(`Persistence`), o meta-objeto rastreador(`Trace`), o meta-objeto carregador(`Loader`) e por último a parte variável da meta-lista, que permite a inclusão de qualquer tipo de extensão e armazena um apontador para uma lista encadeada(`Class`) gerada e manipulada dinamicamente.

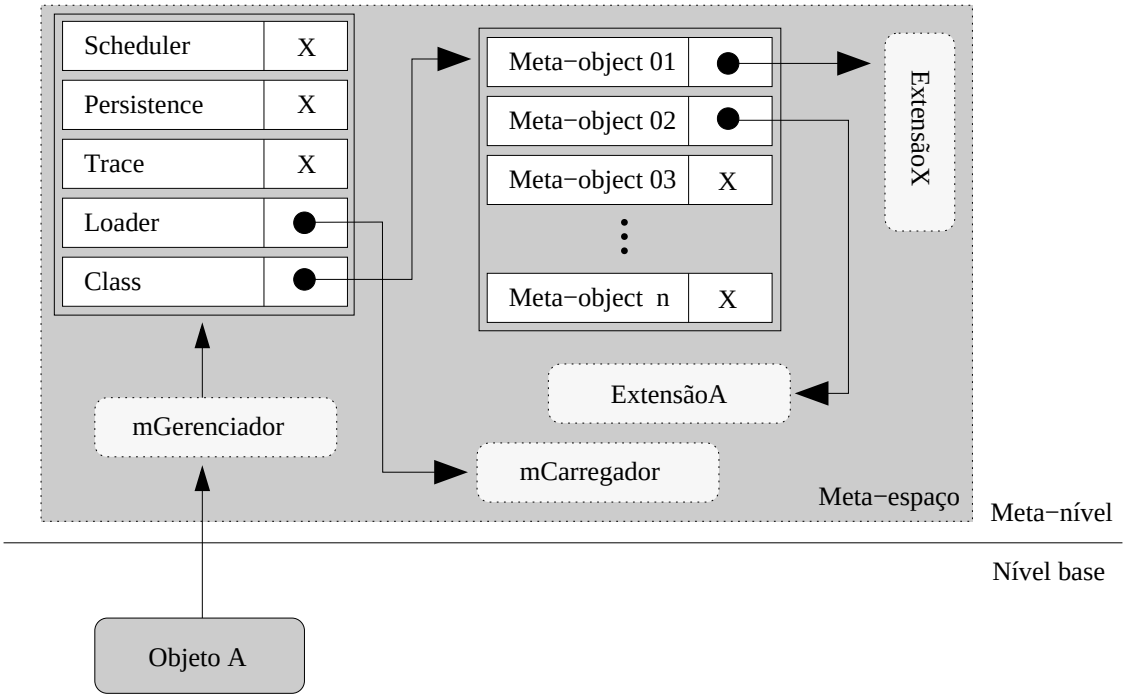


Figura 4.4: Meta-lista com as informações sobre meta-objetos

No exemplo apresentado na figura 4.4, temos um objeto no nível base(`Objeto A`) que supostamente está tentando executar alguma operação. Quando a operação for acionada, o meta-objeto `mGerenciador` interceptará esta ação, pois eles estão inter-conectados através do *proxy* de objeto, e o controle do sistema passará para o meta-nível onde será realizada uma analise. O `mGerenciador` sabe que nenhum dos meta-objetos da parte fixa da meta-lista recebe chamadas de verificação[6], baseado na seguinte definição: Os meta-objetos armazenados nas por-

[6]Chamadas de verificação são enviadas pelo `mGerenciador` para cada uma das extensões presentes na parte genérica da meta-lista, com o objetivo de descobrir se uma determinada extensão é responsável por atender aquela chamada do nível base.

ção fixa da lista, quando presentes, responderão todas as chamadas, seja ela qual for, devido as características dos serviços prestados por eles.

Ou seja, um meta-objeto escalonador estará sempre executando, definindo quais dos fluxos devem continuar e quais serão interrompidos. A mesma situação acontece com o rastreador, quando este for habilitado no meta-espaço, deverá agir em todas as chamadas, e ainda a mesma necessidade do meta-objeto de persistência que precisa manter-se informado sobre as modificações do sistema para não produzir inconsistências nos dados. O meta-objeto `mCarregador`, é conhecido pelo `mGerenciador` e só receberá chamadas de carregamento de classes. Todas estas extensões alocadas na parte fixa da meta-lista, devem implementar ao menos uma função, que pode ser visualizada na interface demonstrativa da figura 4.5.

```
1 public interface interface_extensao{
2         public abstract int run_MetaCall(Object MetaInf);
3 }
```

Figura 4.5: Interface de uma extensão alocada na parte fixa da meta-lista

Eliminando as extensões da parte fixa da meta-lista, a busca é iniciada na parte variável. Para encontrar o meta-objeto necessário, o `mGerenciador` executa uma chamada de verificação, passando a mensagem recebida por ele e o controle de execução para a primeira extensão, como é obrigação das extensões reconhecer as chamadas que são referentes aos seus serviços, ela deverá responder se é sua responsabilidade ou não. Para isto, todas as extensões da parte genérica da meta-lista precisam ter duas funções específicas(como pode ser observado na figura 4.6).

```
1 public interface interface_extensao{
2         public abstract boolean check_MetaCall(Object MetaInf);
3         public abstract int run_MetaCall(Object MetaInf);
4 }
```

Figura 4.6: Interface de uma extensão alocada na parte genérica da meta-lista

A primeira função(figura 4.6 linha 2) é a chamada de verificação em si, que serve para identificar se uma determinada ação pertence a extensão que está sendo consultada. Existem somente duas respostas possíveis, “`sim`” e “`não`”, respectivamente `True` e `False` para o caso da

variável `boolean`. Com base na resposta o `mGerenciador` pode tomar uma decisão e executar a segunda função(figura 4.6 linha 3) que permitirá que a extensão execute os seus serviços. A resposta para esta segunda função possuí quatro alternativas, "`execute e retorne`", "`execute e continue`", "`não execute e retorne`", "`não execute e continue`". As suas equivalências para o retorno da variável do tipo inteiro estão descritas nas tabela 4.1.

Valor	Equivalência
0	Execute e retorne
1	Execute e continue
2	Não execute e retorne
3	Não execute e continue

Tabela 4.1: Equivalências dos valores de retorno da execução de extensões

Desta forma, se a extensão, após a execução responder "`execute e retorne`", o `mGerenciador` executará o método e retornará o processo de execução ao nível base. A segunda opção de resposta é "`execute e continue`", neste caso, será executado o método e continuará na busca por outras extensões, e o método pode ser executado novamente. A terceira opção de resposta é "`não execute e retorne`", onde o método não será executado e o processo de execução retornará ao nível base. A última opção de resposta é "`não execute e continue`" onde o método não será executado, mas a busca por extensões continuará.

Por exemplo, um caso onde pode ser utilizada a resposta "`não execute e continue`" é na implementação de um rastreador com características diferentes do `mTrace` e alocado no espaço genérico da meta-lista, que na verdade não deve executar a operação, apenas armazenar o método chamado e os parâmetros passados. Outra extensão que poderia ser executada e permitir a continuidade do processo de busca ou a sua interrupção seria um meta-objeto de controle de acesso. Este meta-objeto poderia armazenar em suas estruturas internas uma lista de recursos e/ou operações permitidas para um determinado meta-espaço, e quando identificada uma ação não permitida, poderia retornar o controle ao nível base sem executar a determinada operação, ou se esta fosse permitida, continuaria a execução no meta-nível, retornando o controle ao nível base quando terminada a execução.

Para fins de implementação, a resposta padrão após a execução de uma extensão é "`execute e retorne`", sendo assim se nenhuma resposta for explicitamente passada

o mGerenciador realizará o procedimento de retorno do controle da execução para o nível base. No exemplo da figura 4.4, a primeira extensão consultada(ExtensãoX) responde que não é responsabilidade dela, retornando ao mGerenciador. Então é verificada a segunda extensão(ExtensãoA), que reconhece a chamada e executa a operação que foi solicitada, e poderá solicitar o retorno ao nível base através da resposta "execute e retorne". Desta forma, quem tem maior controle sobre o sistema são as extensões com maior tempo de vida, que serão consultadas primeiro, e que podem permitir a continuação da execução no meta-nível ou solicitar o retorno a nível base.

4.1.5 Meta-objeto mCarregador

No paradigma de orientação a objetos – especialmente em um sistema operacional que faça uso desta tecnologia – a necessidade de um carregador de classes que atenda as necessidades das aplicações é de suma importância. Se este carregador tiver a habilidade de adaptar-se as necessidades específicas de cada aplicação pode ser obtido um melhor resultado em relação a forma com que as classes serão carregadas e alocadas no sistema.

No modelo proposto, é possível a total adaptação do mCarregador, pois este pode ser desenvolvido e implementado especificamente para atender os requisitos de uma determinada aplicação ou um determinado conjunto de aplicações e ser carregado ou substituído em tempo de execução. A única exigência que deve ser satisfeita na implementação de uma extensão do tipo mCarregador é a utilização de uma interface com pelo menos um método chamado load_class,com pode ser observado na figura 4.7.

```
1 public interface inter_mCarregador{
2         public abstract Class load_class(nomeClasse String);
3 }
```

Figura 4.7: Interface básica para implementação de uma extensão do tipo mCarregador

As funções do mCarregador são relativas a alocação de classes no meta-nível e a passagem de meta-informações que estas classes podem conter ao mGerenciador do sistema. No caso da alocação de classes, pode ser utilizado o carregador nativo do Java, transformando assim, o mCarregador em um simples *wrapper* para esta função. Em uma forma mais

avançada, o próprio `mCarregador` pode implementar suas funções de carregamento com as características que considerar mais apropriada.

As meta-informações obtidos pelo `mCarregador` e passadas ao `mGerenciador` são utilizadas para adaptar ou estender o funcionamento do sistema no momento da inicialização da aplicação, criando assim, um meta-espaço especializado com todos os serviços necessários. Quem realizada todas as operações de modificação é o `mGerenciador`, ficando a cargo do `mCarregador` apenas re-passar as meta-informações.

Como mencionado anteriormente, a implementação das extensões do sistema, como é o caso do `mCarregador`, deve seguir algumas premissas básicas. Pelo fato desta extensão ter um espaço reservado na parte fixa da meta-lista, ela recebe chamadas diretamente do `mGerenciador`, que conhece previamente sua função no sistema, então, toda vez que o `mGerenciador` receber uma meta-instrução `load` ele vai repassar esta instrução para o `mCarregador`, se ainda não existir um carregador alocado, será utilizado o carregador nativo do sistema.

4.1.6 Meta-objeto `mEscalonador`

De acordo com as características estruturais do modelo, o escalonamento de tarefas de um meta-espaço torna-se uma operação simples, porque dentro de cada um destes agrupamentos de meta-objetos existe somente um objeto no nível base, que necessita ser escalonado. Este escalonamento é realizado através da máquina virtual[7], utilizando suas políticas internas para isto. Para efeitos de implementação e simulação, consideraremos que automaticamente quando um objeto do nível base é escalonado todo o seu meta-espaço entrará em funcionamento também.

A partir desta premissa, podemos definir que a prioridade de execução de um meta-espaço será a mesma prioridade de seu objeto no nível base, para o qual o meta-espaço foi originalmente criado. Outra constatação que pode ser feita, é que o único tipo de escalonamento que poderá acontecer dentro de um meta-espaço é o escalonamento de fluxos de execução(*threads*) que são mais simples de serem tratados, pois não envolvem trocas de contexto.

Dentro do modelo proposto, o próprio `mEscalonador` deve ser implementado utilizando-se *threads*, para não depender do acionamento de chamadas para sua execução, sendo assim, uma vez adicionada a extensão de um meta-objeto escalonador este passaria a operar ininterruptamente até sua substituição ou desativação. As políticas implementadas por um meta-objeto deste tipo podem ser as mais variadas e tem a possibilidade de ser substituídas em tempo de exe-

[7] Não é objetivo deste trabalho detalhar o funcionamento do escalonador na máquina virtual Java.

cução através do carregamento de um novo meta-objeto escalonador.

A substituição deste meta-objeto é simples, pois como ele escalona somente *threads*, que são auto-contindas, ou seja, possuem todas as informações que precisam para executar dentro de si mesma, não existe a necessidade de repassar nenhuma informação ao meta-objeto que será alocado no seu lugar, pois este começará a execução da forma que preferir e sem se preocupar como o procedimento que era realizado no passado.

Outra característica interessante é que este meta-objeto executa independentemente das extensões ativas no sistema e também do objeto no nível base, desta forma, ele não precisa seguir nenhuma interface padrão de comunicação ou ativação. E ainda, como mencionando anteriormente, as *threads* são auto-contidas, então não será necessário que este meta-objeto tenha métodos específicos para manipulação, ficando a cargo do desenvolvedor da extensão realizar, da sua maneira, a implementação.

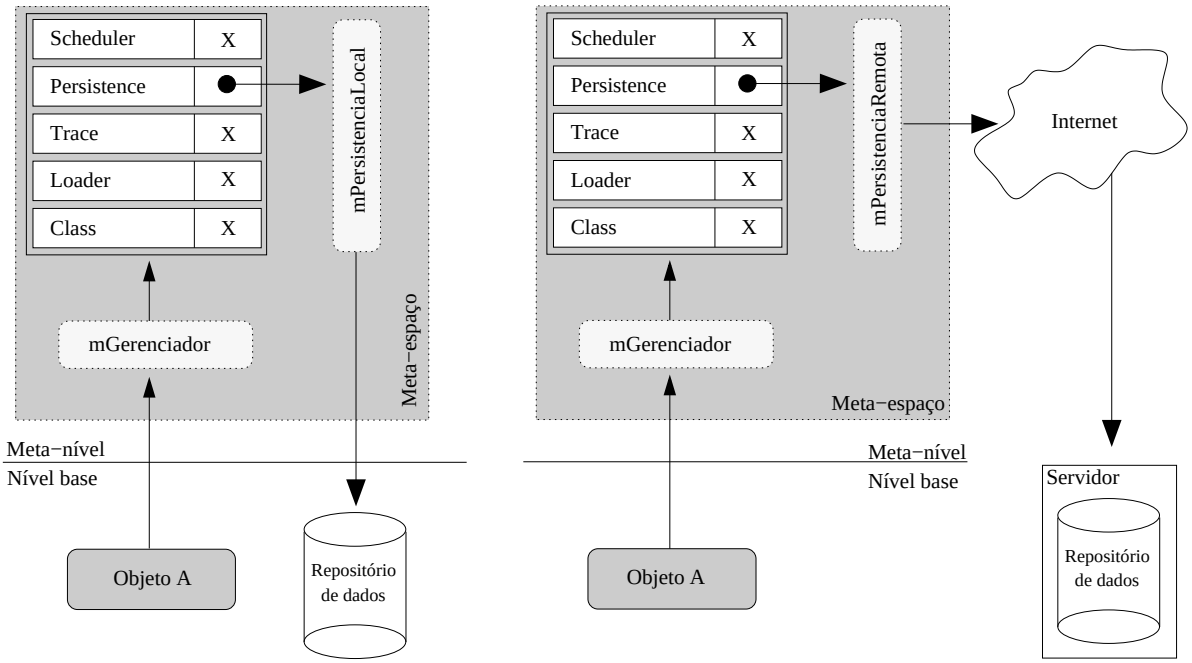
Com o objetivo de possibilitar a simulação deste meta-objeto na máquina virtual Java, foram utilizadas as funções de gerenciamento de *threads* para interromper a execução de métodos, pois não foi possível obter acesso ao escalonador nativo da plataforma. A função utilizada para interromper momentaneamente a execução de um método foi a `wait()`, inserida em tempo de projeto no objeto. E o meta-escalonador implementou a função `notify()` para retornar a execução.

4.1.7 Meta-objeto `mPersistência`

No modelo proposto, a persistência de um objeto e do seu meta-espaço é tratada de forma transparente a aplicação. Se nenhuma extensão de persistência for alocada e ativada, este sistema não manterá seus dados e estados de execução após uma reinicialização sendo que, somente será possível realizar este procedimento se um objeto de persistência for alocado. A alocação do meta-objeto persistência pode acontecer da mesma forma que todos os outros, ou seja, através das palavras reservadas durante o carregamento da aplicação ou através das funções reservadas a qualquer momento da execução.

Uma vez realizados a alocação e ativação desta extensão (figura 4.8), o `mGerenciador` reconhece através da sua posição na meta-lista, que este é um meta-objeto que sempre requer o redirecionamento de todas as chamadas para si. Através destas chamadas será possível identificar modificações nos objetos e executar o procedimento de persistência sobre eles. Como pode ser observado na figura 4.8(a), por exemplo, durante a inicialização da aplicação foi carre-

gada uma extensão que proverá a persistência local do meta-espaço, ou seja, todos os objetos e meta-objetos serão gravados no dispositivo de armazenamento oferecido pelo equipamento onde esta sendo executada a aplicação. Esta operação de persistência[8] será totalmente transparente para as aplicações e para os meta-objetos.



(a) Persistência local (b) Persistência remota

Figura 4.8: Persistência local e remota através de uma extensão

A qualquer momento no período de execução da aplicação, como ilustrado na figura 4.8(b), a extensão de persistência pode ser substituída. De acordo com as necessidades da aplicação esta operação, por exemplo, poderia passar a ser realizada enviando-se os dados para um servidor remoto. Neste exemplo, os objetos e meta-objetos deste meta-espaço seriam enviados através de uma conexão de rede ou outro meio de transmissão, para um servidor de dados que armazenaria estas informações. É de responsabilidade da extensão, realizar todo o processo, desde a autenticação com o servidor que será utilizado, até a transferência dos dados e verificação da consistência dos mesmos. Todos estes procedimentos não devem ser conhecidos pela aplicação, mantendo a persistência de dados transparente.

Esta é uma das formas de se implementar um meta-objeto de persistência, porém devido a grande flexibilidade do sistema nada impede que sejam desenvolvidas outras técnicas e maneiras tanto de utilizar como de implementar esta função. Por exemplo, um meta-objeto de

[8] A implementação de tal serviço não é o objetivo deste trabalho, por isso, não será detalhada.

persistência, que fique alocando na parte genérica da meta-lista e realize a persistência dos dados apenas do objeto do nível base, ou ainda que armazene somente as informações que considerar necessárias, em fim, as formas de implementações e funcionalidades oferecidas podem ser das mais variadas.

4.1.8 Meta-objeto `mTrace`

A existência de uma função de rastreamento no sistema é de extrema importância durante o desenvolvimento de um projeto de software complexo, como por exemplo, o de um sistema operacional. Através de uma função com esta, é possível localizar falhas no sistema mais rapidamente e identificar pontos com problemas ou que possam ser melhorados. Em alguns sistemas existentes, as funções de rastreamento misturam-se ao código das aplicações ou dos componentes do sistema, poluindo e dificultando o entendimento do sistema. Podem existir casos em que estas operações, mesmo quando não utilizadas, estejam presentes após a compilação, ocupando espaço na memória e possivelmente degradando o desempenho do sistema.

Uma solução mais dinâmica, pode ser implementada utilizando-se a técnica de programação orientada a aspectos[KIC 97]. Através desta técnica é possível separar as operações não funcionais da aplicação em diferentes arquivos de código fonte(como por exemplo, as funções de rastreamento), que serão “unidos” por um *weaver* através de indicações de *join points* e *cut points*. Somente após esta união, o código será compilado, tornando o processo estático(em tempo de compilação) sendo que em qualquer modificação realizada, todo o código do sistema precisará ser recompilado, e reinicializado.

No modelo proposto, é possível habilitar, desabilitar e modificar a forma de rastreamento do sistema em tempo de execução. Uma vez que o meta-objeto `mTrace` esteja ativo no sistema, este receberá todas as chamadas de funções acontecidas no seu meta-espaço, estando de posse destas chamadas, ele poderá realizar as operações que considerar necessárias com elas. Provendo assim, uma forma flexível de obter as informações sobre as ações do sistema.

Este meta-objeto, se alocado na parte fixa da meta-lista, deverá implementar a interface apresentada na figura 4.5, a qual é composta de apenas um método, pois este meta-objeto receberá todas as chamadas interceptadas. Devido à grande flexibilidade do modelo, é possível alocar um meta-objeto com estas características na parte genérica da meta-lista, porém desta forma, este meta-objeto deverá passar pelas chamadas de verificação e implementar a interface apresentada pela figura 4.6 com os dois métodos.

4.1.9 Meta-objeto `mCache`

O conceito por trás de um meta-objeto `mCache` seria a possibilidade de manipulação das políticas de alocação e utilização da memória *cache* do sistema, como isto não é possível em relação a máquina virtual, a idéia empregada neste modelo foi de um meta-objeto que mantém uma memória local, servindo de intermediário entre o armazenamento persistente em disco rígido e a aplicação que esta fazendo uso destes dados.

Uma das implementações possíveis, no caso de aplicações multimídias, poderia ser através de um *buffer* circular, onde os dados seriam lidos do disco rígido em uma periodicidade constante em relação ao tempo e armazenados neste *buffer*, desta forma, quando a aplicação solicitar estas informações, elas já estarão na memória volátil(*RAM*), o que significa um menor tempo de acesso em relação aos dispositivos de armazenamento persistente.

Este *buffer* circular precisaria manter um controle sobre os seus dados, para que os dados mais antigos não sejam sobrepostos pelos novos, em um caso de utilização ideal, onde a sincronização entre a aplicação e a extensão `mCache` fosse perfeita, não existiria a necessidade deste controle, pois sempre teriam informações para serem lidas e lugares para serem armazenadas novas informações.

Em operações deste tipo pode ser percebida a importância da flexibilidade do sistema operacional, pois em uma aplicação, por exemplo, de acesso a algum banco de dados, onde estas leituras são praticamente aleatórias, não adiantaria ter um meta-objeto `mCache` associado, e ainda pior, este meta-objeto realizaria operações totalmente desnecessárias, podendo comprometer o desempenho do sistema. Como citado anteriormente, em uma aplicação multimídia este meta-objeto seria de grande valia, devido ao fato de que em sistemas de uso geral, não é possível prever quais requisitos serão necessários, uma das melhores opções seria permitir a sua modificação deste ambiente em tempo de execução, adaptando-se as necessidades específicas das aplicações.

Da mesma forma que as outras extensões, é preciso implementar uma interface padrão[9], para receber as chamadas de verificação e de execução dos seus métodos. A parte referente à leitura das informações do dispositivo de mpersistencia deve ser implementada utilizando-se *threads* para permitir um preenchimento satisfatório do *buffer*, e as leituras destas são relativas aos acessos da aplicação.

[9] A interface padrão foi apresentada na sessão4.1.4, figura 4.6.

4.2 Resultados

Esta sessão apresenta os resultados obtidos da implementação do modelo proposto e a sua simulação em uma máquina virtual. Todos os testes mencionados, foram realizados no mesmo equipamento, um Pentium Centrino 1.6Ghz, com o sistema operacional Debian e o kernel 2.6.14, executando a maquina virtual Java da Sun versão 1.5.0_06. A precisão dos tempos obtidos foi em nanosegundos, sendo que estes foram convertidos para milisegundos com o objetivo de facilitar a visualização.

4.2.1 Tempos da máquina virtual Java

Com o objetivo de prover uma base para comparação com os tempos obtidos na simulação do modelo, foram realizados experimentos com o tempo de criação de objetos, acesso a variáveis (locais e externas) e chamadas de métodos. O primeiro teste foi realizado para obter os tempos referentes à criação de objetos. Inicialmente foi medido o tempo para a criação de objetos pela forma tradicional, através da operação "`obj = new Object()`", para comparação foi utilizado o método reflexivo, através da operação "`obj = Class.newInstance()`".

A média apresentada na tabela 4.2 em milisegundos foi calculada através de 100 interações do algoritmo sendo que em cada uma delas foram criados 500.000 objetos. O tempo necessário para a criação de objetos de forma reflexiva é aproximadamente 2.3 vezes maior que o tempo para a criação do mesmo objeto de forma convencional.

	Criação direta	**Criação reflexiva**
Tempo (ms)	86	198

Tabela 4.2: Média do tempo de criação de objetos na máquina virtual

O segundo experimento realizado foi em relação ao tempo de acesso as variáveis de objetos. Na primeira etapa do teste foi medido o tempo de acesso e gravação de uma variável local de um objeto, através da operação "`var = var + 1`", na segunda etapa, foi lida e gravada uma variável de um outro objeto ("`obj.var = (obj.var) + 1`"), e por último foi registrado o tempo através da utilização de reflexão computacional, que envolve a utilização do método `getDeclaredField()` para localizar a variável, o método `getInt()` para retornar o valor e `setInt()` para modificá-lo.

A tabela 4.3 apresenta a média dos tempos em milisegundos. Esta média foi calculada através de 100 interações do algoritmo, onde cada interação realizou 5.000.000 operações de leituras e escritas. Neste caso, é possível perceber uma grande diferença dos tempos em relação as técnicas tradicionais, pois a forma reflexiva apresenta um tempo aproximadamente 296 vezes maior.

	Acesso local	**Acesso à outro objeto**	**Acesso reflexivo**
Tempo(ms)	42	49	14.523

Tabela 4.3: Média do tempo de acesso a variáveis na máquina virtual

O terceiro experimento foi conduzido para obter o tempo de chamada a métodos de objetos externos. As chamadas utilizadas foram de dois tipos. Primeiramente foi utilizado passagem de um parâmetro e recebimento de um retorno, através da operação "`var = obj.metodo(var)`". Posteriormente foi realizado o teste sem passagem de parâmetros e sem retorno, através da operação "`obj.metodo()`". Os mesmos procedimentos foram adotados para as chamadas reflexivas, uma como passagem de parâmetro e retorno de valor, através da operação "`var = obj.invoke(var)`" e outra sem a passagem de parâmetro e sem retorno de valor, através da operação "`obj.invoke()`".

	Chamadas convencionais	**Chamadas reflexivas**
Sem parâmetros e retorno	6	89
Com parâmetros e retorno	7	112

Tabela 4.4: Média do tempo de chamadas à métodos na máquina virtual

A tabela 4.4 apresenta o tempo médio de chamada em cada um dos casos citados. As médias foram obtidas através de um algoritmo com 100 interações, sendo que em cada interação foram realizadas 500.000 chamadas ao mesmo método. As chamadas sem passagem de parâmetros e sem retorno de valores apresentam-se aproximadamente 14.8 vezes mais lentas se utilizada reflexão computacional, este valor aumenta se comparada as chamadas com passagem de parâmetros e retorno de valores, passando para algo em torno de 16 vezes mais lentas.

O gráfico apresentado na figura 4.9 resume os tempos de criação de objetos, acesso a variáveis e chamadas de métodos. Para criação de objetos, a reflexão computacional

mostrou o melhor desempenho dos testes, com aproximadamente 2.3 vezes mais lenta. Para a chamada de métodos, a sobrecarga adicionada foi maior, em torno de 16 vezes, devido às inúmeras operações que precisam ser realizadas dinamicamente, porém o pior resultado foi no acesso e modificação de variáveis, aonde o atraso chegou a ser de 296 vezes, demonstrando que este é um recurso que deve ser evitado ou sua implementação otimizada.

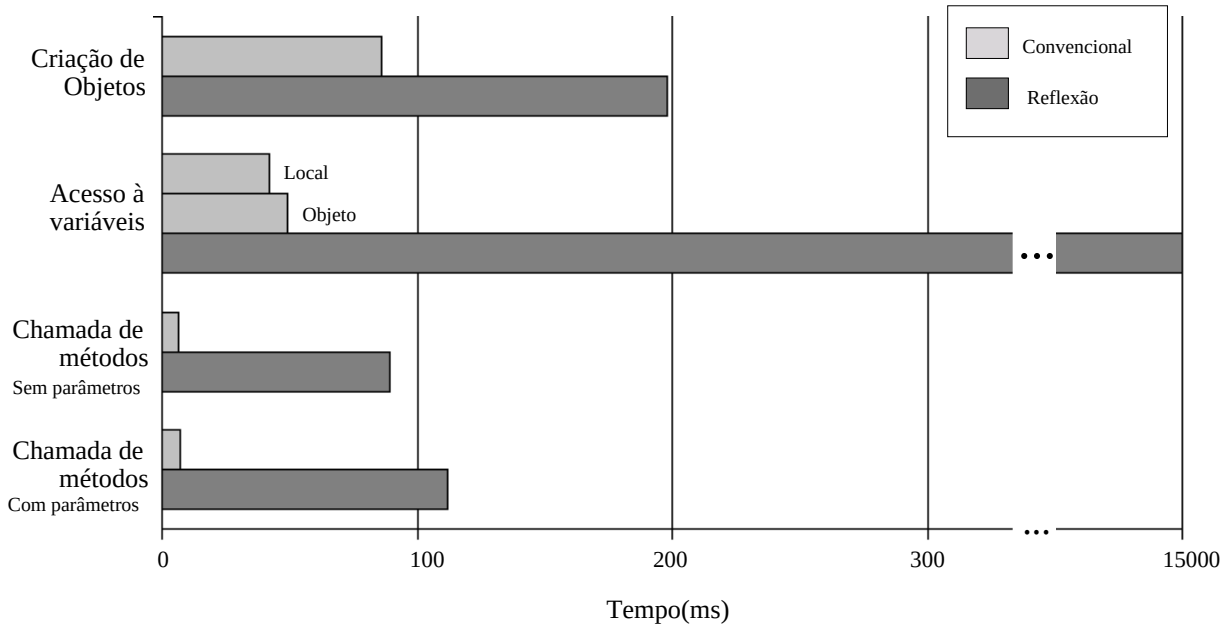


Figura 4.9: Tempos de criação de objetos, acesso a variáveis e chamadas à métodos

Através da análise dos valores apresentados e demonstrados graficamente na figura 4.9, é possível visualizar a carga adicional introduzida pela reflexão computacional. Porém cabe lembrar, que esta é uma função oferecida pela máquina virtual Java, que executa sobre um sistema operacional, estando sujeita as políticas implementadas por ele e as suas limitações, sendo assim, estes valores podem apresentar diferentes resultados dependo da arquitetura e do sistema hospedeiro. Deve-se ainda ressaltar o fato de que a máquina virtual Java está em constante aprimoramento, não sendo correto, considerar estes números como definitivos.

4.2.2 Tempos de carga de extensões

O tempo de carga de uma determinada extensão no sistema está relacionada a algumas operações descritas na sessão 4.1.3. Estas operações são responsáveis pela preparação do sistema para que este possa oferecer os serviços básicos descritos neste modelo.

As medições, aqui apresentadas são referentes a dois conjuntos de testes. Pri-

meiramente foi avaliado o tempo de carga das extensões durante a inicialização da aplicação e posteriormente o tempo de carga dinâmica, ou seja, com o sistema e a aplicação em execução. Em ambos os casos foram realizadas medições em relação a carga das extensões alocadas na parte fixa e na parte genérica da meta-lista.

Tempo de carga durante a inicialização

Durante a inicialização do sistema, as extensões podem ser carregadas através do uso de meta-informações, que são passadas pela aplicação no momento da sua inicialização. Existem dois locais onde estas extensões podem ser alocadas, na parte fixa da meta-lista ou na parte genérica. A parte genérica da meta-lista, abriga quaisquer tipos de extensões e não tem tamanho pré-definido, já a parte fixa, tem apenas 4 posições que podem ser ocupadas, sendo que estas estão reservadas para serviços básicos.

Número de Extensões	Tempo total	Tempo médio por extensão
0	18.10	—
1	19.98	19.98
2	20.70	10.35
3	20.66	6.89
4	22.24	5.56
5	22.41	4.48
6	22.63	3.77
7	22.88	3.27
8	23.40	2.92
9	23.68	2.63
10	23.94	2.39

Tabela 4.5: Média do tempo de carga de extensões na parte genérica da meta-lista durante a inicialização

A tabela 4.5 apresenta os tempos de inicialização de uma aplicação que solicita a carga de extensões na parte genérica da meta-lista. Quando nenhuma extensão é carregada, pode ser visualizado o tempo de inicialização do sistema com as características do modelo proposto. Esta inicialização apresenta uma considerável onerosidade em relação ao tempo, sendo que o número de extensões que são alocadas na parte genérica da meta-lista não promovem grande

aumento. A terceira coluna desta tabela contém a média do tempo de carga por extensão, ou seja, a estratégia que pode ser tomada para esconder a latência da carga, é alocar o máximo de extensões possíveis durante a inicialização da aplicação, ou deixar para realizar esta alocações dinamicamente.

A carga de extensões na parte fixa da meta-lista, assemelha-se bastante, em relação ao tempo necessário ao carregamento de extensões da parte genérica. A tabela 4.6 apresenta os resultados. Podemos perceber a mesma desvantagem em relação a sobrecarga inicial do sistema, com os procedimentos de criação do *proxy* de objetos, carga das interfaces, entre outras operações apresentadas na sessão 4.1.3, sendo que as extensões subseqüentes não acrescentam muito tempo. Através da média do tempo de carga por extensão apresentada na terceira coluna, pode-se tomar as mesmas medidas mencionadas para a carga de extensões na parte genérica da meta-lista com o objetivo de esconder a latência.

Número de extensões	**Tempo total**	**Tempo médio por extensão**
0	18.11	—
1	19.85	19.85
2	20.30	10.15
3	20.51	6.84
4	21.19	5.30

Tabela 4.6: Média do tempo de carga de extensões na parte fixa da meta-lista durante a inicialização

As médias apresentadas nas tabelas 4.5 e 4.6 foram obtidas através de um algoritmo que executou cada uma das operação 10.000 vezes. A figura 4.10 apresenta o gráfico do comportamento do tempo médio para cargas de extensões no sistema na parte genérica da meta-lista durante a inicialização da aplicação. No eixo “Y” está representado o tempo médio de carga por extensão(tabela 4.5 coluna 3), e no eixo “X” estão representados os números de extensões no sistema. Como mencionado anteriormente, é possível esconder parte da latência desta operação através do carregamento de múltiplas extensões ao mesmo tempo.

Tempo de carga com o sistema em execução

Após o processo de inicialização das funções básicas do sistema e a carga da aplicação, é possível, de forma dinâmica, solicitar o carregamento de extensões com serviços que

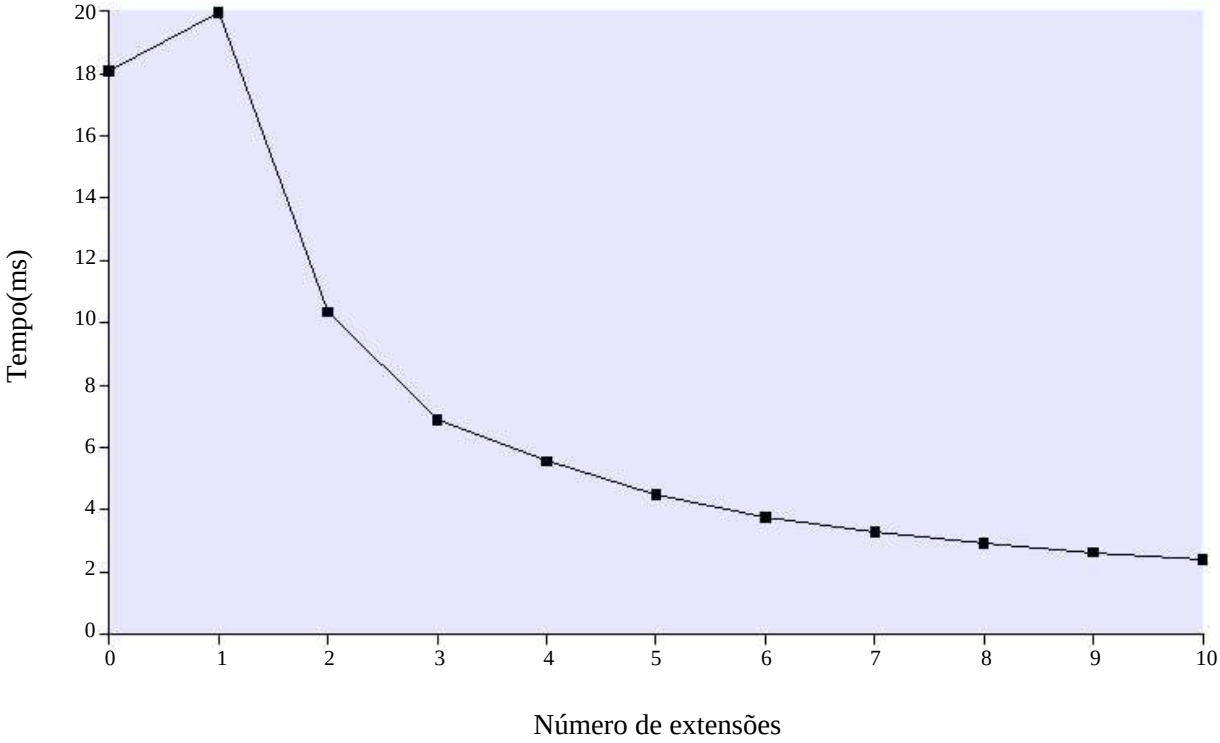


Figura 4.10: Média do tempo de carga por extensão durante a inicialização

complementem as suas funcionalidades. Esta operação é realizada através de um método reservado (este processo é descrito na sessão 4.1.1) que pode ser chamado a qualquer momento durante a execução da aplicação.

A tabela 4.7 apresenta os tempos médios para carga de extensões em tempo de execução, tanto na parte fixa como na parte genérica da meta-lista. Esta média foi calcula executando-se 10.000 cargas de cada um dos tipos descritos, sendo que a diferença dos tempos é praticamente irrisória, podemos considerar que ambos têm o mesmo impacto sobre o sistema.

	Parte fixa da meta-lista	**Parte genérica da meta-lista**
Tempo(ms)	0.4412	0.4414

Tabela 4.7: Média do tempo de carga de extensões dinamicamente

A opção de carga dinâmica (em tempo de execução) de extensões que complementam as funcionalidades do sistema mostrou-se a melhor opção para adaptar o ambiente de execução as necessidades específicas das aplicações. A sobrecarga adicionada ao sistema é menor do que a gerada se as extensões fossem carregadas em tempo de inicialização da aplicação. Outro

fato constatado é que o tempo médio para substituir uma extensão que está em execução por outra, é exatamente o mesmo tempo de carga de uma nova extensão.

4.2.3 Tempos de chamadas às extensões

O tempo de chamada às extensões, de acordo com o modelo proposto, é relativo à implementação específica de cada uma delas. Todas as extensões destinadas a serem alocadas na parte genérica da meta-lista, devem implementar duas funções básicas. Através de uma destas funções a extensão verificará se tem alguma correlação com o método que esta prestes a ser executado, desta forma, dependendo das operações que esta realizar, levará mais ou menos tempo. O objetivo da outra função é servir como um método de entrada, possibilitando a passagem da permissão de execução para a extensão.

Por outro lado, as extensões que ficam alocadas na parte fixa da meta-lista precisam implementar somente uma função. O meta-objeto `mGerenciador` conhece as características destas extensões, então a única função que estas necessitam implementar é a que passa o controle da execução para a extensão, possibilitando que esta realize as operações que esta programada para executar. A tabela 4.8 apresenta os resultados obtidos.

	Extensões parte fixa	**Extensões parte genérica**
Tempo(ms)	0.211	0.259

Tabela 4.8: Média do tempo de chamadas as extensões

As médias apresentadas na tabela 4.8 foram obtidas através da execução de 10.000 chamadas as extensões em cada um dos casos. Apesar da pequena diferença de tempo, esta se manteve praticamente constante, representando as operações a mais, que uma extensão alocada na parte genérica da meta-lista, precisa realizar para estar apta a receber o controle de execução do sistema e poder realizar suas operações. Neste caso específico, a operação realizada pela extensão para identificar se um método executado por um objeto no nível base, poderia requerer os seus serviços foi realizada através da comparação do nome de um método pré-definido com o que está executando.

4.2.4 Aplicação de exemplo

O exemplo implementado para demonstrar a extensibilidade do sistema foi o problema clássico do Produtor Consumidor. A aplicação foi implementada com um *buffer* de uma posição e sem nenhum controle sobre a produção ou consumo dos elementos, ou seja, podem ser produzidos mais elementos que o tamanho do *buffer*. Caso isto aconteça, a aplicação sobrescreve o seu conteúdo. Também podem ser consumidos elementos que ainda não formam produzidos, neste caso é retornando um valor qualquer.

A adaptabilidade do sistema foi testada através de extensões projetadas para este caso. Uma das extensões implementadas foi o rastreamento do sistema, a única responsabilidade desta extensão é gravar os métodos que estão sendo executados em um arquivo. Esta opção pode ser habilitada e desabilitada em tempo de execução, igualmente a todas as outras extensões do sistema. Outra extensão desenvolvida foi a de persistência. Neste caso específico a persistência foi voltada para apenas a variável *buffer*, ou seja, a sua função é armazenar o valor atual desta variável. A terceira extensão que foi implementada, é responsável pela sincronização do *buffer*, ou seja, somente será permitido produzir quando tiver espaço para o armazenamento e consumir quando tiver algum dado disponível.

Todas as chamadas a métodos originadas no nível base, são interceptadas pelo `mGerenciador` e analisadas. Neste caso específico, quando é identificada uma chamada de manipulação do *buffer*, o meta-objeto `mGerenciador` consultará a extensões alocadas no sistema verificando se alguma delas possui operações a serem realizadas. Uma das extensões alocadas pode ser a de sincronização, que tem sua função representada na figura 4.11. Esta figura não apresenta o código real, mas uma idéia geral das operações que devem ser realizadas, onde antes da execução do método original, o controle do sistema é passado ao meta-nível, onde é verificado o tamanho do *buffer*, e somente será permitida a adição no momento em que tiver espaço disponível. Uma vez permitida a adição de elementos, é invocado o método original no nível base, através da chamada `metodo.invoke(obj, argumentos)`. Após a execução do método se este não resultar em nenhuma exceção, o número de elementos no *buffer* é incrementado e liberado o acesso a outras chamadas que podem estar sendo feitas, para finalizar, o controle do sistema é devolvido ao nível base.

Da mesma forma que Produtor, as chamadas do Consumidor também são interceptadas ao tentar acessar o *buffer*. Através da passagem do controle ao meta-nível, é verificado se existem elementos que podem ser retirados do *buffer*, quanto tiver, é invocado o método de reti-

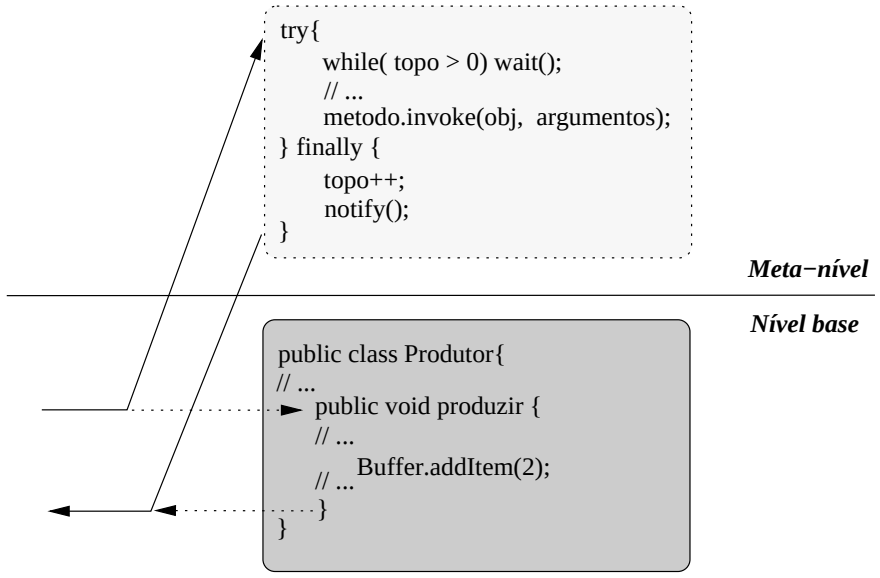


Figura 4.11: Exemplo do problema do Produtor Consumidor com a sincronização sendo feita no meta-nível do sistema.

rada no nível base, e atualizada as informações no meta-nível, para finalmente repassar o controle novamente ao nível base.

Com o objetivo de comparar o desempenho do sistema, foram realizados alguns testes, que tem seus valores apresentados na tabela 4.9. Basicamente o sistema foi testado em cinco etapas. Na primeira, somente a aplicação foi executada, sem nenhuma extensão, com o objetivo de prover um tempo base. Os valores apresentados não podem ser considerados como absolutos, pois o fluxo de execução do Produtor e do Consumidor, estão programados para interromper momentaneamente seu funcionamento por um período de 0 a 2 milisegundos a cada ação executada[10].

	Somente Aplicação	**Aplicação + Sincronização**	**Aplicação + Persistência**	**Aplicação + Rastreamento**	**Todas as Extensões**
Tempo (ms)	63.39	85.40	81.36	82.20	117.24
Aumento (%)	—	+ 36.29%	+ 29.92%	+ 31.25%	+ 86.52%

Tabela 4.9: Médias obtidas da aplicação exemplo simulada na máquina virtual

Na segunda, terceira e quarta etapa dos testes, o sistema foi executado com apenas uma extensão sendo alocada e utilizada, que foram respectivamente, a sincronização em seguida a persistência e por último o rastreamento. Na quinta etapa, foram alocadas todas as ex-

[10]Este tempo é resultado da utilização da função de geração de números aleatórios.

tensões ao mesmo tempo, produzindo o teste final. Os valores apresentados na tabela 4.9 são referentes a média de 10.000 testes para cada caso. Na última linha são apresentadas as diferenças de tempos. Em cada teste o Produtor foi programado para produzir 10 valores e o Consumidor foi programado para parar de consumir ao encontrar o último valor gerado pelo Produtor, quando a extensão de sincronização estava ativa, ele consumia 10 vezes, porém quando esta não estava ativa o número era incerto, mas nunca passou de 20 vezes, e também não menos de 4 vezes.

4.2.5 Conclusões sobre a implementação

Através da simulação, foi possível identificar que a melhor forma de alocar uma única extensão é em tempo de execução. O procedimento de carga durante a inicialização, demanda um pouco a mais de tempo para carregar poucas extensões, porém com uma maior quantidade mostrou uma melhor performance. A passagem de meta-informações através de variáveis reservadas pode ser útil no caso do desenvolvimento de um programa que realize uma verificação estática das necessidades da aplicação, podendo assim, adicionar o resultado ao código fonte, permitindo a adaptação do sistema a ela.

Outro resultado obtido durante a simulação do modelo, mostrou que as extensões da parte fixa da meta-lista tem um tempo de chamada menor, o que pode ser caracterizado pelo fato delas implementarem somente uma função, a que executa as operações, não havendo a necessidade de verificar a finalidade da extensão. Dependendo da implementação da função de reconhecimento pelas extensões da parte genérica da meta-lista pode-se obter um melhor desempenho também.

A linguagem de programação Java oferece uma vasta coleção de recursos para a utilização da reflexão computacional, seja ela estrutural ou comportamental, porém, não oferece um protocolo de comunicação de meta-objetos. Optou-se pela não utilização de um protocolo externo, para manter a compatibilidade do sistema com outras implementações, então, a forma encontrada para superar esta deficiência foi através da implementação de um *proxy* de objetos global, ou seja, o próprio meta-objeto `mGerenciador` que é o único meta-objeto compartilhado por todos os meta-espaços. Para o funcionamento apropriado, no momento da alocação dos objetos no nível base, eles devem ser ligados com o `mGerenciador` que terá como uma de suas funções a interceptação e analise das chamadas de métodos destes objetos.

Na aplicação de exemplo implementada e testada, ficou caracterizada a adaptabilidade do sistema como um todo. Apesar de poucas extensões, foi possível complementar e tornar o ambiente de execução mais apropriado para o objetivo em questão, como por exem-

plo, adicionando um meta-objeto de persistência extremamente específico para um caso de uso e mesmo assim, mantendo-o de forma transparente a aplicação. O mesmo resultado foi obtido com as demais extensões utilizadas. As médias de tempo encontradas servem para demonstrar que é possível atingir a extensibilidade do sistema, provendo recursos extras sem adicionar uma grande sobrecarga excessiva aos tempos de execução.

Capítulo 5

Conclusões

O modelo de um núcleo extensível apresentado neste trabalho oferece a possibilidade da modificação do ambiente de execução, deixando-o com as características específicas de cada aplicação, provendo somente os requisitos solicitados. Isto foi possível, pois o ambiente é dividido em meta-espaços, onde cada aplicação instanciada é anexada a um meta-espaço próprio, que oferece os recursos necessários, desta forma as aplicações enxergam somente os serviços disponíveis no seu meta-espaço e executam como se estivessem sozinhas no sistema. A especialização dos meta-espaços pode ser alcançada através da utilização de meta-informações que são passadas pela aplicação durante a sua inicialização ou em tempo de execução.

A passagem de meta-informações durante a inicialização da aplicação ocorre através de variáveis reservadas, que apresentam características estáticas, já a passagem em tempo de execução é realizada dinamicamente através de métodos reservados. Estas duas formas mostraram-se eficientes em relação ao tempo gasto pela operação, sendo que ainda permitem que a aplicação escolha o momento mais adequado para solicitar modificações no sistema.

As extensões do sistema devem implementar características não funcionais das aplicações, como por exemplo, segurança e rastreamento, ou ainda, serviços do sistema operacional, como persistência, escalonamento, entre outros. Existem dois tipos possíveis de extensões, as pré-definidas, que ficam armazenadas na parte fixa da meta-lista e as gerais, que ficam alocadas na parte genérica da meta-lista. As extensões pré-definidas devem implementar apenas uma função básica, que é através da qual é passado o controle de execução do sistema para ela. As extensões genéricas, devem implementar duas funções básicas, além da que passa o controle da execução, deve ser implementada também uma que realiza a verificação de quais chamadas interceptadas no sistema são referentes aos serviços oferecidas por ela, pois é de responsabilidade da extensão

identificar quando ela é necessária.

No modelo proposto, de acordo com a taxonomia adotada e descrita na sessão 3.2, é apresentada uma extensibilidade procedural, pois as operações implementadas pelas extensões dizem respeito a métodos, ou seja, podem ser executadas operações antes ou depois de um método, e ainda, o método original pode ser impedido de executar, ou trocado por outro. No momento da alocação a granularidade pode ser considerada como modular, pois não é possível a carga de somente um método, necessitando que a classe inteira da extensão seja alocada e instanciada, mesmo que esta não seja utilizada por completo.

A utilização conjunta dos paradigmas de orientação a objetos e reflexão computacional amparadas pelos conceitos da técnica de separação de interesses, revelou-se uma boa estratégia. O resultado final foi um modelo de um núcleo com a forma de identificação, proteção, atomicidade e sincronização da orientação a objetos, a separação das características não funcionais das aplicações que foram distribuídas na forma de extensões, seguindo os conceitos da separação de interesses, e o acesso às informações do sistema, bem como a sua modificação tanto estrutural como comportamental que foi provida pela reflexão computacional.

Como trabalhos futuros, é possível ressaltar a necessidade da inclusão do modelo proposto em algum dos sistemas operacionais baseados em Java existentes, com o objetivo de obter resultados que expressem maior proximidade com um ambiente real em execução. Alguns meta-objetos como o `mEscalonador`, `mPersistência`, `mCarregador`, entre outros, podem se tornar objetos de estudo, sendo aprimorados ou implementados com características diferentes das apresentadas neste trabalho.

Referências Bibliográficas

[AND 93] ANDERSON, T. E. **The Case For Application-Specific Operating Systems**. EECS Department, University of California, Berkeley, 1993. Relatório Técnico UCB/CSD-93-738.

[BAB 81] BABAOGLU, O.; JOY, W. **Converting A Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits**. In: PROCEEDINGS OF THE 8TH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP), 1981. v.15, p.78–86.

[BAC 98] BACK, G. et al. **Java Operating Systems: Design and Implementation**. jun, 1998. 15 p. Relatório Técnico UUCS-98-015.

[BAC 00] BACK, G.; HSIEH, W. C.; LEPREAU, J. **Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java**. In: OSDI, 2000. p.333–346.

[BER 94] BERSHAD, B. N.; CHAMBERS, C.; S. EGGERS, E. A. **SPIN: An Extensible Microkernel for Application-specific Operating System Services**. In: IN PROCEEDINGS OF THE 6TH ACM SIGOPS WORKSHOP ON MATCHING OPERATING SYSTEMS TO APPLICATION'S NEEDS, 1994. Warden, Germany.

[BER 95] BERSHAD, B. N. et al. **Extensibility, Safety and Performance in the SPIN Operating System**. In: 15TH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1995. p.267–284.

[BER 98] BERBERS, Y. **Handling adaptive behavior in real-time systems**. Technologies for the Information Society, v.Developments and Opportunities (Roger, J.-Y. and Stanford-Smith, B. and Kidd, P.T., eds.), p.851–857, 1998.

[BEU 97] BEUCHE, D. **An Approach for Managing Highly Configurable Operating Systems**. In: Bosch, J.; Mitchell, S., editors, OBJECT-ORIENTED TECHNOLOGY: ECOOP'97 WORKSHOP READER, 1997. v.1357 of **Lecture Notes in Computer Science**, p.531–536. Workshop on Object-Orientation and Operating Systems.

[BLU 02] BLUNDEN, B. **Virtual machine design and implementation in C/C++**. Plano, TX, USA: Wordware Publishing, 2002. xvii + 668 p.

[BOO 04] BOOCH, G. **Object-Oriented Analysis and Design with Applications (3rd Edition)**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[BOU 02] BOURAQADI, N.; LEDOUX, T. **Aspect-Oriented Programming Using Reflection**. Ecole des Mines de Douai, Outubro, 2002. Relatório Técnico 2002-10-3.

[BOY 02] BOYAPATI, C. **Towards an extensible virtual machine**. MIT Laboratory for Computer Science, Cambridge, MA, 2002. Relatório técnico .

[CAH 96] CAHILL, V. **Flexibility in Object-Oriented Operating Systems: A Review**. 1996. Relatório Técnico TCD-CS-96-05.

[CAM 93] CAMPBELL, R. H.; MADANY, P. **The design of an object-oriented operating system: a case study of choices**. OOPS Messenger, v.4, n.2, p.226, 1993.

[CAM 95] CAMPBELL, R.; TAN, S. **Choices: An ObjectOriented Multimedia Operating System**. In: FIFTH WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS (HOTOS V), 1995.

[CAM 96] CAMPBELL, R. H. et al. **Designing and Implementing Choices: An Object-Oriented System in C++**. Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor), v.36, n.9, p.117–126, 1996.

[CHE 95] CHEUNG, W. H.; LOONG, A. H. S. **Exploring issues of operating systems structuring: from microkernel to extensible systems**. ACM SIGOPS Operating Systems Review, v.29, n.4, p.4–16, October, 1995.

[CHE 01] CHEN, P. M.; NOBLE, B. D. **When Virtual is Better than Real**. In: HOTOS, 2001. p.133–138.

[CLA 00] CLARKE, M. **Operating System Support for Emerging Application Domains**. Computing Department; Lancaster University, 2000. Tese de Doutorado.

[COL 05] COLE, L.; BORBA, P. **Deriving refactorings for AspectJ**. In: AOSD ’05: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2005. p.123–134.

[COW 96] COWAN, C. et al. **Fast concurrent dynamic linking for an adaptive operating system**. In: IN PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON CONFIGURABLE DISTRIBUTED SYSTEMS, 1996. p.108–15.

[DEA 00] DEARLE, A.; HULSE, D. **Operating system support for persistent systems: past, present and future**. Softw, Pract. Exper, v.30, n.4, p.295–324, 2000.

[DEN 02] DENYS, G.; PIESSENS, F.; MATTHIJS, F. **A Survey of Customizability in Operating Systems Research**. ACM Computing Surveys, v.34, n.4, p.450–468, December, 2002.

[DIJ 76] DIJKSTRA, E. W. **A Discipline of Programming**. Prentice-Hall, 1976.

[dO 01] DE OLIVEIRA, R. S.; CARISSIMI, A.; TOSCANI, S. S. **Sistemas Operacionais**. RITA, v.8, n.3, p.7–39, 2001.

[DOW 01] DOWLING, J.; CAHILL, V. **The K-Component Architecture Meta-Model for Self-Adaptive Software**. In: Yonezawa, A.; Matsuoka, S., editors, THIRD INTERNATIONAL CONFERENCE ON METALEVEL ARCHITECTURES AND SEPARATION OF CROSSCUTTING CONCERNS, 2001. v.2192 of **Lecture Notes in Computer Science**, p.81–88. ISSN: 0302-9743.

[ELR 01] ELRAD et al. **Discussing Aspects of AOP**. CACM: Communications of the ACM, v.44, 2001.

[ENG 95] ENGLER, D. R.; KAASHOEK, M. F.; JR, J. O. **Exokernel: an operating system architecture for application-specif ic resource management**. In: IN PROCEEDINGS OF THE FIFTEENTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1995.

[ENG 98] ENGLER, D. R. **The Exokernel Operating System Architecture**. Massachusetts Institute of Technology, October, 1998. Phd thesis.

[FER 89] FERBER, J. **Computational Reflection in Class Based Object-Oriented Languages**. OOPSLA. 1989, v.24, n.10, p.317–326, October, 1989.

[FIG 03] FIGUEIREDO, R.; DINDA, P. A.; FORTES, J. A. B. **A Case for Grid Computing on Virtual Machines**. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS (ICDCS), 2003.

[FOO 93] FOOTE, B. **Architectural Balkanization in the Post-Linguistic Area**. In: OOPSLA ’93 WORKSHOP ON REFLECTION AND METALEVEL ARCHITECTURES IN OBJECT-ORIENTED PROGRAMMING, 1993.

[FRO 01] FROEHLICH, A. A. M. **Application-Oriented Operating Systems**. 1. ed. GMD - Forschungszentrum Informationstechnik, 2001.

[GIV 06] GIVARGIS, T. **Zero cost indexing for improved processor cache performance**. ACM Transactions on Design Automation of Electronic Systems, v.11, n.1, p.3–25, Janeiro, 2006.

[GOL 98] GOLM, M. **metaXa and the Future of Reflection**. Erlangen , Germany: University of Erlangen-Nürnberg, Novembro, 1998. Relatório Técnico D-91058.

[GOS 95] GOSLING, J. **Java Intermediate Bytecodes**. In: PROC. ACM SIGPLAN WORKSHOP ON INTERMEDIATE REPRESENTATIONS, 1995. v.30:3 of **ACM Sigplan Notices**, p.111–118.

[GOS 96] GOSLING, J.; JOY, B.; STEELE, G. **The Java Language Specification**. The Java Series. Reading, MA: Addison-Wesley, 1996.

[GUR 05] GURP, J. V.; BRINKKEMPER, S.; BOSCH, J. **Design preservation over subsequent releases of a software product: a case study of Baan ERP**. In: JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE, 2005. v.17, p.277306.

[HAN 05] HANNEMANN, J.; MURPHY, G.; KICZALES, G. **Role-Based Refactoring of Crosscutting Concerns**. In: Tarr, P., editor, PROC. 4RD INT' CONF. ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD-2005), 2005. p.135–146.

[HAR 99] HARRIS, T. **An Extensible Virtual Machine Architecture**. In: OOPSLA'99 WORKSHOP ON SIMPLICITY, PERFORMANCE AND PORTABILITY IN VIRTUAL MACHINE DESIGN, 1999.

[HAR 01] HARRIS, T. L. **Extensible Virtual Machines**. Computer Laboratory, University of Cambridge, 2001. Tese de Doutorado.

[HER 88] HERRMANN, F. E. A. **Chorus distributed operating system**. Computing Systems, v.1, n.4, p.305–367, 1988.

[HMD 01] HARVEY M. DEITEL, PAUL J. DEITEL, H. M. D. **Advanced Java 2 Platform: How to Program**. Prentice Hall, 2001.

[HOR 97] HORIE, M. et al. **Designing Meta-Interfaces for Object-Oriented Operating Systems**. In: IEEE PACIFIC RIM CONF. COMMUNICATIONS, COMPUTERS, AND SIGNAL PROCESSING, 1997. p.989–992.

[HOR 98] HORIE, M. et al. **Using Meta-Interfaces to Support Secure Dynamic System Reconfiguration**. In: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON CONFIGURABLE DISTRIBUTED SYSTEMS (ICCDS), 1998.

[HOR 99] HORIE, M. **On secure, dynamic customizing of a meta-space-based operating system**. National Library of Canada = Bibliothèque nationale du Canada, 1999. Tese de Doutorado.

[HäR 97] HäRTIG, H. et al. **The Performance of μ-Kernel-Based Systems**. In: 16TH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP'97), 1997.

[JAN 03] JANZEN, D.; VOLDER, K. D. **Navigating and querying code without getting lost**. In: AOSD '03: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2003. p.178–187.

[KEA 89] KEARNS, J. P.; DEFAZIO, S. **Diversity in Database Reference Behavior**. In: PROCEEDINGS OF THE ACM SIGMETRICS INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 1989. v.17(1) of **ACM SIGMETRICS Performance Evaluation Review**, p.11–19.

[KHA 92] KHANNA, S. et al. **Realtime scheduling in SunOS 5.0**. In: IN PROCEEDINGS OF THE WINTER 1992 USENIX CONFERENCE, 1992. p.pages 375–390.

[KIC 93] KICZALES, G. et al. **The need for customizable operating systems**. In: IEEE, editor, PROCEEDINGS. FOURTH WORKSHOP ON WORKSTATION OPERATING SYSTEMS, NAPA, CA, USA, OCTOBER 14–15, 1993, 1993. p.165–169.

[KIC 97] KICZALES, G. et al. **Aspect-Oriented Programming**. Xerox PARC, Fevereiro, 1997. Relatório Técnico SPL97-008 P9710042.

[KIC 01] KICZALES, G. et al. **An Overview of AspectJ**. In: OBJECT-ORIENTED PROGRAMMING 15TH EUROPEAN CONFERENCE, v.2072 of **Lecture Notes in Computer Science**, p.327–353. Springer-Verlag, Budapest, Hungary, Junho, 2001.

[KIC 05] KICZALES, G.; MEZINI, M. **Aspect-oriented programming and modular reasoning**. In: ICSE '05: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2005. p.49–58.

[KLE 98] KLEINOEDER, J.; GOLM, M. **MetaJava — A Platform for Adaptable Operating-System Mechanisms**. Lecture Notes in Computer Science, v.1357, p.507–517, 1998.

[KRI 93] KRIEGER, O.; STUMM, M. **HFS: A Flexible File System for large-scale Multiprocessors**. In: PROCEEDINGS OF THE 1993 DAGS/PC SYMPOSIUM, 1993.

[LAD 03] LADDAD, R. **AspectJ in Action: Practical Aspect-Oriented Programming**. Manning, 2003.

[LIN 96] LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. Addison-Wesley, 1996.

[MAC 02] MACHADO, F.; MAIA, L. **Arquitetura de sistemas operacionais**. LTC Livros Técnicos e Científicos Editora S.A., 2002.

[MAE 87] MAES, P. **Concepts and Experiments in Computational Reflection**. In: ACM SIGPLAN NOTICES, 1987. v.22(12), p.147–155.

[MCK 04] MCKINLEY, P. K. et al. **A Taxonomy of Compositional Adaptation**. Department of Computer Science and Engineering Michigan State University, May, 2004. Relatório Técnico MSU-CSE-04-17.

[MON 94] MONTZ, A. B. et al. **Scout: A Communications-Oriented Operating System (Abstract)**. In: OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 1994. p.200.

[MON 05] MONTEIRO, M. P.; FERNANDES, J. M. **Towards a catalog of aspect-oriented refactorings**. In: AOSD ’05: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2005. p.111–122.

[NEL 91] NELSON, G. **Systems Programming with Modula-3**. Prentice Hall, 1991.

[OGE 05] OGEL, F.; THOMAS, G.; FOLLIOT, B. **Supporting efficient dynamic aspects through reflection and dynamic compilation**. In: SAC ’05: PROCEEDINGS OF THE 2005 ACM SYMPOSIUM ON APPLIED COMPUTING, 2005. p.1351–1356.

[OLI 99] OLIVIA, A.; BUZATO, L. E. **The Design and Implementation of Guaraná**. In: PROCEEDINGS OF THE 5TH USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS (COOTS’99), 1999. p.203–216. http://www.dcc.unicamp.br/∼oliva/guarana/docs/desimpl.ps.gz.

[OUS 85] OUSTERHOUT, J. K. et al. **A Trace-Driven Analysis of the UNIX 4.2 BSD File System**. In: IN PROCEEDINGS OF THE TENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1985. p.15–24.

[POL 05] POLPETA, F. V.; FRöHLICH, A. A. **On the Automatic Generation of SoC-based Embedded Systems**. In: IN: PROCEEDINGS OF THE 10TH IEEE INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION, 2005.

[PRA 04] PRATIKAKIS, P.; SPACCO, J.; HICKS, M. **Transparent proxies for java futures**. In: OOPSLA '04: PROCEEDINGS OF THE 19TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2004. p.206–223.

[REN 00] RENAUD, K.; EVANS, H. **JavaCloak: Engineering Java Proxy Objects using Reflection.** In: NET.OBJECTDAYS, 2000.

[ROS 04] ROSENBLUM, M. **The Reincarnation of Virtual Machines**. ACM Queue, v.2, n.5, p.34–40, 2004.

[RUM 97] RUMBAUGH, J. E. **Models Through the Development Process**. JOOP, v.10, n.2, p.5–8, 14, 1997.

[SCH 01] SCHWABER, K.; BEEDLE, M. **Agile Software Development with Scrum**. 1. ed. Alan R. Apt, 2001.

[SEL 94] SELTZER, M. et al. **An introduction to the architecture of the VINO kernel**. EECS Department, Harvard University, 1994. Relatório Técnico TR-34-94.

[SEL 95] SELTZER, M.; SMALL, C.; SMITH, K. **The Case for Extensible Operating Systems**. Harvard University Center for Research in Computing Technology, 1995. Relatório Técnico TR16-95.

[SEL 97a] SELTZER, M. et al. **Issues in extensible operating systems**. Harvard University Center for Research in Computing Technology, 1997. Relatório Técnico TR-18-97.

[SEL 97b] SELTZER, M. I.; SMALL, C. **Self-monitoring and self-adapting operating systems**. Proceedings of the Sixth workshop on Hot Topics in Operating Systems, v.16, 1997.

[SMA 96] SMALL, C.; SELTZER, M. I. **A Comparison of OS Extension Technologies**. In: USENIX ANNUAL TECHNICAL CONFERENCE, 1996. p.41–54.

[SMA 98] SMALL, C. **Building an Extensible Operating System**. Harvard University, Division of Engineering and Applied Sciences, Octobe, 1998. Ph.d. thesis.

[SOL 95] Soley, R., editor. **Object Management Architecture Guide: Revision 3.0**. Frameington, MA: Object Management Group, Junho, 1995.

[STA 98] STANKOVIC, J. **Admission Control, Reservations and Reflection in Operating Systems**. In: IEEE BULLETIN OF THE TECHNICAL COMMITTEE ON OPERATING SYSTEMS AND APPLICATION ENVIRONMENTS (TCOS), 1998. v.10.

[STE 94] STEEL, L. **Beyond objects. European Conference on Object-Oriented Programming**. Lecture Notes in Computer Science, v.1, 1994.

[STE 06] STELLMAN, A.; GREENE, J. **Applied software project management**. O'Reilly, 2006.

[STO 81] STONEBRAKER, M. **Operating system support for database management**. Communications of the ACM, v.24, n.7, p.412–418, 1981.

[TAN 87] TANENBAUM, A. **Operating Systems: Design and Implementation**. Engelwood Cliffs, NJ: Prentice Hall, 1987.

[TAN 92] TANNENBAUM, A. S. **Modern Operating Systems**. Englewood Cliffs, NJ: Prentice–Hall, 1992.

[TAN 01] TANTER, E.; BOURAQADI, N.; NOYÉ, J. **Reflex – Towards an Open Reflective Extension of Java**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON METALEVEL ARCHITECTURES AND ADVANCED SEPARATION OF CONCERNS (REFLECTION 2001), 2001. v.2192 of **Lecture Notes in Computer Science**, p.25–43.

[WAH 93] WAHBE, R. et al. **Efficient Software-Based Fault Isolation**. ACM SIGOPS Operating Systems Review, v.27, n.5, p.203–216, December, 1993.

[WIE 05] WIEGERS, K. E. **More About Software Requirements: Thorny Issues and Practical Advice**. Microsoft Press, 2005.

[WU 98] WU, Z. **Reflective Java and A Reflective Component-Based Transaction Architecture**. In: PROCEEDINGS OF THE ACM OOPSLA WORKSHOP ON REFLEXIVE PROGRAMMING IN JAVA AND C++, 1998.

[YOK 92] YOKOTE, Y. **The Apertos Reflective Operating System: The Concept and its Implementation**. Proc. OOPSLA '92 ACM, v.1, p.414–434, 1992.

[YOK 93] YOKOTE, Y. **Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach**. In: IN PROCEEDINGS OF THE 1 ST INTERNATIONAL SYMPOSIUM ON OBJECT TECHNOLOGIES FOR ADVANCED SOFTWARE, 1993. p.145–162.

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

[ZAN 94] ZANCANELLA, L. C.; NAVAUX, P. O. A. **Object-Oriented Operating Systems: The Aurora Approach**. Conferência Chilena, v.1, p.271–280, 1994.

[ZAN 95a] ZANCANELLA, L. C.; NAVAUX, P. O. A. **Flexible Kernel: The AURORA approach for Multiprocessor Operating System**. Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'95), v.1, 1995.

[ZAN 95b] ZANCANELLA, L. C.; NAVAUX, P. O. A. **A Reflective Multiprocessor Object-Oriented Operating Systems**. 4th Nordic Transputer Conference on Parallel Computing and Transputers, v.4, p.421–427, 1995.

[ZAN 97] ZANCANELLA, L. C. **Estrutura Reflexiva para sistemas Operacionais Multiprogramados**. Tese de Doutorado, CPGCC da UFRGS, 1997. Tese de Doutorado.

[ZIM 96] ZIMMERMANN, C. **Metalevels, MOPs and what the Fuzz is all about**. In: ADVANCES IN OBJECT-ORIENTED METALEVEL ARCHITECTURES AND REFLECTION. CRC Press, 1996.