UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Luiz Henrique Shigunov

ModulOS: um sistema operacional modular baseado em interfaces

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

Orientador: Thadeu Botteri Corso

Florianópolis, fevereiro de 2003

MODULOS: UM SISTEMA OPERACIONAL MODULAR BASEADO EM INTERFACES

Luiz Henrique Shigunov

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

____

Fernando Alvaro Ostuni Gauthier

Banca Examinadora

____

Thadeu Botteri Corso

____

Luiz Fernando Friedrich

____

José Mazzucco Jr.

Sumário

1 Introdução **8**

1.1 Objetivo geral 9

1.2 Objetivos específicos 9

2 Sistemas monolíticos **11**

2.1 Introdução 11

2.2 Iniciação 12

2.2.1 Primeiro estágio 13

2.2.2 Segundo estágio 13

2.2.3 Terceiro estágio 15

2.3 Finalização 15

2.4 Desenvolvimento do sistema 16

2.4.1 Atualização e novas funções 16

2.4.2 Drivers de dispositivos 17

2.5 Estudo de caso: Linux 17

2.6 Conclusões 21

3 Sistemas micronúcleo **22**

3.1 Introdução 22

3.2 Caracterização do QNX 23

3.3 Iniciação 26

3.4 Finalização 29

3.5 Desenvolvimento do sistema 29

3.6 Estudo de caso: HURD 30

3.7 Conclusões 31

4 Sistemas modulares **32**
4.1 Introdução 32
4.2 Iniciação 33
4.2.1 Arquivo de log da iniciação 36
4.2.2 Modo de segurança 37
4.3 Finalização 38
4.4 Desenvolvimento do sistema 39
4.5 Estudo de caso: Solaris 40
4.6 Conclusões 43

5 ModulOS **45**
5.1 Introdução 45
5.2 Iniciação 47
5.3 Finalização 49
5.4 Desenvolvimento do sistema 50
5.4.1 Interfaces 51
5.4.2 Módulos do sistema 54
5.4.3 Módulos do usuário 56
5.5 Ferramentas criadas para o projeto 59
5.5.1 Geradores de arquivos de módulos 59
5.6 Estado atual do desenvolvimento 61
5.6.1 Administrador de memória 61
5.6.2 Administrador de processos 63
5.6.3 Administrador de arquivos 64
5.6.4 Administrador de módulos do usuário 65
5.7 Trabalhos futuros 67

6 Conclusões **69**

Lista de Figuras

3.1 Arquitetura do QNX 24
3.2 Estados de um fluxo de controle usando passagem de mensagens 25

5.1 Visão geral do ModulOS 48
5.2 Espaço de endereços 62
5.3 Administrador de arquivos 65
5.4 Chamada do sistema 66

Resumo

Existem diversos modelos de sistemas operacionais. Um dos primeiros modelos usados foi o do núcleo monolítico. Nesse modelo todas as funcionalidades estão contidas num único arquivo binário chamado de núcleo.

Por muitos anos esse modelo foi adequado, porém com o aumento das funcionalidades e serviços implementados o núcleo monolítico começou a apresentar problemas de complexidade tornando a sua manutenção difícil.

Para tentar solucionar esse problema várias alternativas foram propostas.

Uma das alternativas são os sistemas micronúcleo que reduziram as funcionalidades do núcleo do sistema ao mínimo e as implementaram utilizando programas convencionais que colaboram entre si.

No entanto, essa alternativa trouxe uma perda de desempenho muito grande que fez com que esses sistemas não fossem usados largamente.

Outra alternativa aos sistemas monolíticos são os sistemas modulares que se mostram muito adequados ao ambiente computacional atual de rápidas mudanças.

Nesses sistemas operacionais as diversas funcionalidades do sistema são divididas em módulos que cooperam entre si para formar o sistema.

O sistema operacional apresentado neste trabalho é um sistema modular que se utiliza de um sistema de interfaces para permitir a colaboração entre os diversos módulos.

Cada módulo deve implementar uma interface conhecida e é através dessas interfaces que os módulos se utilizam dos serviços oferecidos por um módulo.

Abstract

There are many different types of operating systems. One of the first used was the monolithic kernel. In this design, all functionality is kept inside one binary file called the kernel.

For many years this design was adequated, but with the incresing of functionalities and services implemented, the monolithic kernel started to suffer of complexity problems making it's maintenance difficult.

Trying to solve this problem different designs were made.

Microkernel operating systems are one of this different designs. The microkernel implements only the minimal necessary functionalities. Everything else is implemented using conventional programs that collaborate with each other.

However, this alternative has bring a big loss of performance which is why this type of operating system is not so commonly used.

Another alternative to monolithic kernels are the modular operating systems which have demonstrated to be very adequated for today computing environment of very fast changes.

In this type of operating system the many system's functionalities are divided in modules that collaborate with each other to make the whole system possible.

The operating system discussed in this work it's a modular operating system which uses a set of interfaces to make the collaboration of each module easier.

Every module of this system must implement at least one known interface and is through these interfaces that modules use all services offered by a module.

Capítulo 1

Introdução

No começo, o desenvolvimento de sistemas operacionais era semelhante ao desenvolvimento de um grande programa. Tudo o que o sistema operacional precisava para funcionar e também o que oferecia para os programas estava contido no núcleo do sistema. Depois de compilado, esse núcleo se transformava num único arquivo binário, motivo pelo qual passou a ser denominado núcleo monolítico.

Nos anos de 1960 esse modelo não era de todo ruim já que não existiam muitos dispositivos a serem suportados nem muitos serviços a serem oferecidos, enfim um sistema operacional era muito mais simples que atualmente.

Porém, a medida que mais dispositivos foram sendo suportados e mais serviços oferecidos, o núcleo do sistema começou a ficar grande. Isso trouxe dificuldades para o desenvolvimento de novas funcionalidades e também para a manutenção do sistema, pois um amplo conhecimento do sistema era necessário.

Com os problemas dos sistemas de núcleo monolítico crescendo, novos modelos começaram a aparecer. Os sistemas com micronúcleo tinham um modelo muito bom que permitia facilmente expandir e manter o sistema. No entanto, sofriam com um problema sério: a falta de um bom desempenho.

Esse problema fez com que os sistemas com micronúcleo fossem por muito tempo usados apenas no meio acadêmico e até hoje existem poucos sistemas comerciais usando o modelo com micronúcleo.

Outro modelo que surgiu foi o modular que permitia que certas partes do sistema fossem carregadas para a memória somente quando necessário e, principalmente, permitia um desenvolvimento mais focado, ou seja, uma pessoa não precisava conhecer todo o

sistema para desenvolver um módulo. Esse modelo permitia expandir e manter o sistema facilmente, mas sem o problema da falta de desempenho.

Mesmo sem conhecer os diversos modelos de sistemas operacionais, desde o início o modelo modular me atraiu e foi adotado para o desenvolvimento do ModulOS.

No início, o desenvolvimento do sistema foi muito lento já que eu tinha que aprender tudo e também porque eu não sabia direito o que e como fazer.

Porém, com o passar dos anos fui adquirindo conhecimento nos livros e na internet que somado à experiência já obtida fez com que o desenvolvimento do sistema ficasse mais fácil e rápido. Isso tornou possível que novas funcionalidades fossem acrescentadas rapidamente.

O ModulOS não está completo, mas já tem muitos serviços encontrados nos sistemas operacionais modernos.

1.1 Objetivo geral

Desenvolver um sistema operacional de propósito geral que suporte conceitos modernos como processos com várias linhas de execução, gerenciamento de memória com paginação, entre outros e que ao mesmo tempo seja extremamente modular permitindo que seja facilmente personalizado para cada situação e ambiente.

1.2 Objetivos específicos

O desenvolvimento de um sistema operacional de propósito geral compreende muita coisa que infelizmente é impossível apenas uma pessoa fazer num tempo tão curto. Por isso, ao menos os seguintes objetivos serão alcançados:

- Desenvolver um mecanismo que permita o sistema iniciar;
- Suportar todos os dispositivos de hardware necessários para o sistema iniciar através do disquete num computador padrão;
- Suportar pelo menos um sistema de arquivos para permitir que os módulos sejam lidos do disco quando necessários;
- Suportar pelo menos o teclado padrão para permitir que o usuário entre com dados;

- Suportar pelo menos o vídeo padrão na arquitetura PC/AT permitindo que dados sejam mostrados ao usuário;
- Desenvolver todos os mecanismos necessários para que programas possam ser executados pelo sistema;
- Desenvolver um simples interpretador de comandos para que seja possível verificar que o sistema funciona.

Capítulo 2

Sistemas monolíticos

2.1 Introdução

Um sistema operacional monolítico se caracteriza por ter todos os seus subsistemas (gerenciador de memória, sistema de arquivos, etc.) num único arquivo binário denominado de núcleo do sistema (kernel em inglês). Este modelo de sistema ficou muito popular e, hoje em dia, os sistemas mais conhecidos e utilizados são os diversos sistemas derivados do Unix original que data de 1969 [Vaha 96].

Nesse tipo de sistema operacional todas as partes estão fortemente inter-relacionadas, sendo, portanto, difícil substituir ou mudar uma dessas partes por outra com uma outra abordagem, por exemplo. Um bom exemplo foi o caso da mudança do gerenciador de memória no sistema Linux. A alteração trouxe instabilidade e diversos problemas em várias versões seguintes à mudança [Riel 02].

Apesar desses problemas, os sistemas monolíticos apresentam uma vantagem muito importante, a velocidade. Esse modelo de sistema é o mais veloz, principalmente porque as chamadas do sistema e a as chamadas entre os diversos subsistemas são feitas diretamente, não existindo passagem de mensagens como nos sistemas micronúcleo.

Por possuírem essa característica é que os sistemas monolíticos foram e ainda são muito utilizados. No passado, eram muito utilizados porque os computadores não eram rápidos o suficiente para executar sistemas micronúcleo satisfatoriamente, já hoje em dia costumam ser muito utilizados pela grande popularidade dos sistemas Unix, principalmente o Linux.

Atualmente, existem alguns sistemas operacionais que, apesar de serem monolíticos,

começam a utilizar o conceito de módulo. Nesses sistemas os módulos são utilizados para manter na memória apenas as partes do núcleo do sistema (*drivers* de dispositivos, sistemas de arquivos suportados, etc.) que estão sendo utilizadas [Maxw 99].

Entretanto, estes módulos são diferentes dos módulos discutidos no capítulo 4. Esses *drivers* de dispositivos ou sistemas de arquivos, ficam em arquivos binários denominados de módulos do núcleo (kernel modules em inglês) e são fortemente ligados ao núcleo do sistema, não sendo possível utilizar um módulo compilado para um núcleo em outro núcleo.

Ainda hoje existem muitos sistemas operacionais monolíticos. Isso se deve, sem dúvida, à facilidade relativa com que um sistema desse tipo é construído. Além do mais, esse tipo de sistema já tem várias décadas (os sistemas Unix já têm mais de 25 anos).

Por serem muito comuns, a ênfase neste capítulo será na discussão dos sistemas monolíticos Unix (Linux, FreeBSD, OpenBSD, etc.).

2.2 Iniciação

A iniciação de um sistema monolítico Unix é conceitualmente simples se comparada com os sistemas micronúcleo e modulares. Primeiro, nos sistemas monolíticos basta apenas carregar e executar um único arquivo executável, o núcleo do sistema. Segundo, todos os subsistemas são iniciados em uma ordem pré-estabelecida e conhecida por todos.

Com isso, o sistema operacional não precisa se preocupar com a existência de uma dependência cíclica: o subsistema A para iniciar precisa do subsistema B que, por sua vez, precisa do subsistema A.

Esse tipo de problema é comum em sistemas operacionais micronúcleo e modulares, onde um conhecimento prévio do que precisa ser iniciado e de suas dependências não existe, fazendo com que a iniciação seja mais complexa.

Normalmente, a iniciação de um sistema monolítico é feita em duas etapas. Na primeira, um pequeno programa chamado **boot** é carregado, por um *firmware* da própria máquina, na memória. É o **boot** que deve encontrar e carregar o núcleo do sistema para a memória.

A segunda etapa da iniciação é dividida em três grandes estágios [McKu 96]. O primeiro estágio é escrito totalmente em linguagem de montagem (assembly) e faz a iniciação necessária para que código escrito em linguagem de alto nível possa executar. O segundo estágio faz a iniciação dependente de máquina, incluindo a iniciação e a

configuração de dispositivos de entrada/saída. Por fim, o terceiro estágio faz a iniciação independente de máquina para colocar o sistema num estado que permita executar programas.

2.2.1 Primeiro estágio

Como foi dito, o primeiro estágio da iniciação do núcleo do sistema é escrito totalmente em linguagem de montagem já que é muito dependente da máquina e inclui as seguintes tarefas:

- Definir uma pilha
- Identificar o tipo de CPU
- Calcular a quantidade de memória física da máquina
- Habilitar o mecanismo de endereçamento virtual
- Iniciar o dispositivo de gerência de memória
- Criar o contexto para o processo 0
- Chamar o ponto de entrada do código escrito em linguagem de alto nível

Quando o programa **boot** passa o controle para este estágio as interrupções estão desabilitadas e o mecanismo de memória virtual também não está funcionando.

2.2.2 Segundo estágio

Após o código escrito em linguagem de montagem ter executado, ele chama a primeira rotina escrita em linguagem de alto nível, a função main(). Essa função inicia vários subsistemas, começando pelo *console* e o sistema de memória virtual. Além disso, várias tarefas dependentes da máquina são realizadas incluindo:

- Iniciação de uma área de memória para as mensagens de erros
- Alocação de memória para as estruturas de dados do sistema
- Iniciação do gerenciador de memória do sistema

- Autoconfiguração e iniciação dos dispositivos de entrada/saída

A área de memória para as mensagens de erro é um *buffer* circular que guarda as mensagens de diagnóstico dos diversos subsistemas que utilizaram a função printk(). Essas mensagens são de grande valia para identificar possíveis problemas no sistema e, por isso, também são armazenadas nos arquivos de *log*.

A alocação de memória para as estruturas de dados do sistema é bem simples neste estágio da iniciação já que nem todo o subsistema de gerência de memória está funcionando.

Muitas das estruturas alocadas neste estágio têm o seu tamanho determinado na configuração do sistema e, por isso, têm tamanhos fixos. Porém, existem outras estruturas que têm o seu tamanho determinado dinamicamente baseado em dados da própria máquina. Por exemplo, o tamanho do *cache* de dados do disco e o tamanho de estruturas de dados relacionadas com a quantidade de páginas de memória do sistema. Ambas têm o seu tamanho baseado na quantidade de memória disponível na máquina.

Após a iniciação das estruturas de dados do sistema, uma parte muito complexa e importante do sistema é iniciada, a configuração dos dispositivos de hardware. Esta iniciação varia enormemente entre os diversos sistemas operacionais monolíticos. Alguns suportam apenas configuração estática de dispositivos, ou seja, os dispositivos são determinados e configurados na hora de compilar o sistema. Outros, suportam configurações estáticas e dinâmicas feitas com o auxílio de arquivos de configuração.

Antigamente, a configuração estática de dispositivos de hardware não era problema, pois não existiam os dispositivos que existem hoje que podem ser inseridos e retirados facilmente do computador, inclusive quando este está ligado. Além disso, os sistemas monolíticos eram usados em *mainframes* por empresas que não mudavam com freqüência os computadores. Motivo pelo qual o esquema de configuração estática era muito viável.

Com a popularização dos sistemas monolíticos em computadores de uso doméstico, ficou impraticável ter sistema configurados estaticamente. Existem dispositivos que podem ser inseridos e retirados do computador enquanto este está ligado (USB e firewire) e esses computadores sofrem mudanças muito mais rapidamente (atualizações, mudança de placa de vídeo, som, etc.) que os antigos *mainframes* das grandes empresas.

Por essas e outras razões é que atualmente muitos dos sistemas monolíticos suportam a configuração de dispositivos de hardware via arquivos de configuração e também suportam autoconfiguração dos dispositivos (PCI, Plug and Play, etc.).

2.2.3 Terceiro estágio

Por fim, o terceiro e último estágio da iniciação do sistema é executado, realizando a iniciação independente de máquina. A primeira tarefa deste estágio é criar o processo número 0. É este processo que possui todas as configurações que são padrão para um processo nos sistemas Unix.

Após criar o processo 0, vários subsistemas são iniciados, entre eles, pode-se citar: o subsistema de paginação, o subsistema de arquivos, o subsistema de rede, etc. Como explicado anteriormente, todos esses subsistemas são iniciados numa ordem pré-definida e conhecida por todos os subsistemas, não existindo problema de algum subsistema usar outro subsistema que não tenha sido iniciado.

Depois de iniciar os diversos subsistemas, resta ainda montar o sistema de arquivos raiz e criar o processo 1 que faz a iniciação em modo usuário. O sistema de arquivos raiz é montado e o diretório raiz é atribuído ao processo 0.

Finalmente, o sistema está pronto para executar o primeiro programa em modo usuário. O processo 1 é criado e configurado para rodar o programa **init** nos sistemas Unix. Esse programa realiza diversas operações para permitir que usuários possam usar o sistema, incluindo: executar os *scripts* de iniciação, iniciar o console e, em sistemas multiusuários, fazer a autenticação dos usuários.

2.3 Finalização

O processo de finalização de um sistema monolítico é, da mesma forma como a iniciação, um procedimento conceitualmente simples que envolve realizar a finalização dos diversos subsistemas na ordem inversa da iniciação.

Normalmente, grande parte desse processo é realizado por programas executando em modo usuário. Esses programas finalizam todos os processos, desmontam todos os sistemas de arquivos, exceto o sistema de arquivos raiz, entre outras atividades.

Após essa finalização, executada em modo usuário, o núcleo do sistema é chamado para fazer a parte dele, e inclui a finalização de todos os processos que ainda possam estar sendo executados, o desmonte de todos os sistemas de arquivos ainda montados e a execução de uma rotina dependente de máquina que faz a máquina reiniciar ou desligar, dependendo do parâmetro passado para a rotina de finalização.

2.4 Desenvolvimento do sistema

Como foi dito anteriormente, o desenvolvimento do núcleo de um sistema monolítico é como o desenvolvimento de um grande programa. Dessa maneira, para acrescentar novas funções, novos *drivers* ou mesmo para atualizar certa parte do mesmo é preciso um conhecimento profundo de todo o sistema, já que essa mudança pode afetar outras partes do sistema.

Nos sistemas monolíticos, tanto para o desenvolvimento de drivers quanto, principalmente, para o desenvolvimento de novas funções, é necessário ter o código fonte de todo o sistema. Talvez seja por isso que a maioria dos sistemas monolíticos use uma licença de código aberto, permitindo que outras pessoas possam ajudar no seu desenvolvimento. Nos casos de sistemas monolíticos com licenças que não permitem obter o código fonte do mesmo, apenas algumas empresas especializadas em desenvolvimento de novas funções ou drivers para o sistema têm acesso ao código fonte.

Como para desenvolver novas funções ou drivers é preciso um conhecimento muito grande de todo o sistema, ou seja, o tempo de aprendizagem é muito longo, a quantidade de pessoas que começam a trabalhar com o sistema e conseguem construir algo que funcione é relativamente pequena. Nesse sentido, a quantidade de pessoas envolvidas no desenvolvimento do sistema representa um dos grandes problemas no futuro de um sistema monolítico.

Quase todos os sistemas monolíticos usam a linguagem de programação C para codificar o núcleo do sistema, com algumas partes muito dependentes de máquina codificadas em linguagem de montagem. A linguagem C é normalmente utilizada no desenvolvimento de sistemas operacionais por ser de alto nível, mas permitir um grande controle sobre o código gerado.

2.4.1 Atualização e novas funções

Sem dúvida, a parte mais difícil no desenvolvimento de um sistema monolítico é a atualização do mesmo e o desenvolvimento de novas funções (novos mecanismos de comunicação, suporte a um novo sistema de dispositivos (PCI, Plug and Play, USB, etc.). Isso porque as diversas partes do sistema estão fortemente inter-relacionadas dependendo da implementação de cada parte, uma parte do sistema assume que outra parte será implementada de uma determinada maneira.

Com isso, o que parecia somente a mudança de implementação duma parte do

sistema, envolve também a atualização de todo o resto do sistema. Quem acompanha o desenvolvimento do Linux sabe como é complicado mudar certas partes do sistema.

Esse tipo de abordagem onde cada parte depende de **como** outra é implementada ao invés **do que** está sendo implementado torna o desenvolvimento mais complicado e mais sujeito a erros, que, muitas vezes, só aparecem na hora de testar o sistema.

2.4.2 Drivers de dispositivos

O suporte a diversos drivers de dispositivos sem dúvida é uma parte muito importante para um sistema operacional.

No caso dos sistema monolíticos de código aberto, os drivers de dispositivos tem um complicador a mais: devem ter o seu código fonte distribuído para poderem ser compilados em núcleos diferentes. Esse requisito, na maioria das vezes, faz com que as empresas que desenvolvem os dispositivos não desenvolvam drivers para os sistemas monolíticos.

Já no caso dos sistemas monolíticos de código fechado o usuário fica preso ao que o fabricante do sistema oferece ou, no caso de grandes empresas, paga para uma empresa desenvolver o driver de dispositivo que precisa.

Muitos sistemas monolíticos têm uma interface padrão de acesso aos drivers de dispositivo, o que torna o desenvolvimento de um novo *driver* menos complicada, mas não menos trabalhosa.

2.5 Estudo de caso: Linux

Sem dúvida, o sistema operacional Linux é o mais conhecido e usado dos sistemas monolíticos. Muito de seu sucesso no mundo dos sistemas monolíticos deriva do suporte a uma gama muito grande de arquiteturas, entre elas: x86 da Intel, PowerPC da IBM, SPARC da Sun e, ultimamente, vários sistemas embutidos.

Em 1991, Linus Torvards, um estudante de graduação finlandês, queria aprender mais sobre o processador 80386 da Intel e decidiu que a melhor maneira de fazer isso seria construir um sistema operacional. Esse desejo mais o fato de não existirem bons sistemas Unix para essa máquina o levou a implementar um sistema Unix completo e em conformidade com o padrão POSIX [Maxw 99].

Nessa época, o livro "Operating Systems: Design And Implementation" do Andrew S.

Tanenbaum era utilizado numa disciplina cursada por Linus Torvards. Esse livro contém o código fonte do sistema operacional Minix, um sistema feito para o ensino de sistemas operacionais que não utilizava todos os recursos dos novos computadores 80386 da Intel. Porém, apesar de suas limitações, o seu código foi utilizado nas primeiras versões do Linux e ainda hoje é possível ver referências ao Minix no código do Linux.

O Linux foi desenvolvido até a versão 0.02 apenas por Linus Torvards, quando já executava o compilador gcc, o interpretador de comandos bash e alguns utilitários mais. A partir dessa versão ele passou a pedir ajuda de outras pessoas para desenvolver o sistema.

Três anos depois, o Linux já estava na versão 1.0 com milhares de linhas de código, uma implementação preliminar do TCP/IP e algumas dezenas de milhares de usuários.

Hoje em dia o núcleo do sistema já tem milhões de linhas de códigos e outros milhões de usuários em todo o mundo. Em torno do Linux várias grandes empresas surgiram, entre elas a Red Hat, Caldera, Suse e outras gigantes o suportam: Sun, IBM e SGI.

Como [Maxw 99] frisa, para ter o sistema rodando foi mais fácil, no primeiro momento, construir o Linux como um sistema monolítico. Esta decisão evitou a necessidade de se criar um sistema de passagem de mensagem, descobrir uma maneira de carregar módulos, etc. Além disso, existem bastante pessoas trabalhando no núcleo do sistema e nenhum limite de entrega dum produto, o que torna as dificuldades de mudança do núcleo do sistema apenas uma restrição moderada.

Os objetivos que nortearam o desenvolvimento do núcleo do Linux foram e ainda são: clareza, compatibilidade, portabilidade, robustez, segurança e velocidade. Alguns desses objetivos são conflitantes como é o caso da clareza e da velocidade. Algumas vezes a clareza do código é comprometida para torná-lo mais veloz.

Com o passar do tempo e o crescimento do sistema, o núcleo do sistema começou a ficar extremamente grande (na ordem de megabytes) e muito do que estava no núcleo e era carregado para a memória, não era usado por determinados usuários. A única maneira até então para resolver esse problema era recompilar o núcleo do sistema e escolher o que seria inserido no mesmo.

Porém, fazer isso acarretava vários inconvenientes, entre eles: a necessidade de vários programas e arquivos, os quais, para muitos usuários, eram usados apenas para isso, o tempo gasto e a grande possibilidade de erros tanto de compilação quanto de instalação do núcleo do sistema depois de compilado.

Para resolver esses problemas, o conceito de módulos do sistema foi desenvolvido e inserido no núcleo do Linux. Com os módulos do sistema, várias partes do núcleo

foram colocadas em arquivos que só são carregados para a memória quando são utilizados. Isso ajudou a minimizar o problema do tamanho do núcleo do sistema que fica sempre carregado em memória e, principalmente, o problema de ter que recompilar o núcleo do sistema para personalizá-lo.

Agora, com os módulos do sistema, uma distribuição Linux poderia compilar o núcleo do sistema com suporte a diversos drivers de dispositivos, protocolos de rede, etc., distribuir esse núcleo para todos os usuários e deixar o sistema carregar para memória apenas o que fosse usado.

Apesar dessa solução resolver muito bem os dois problemas mencionados, ela deixa muito a desejar no quesito distribuição. Existe uma restrição muito forte nos módulos do sistema: eles só podem ser usados no mesmo núcleo do sistema que foi compilado junto com os módulos. Isso impede, por exemplo, que uma empresa desenvolva e distribua um módulo do sistema para um dispositivo na forma de um arquivo binário sem incluir o código fonte.

À primeira vista pode parecer desnecessário falar sobre a estrutura de diretórios dos fontes do núcleo do sistema, mas essa estrutura de diretórios apresenta de uma forma muito boa como o núcleo do sistema Linux pôde ser desenvolvido para uma gama tão grande de plataformas. Ela se apresenta da seguinte forma:

- **Documentation** - Este diretório contém alguns documentos sobre as partes do núcleo do sistema. Algumas partes, como o sistema de arquivos, são bem documentadas. Para outras não existe nada.

- **arch** - Neste diretório ficam os arquivos dependentes de máquina. Para cada plataforma existe um subdiretório diferente: arch/i386, arch/ppc, arch/sparc, etc. Vale notar como este diretório é pequeno em tamanho se comparado com o resto do núcleo do sistema. Sinal de que muita do código é independente de máquina.

- **drivers** - Este é um dos maiores diretórios do núcleo do sistema. E não poderia ser diferente, já que ele contém todos os drivers de dispositivos suportados pelo Linux como placas de vídeo, de rede, adaptadores SCSI, etc.

- **fs** - Os sistemas de arquivos suportados ficam aqui, cada um num subdiretório. Alguns dos sistemas de arquivos são pseudo-sistemas de arquivos, como o proc, que não têm arquivos de verdade e serve para mostrar dados do sistema de uma forma simples.

- **include** - Neste diretório ficam a maioria dos arquivos de cabeçalho (.h). Existem vários subdiretórios, entre eles *asm-** (onde * é uma das plataformas suportadas: i386, alpha, sparc, etc.) e *linux* para estruturas de dados comuns ao núcleo do sistema e aos aplicativos.
- **init** - Contém o código que coordena a iniciação do núcleo do sistema.
- **ipc** - Diretório com os arquivos que implementam o sistema de comunicação interprocessos do System V (IPC).
- **kernel** - Este diretório contém as funcionalidades do núcleo do sistema que não se encaixam em outros diretórios como o escalonador, código para criar e destruir processos, etc.
- **lib** - Várias funções que são usadas pelo núcleo do sistema e que fazem parte da biblioteca C padrão estão neste diretório. Estão aqui funções para manipular strings e memória, além de funções para descompactar o núcleo do sistema na iniciação.
- **mm** - O código independente de máquina que implementa o gerenciador de memória está neste diretório.
- **net** - Contém os arquivos que implementam os diversos protocolos de redes suportados pelo Linux: AppleTalk, TCP/IP, IPX, etc.
- **scripts** - Este diretório não contém código do núcleo do sistema, mas scripts para configurar o núcleo do sistema para ser compilado.

Um dos pontos principais na organização desses diretórios é a separação do código dependente de máquina do independente. Colocar tudo o que é dependente de máquina num único diretório e utilizar funções e macros para usá-lo provou ser uma estratégia muito boa e que facilita e muito a migração para novas arquiteturas. Além de tornar o código independente de máquina muito mais legível, pois retira os testes para saber qual a arquitetura que está sendo compilada (os famosos #ifdef).

De acordo com [Maxw 99], 26% do total de código escrito para o núcleo do sistema versão 2.2.5 era de código dependente de máquina, mas para uma dada arquitetura esse percentual ficava abaixo dos 4%, o que mostra como é relativamente fácil migrar o núcleo do sistema para novas arquiteturas.

2.6 Conclusões

Os sistemas operacionais monolíticos foram e ainda são muito importantes tanto no mercado de sistemas operacionais de uso geral (Linux, FreeBSD, AIX, etc.) quanto no mercado de sistema embutidos.

No entanto, existem grandes desafios a serem enfrentados por esses sistemas. Um deles está relacionado com as novas tecnologias que permitem retirar e inserir novos periféricos com muita facilidade e que necessitam, para isso, de suporte do sistema operacional.

Esse não é um problema muito grande no mercado de servidores, já que estes não mudam com muita frequência. Porém no mercado doméstico, esse é um problema sério.

Para solucionar esse problema, vários sistemas que eram antigamente totalmente monolíticos desenvolveram soluções que flexibilizam o sistema operacional, permitindo que este seja modificado sem ter que recompilar o núcleo do sistema. Entre esses sistemas se destacam o Linux e o Solaris com as suas soluções de módulos do núcleo.

Mesmo com esse problema de flexibilidade, os sistemas operacionais monolíticos têm uma grande vantagem: a velocidade. Embora os novos modelos de sistemas operacionais (modulares, por exemplo) não sejam muito piores que os monolíticos nesse quesito e a evolução dos processadores torne essa questão menos visível, esses sistemas ainda são os mais rápidos.

O que se percebe no desenvolvimento dos sistemas monolíticos, especialmente no caso do Linux, é um movimento para tornar o sistema mais flexível e permitir que a configuração do sistema mude (adicionando novos periféricos, por exemplo) sem que isso implique em compilar um novo núcleo para suportar essa mudança. Essa tendência ficará mais evidente quanto mais esses sistemas começarem a ser usados cada vez mais em estações de trabalho e no mercado doméstico.

Capítulo 3

Sistemas micronúcleo

3.1 Introdução

Os sistemas operacionais do tipo micronúcleo surgiram como uma alternativa para tentar resolver alguns dos problemas dos sistemas Unixes monolíticos que estavam ficando a cada nova versão maiores e mais complexos, tornando-se, dessa forma, difíceis de serem manejados [Vaha 96].

Para tentar solucionar esses problemas, uma nova arquitetura foi proposta que consiste no seguinte: um pequeno núcleo com apenas as primitivas básicas do sistema e vários processos cooperativos que usando as primitivas do micronúcleo acrescentam funcionalidades ao sistema. Normalmente, a comunicação entre os diversos processos cooperativos é feita através da passagem de mensagens e esta é outra característica marcante neste tipo de sistema operacional.

As primitivas básicas implementadas pelo micronúcleo variam de sistema para sistema, mas geralmente incluem primitivas para gerenciamento de memória, gerenciamento de processos, sincronização e comunicação interprocessos. Existem certos sistemas, os chamados sistemas nanonúcleos, que implementam uma quantidade de primitivas ainda menor.

Apesar de possuir uma arquitetura consistente, os sistemas com micronúcleo sofrem de um problema sério: a falta de desempenho. Por utilizar processos cooperativos e o conceito de passagem de mensagens, o desempenho do sistema fica comprometido já que várias trocas de contexto precisam ser feitas para completar um pedido de serviço. Esse problema era mais acentuado antigamente quando os computadores não eram tão

potentes e não tinham os recursos atuais (por exemplo, manter certas páginas no *cache* mesmo numa troca de contexto).

Justamente por apresentar esse problema sério que este tipo de sistema ficou por muito tempo restrito aos meios acadêmicos e mesmo hoje em dia existem poucos sistemas micronúcleo sendo utilizados comercialmente com sucesso. Os mais conhecidos são o QNX e o Mach.

O sistema operacional Mach começou a ser desenvolvido nos anos de 1980 por pesquisadores da universidade Carnegie-Mellon de Pittsburgh e o objetivo era desenvolver um sistema com um micronúcleo que suportasse a interface de programação do Unix, executasse em uniprocessadores e também em multiprocessadores e fosse adequado para um ambiente distribuído [Vaha 96].

Apesar desse objetivo, as primeiras versões do Mach utilizavam núcleos monolíticos e somente a partir do Mach 3.0 um micronúcleo foi utilizado. Muitos sistemas baseados no Mach utilizam a versão com núcleo monolítico como base e não a nova versão com micronúcleo, entre eles OSF/1, NextStep e mais recentemente o Mac OS X da Apple.

Ao contrário do Mach, o QNX desde o começo em 1980 foi um sistema com micronúcleo utilizando processos cooperativos, passagem de mensagens, suporte a multiprocessadores, memória virtual protegida, suporte a tempo real entre outras características que deram a ele uma reputação muito boa em termos de confiabilidade e estabilidade e que fez com que o QNX fosse, e ainda seja, utilizado em aplicativos de missão-crítica como centrais nucleares, equipamentos médicos e também em sistemas de telecomunicação [Bene 02]. No começo era um sistema somente para máquinas da arquitetura Intel, mas em 1996 uma nova versão do micronúcleo chamada de "Neutrino" foi criada para ser usada em diferentes arquiteturas (PowerPC, MIPS, StrongARM, etc.).

Neste capítulo será analisado com mais detalhes o sistema operacional QNX já que ele continua sendo um dos sistemas micronúcleo mais bem sucedidos comercialmente e por disponibilizar as informações necessárias a este trabalho.

3.2 Caracterização do QNX

O sistema operacional QNX é um sistema de tempo real POSIX utilizando uma arquitetura micronúcleo baseada em passagem de mensagens. A passagem de mensagens é o principal meio de comunicação entre os processos e a maioria das chamadas do sistema usam esse tipo de comunicação. Por exemplo, quando um aplicativo quer abrir

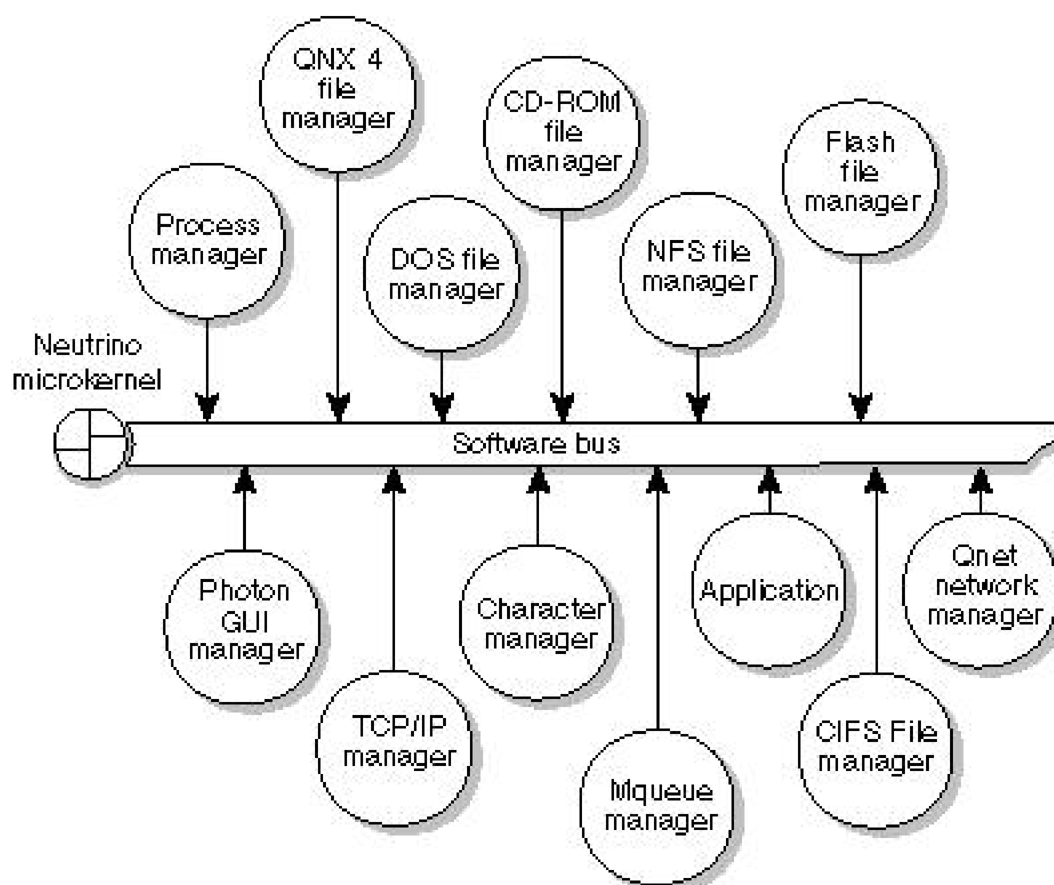

Figura 3.1: Arquitetura do QNX

um arquivo, a chamada do sistema é convertida numa mensagem e enviada para o gerenciador dos sistemas de arquivos que responde com um descritor de arquivo. Como esse mecanismo de passagem de mensagens é transparente para os aplicativos o sistema pode ser distribuído facilmente entre diversos nodos de uma rede.

O sistema é constituído de um pequeno micronúcleo chamado de Neutrino que administra um grupo de processos cooperativos que executam código em modo não privilegiado, ou seja, não têm acesso a todos os recursos da máquina. Como a figura 3.1 mostra, a estrutura do sistema parece mais uma equipe do que uma hierarquia de processos. O micronúcleo serve como uma cola, ligando os diferentes processos.

O QNX foi o primeiro sistema operacional do seu tipo a utilizar passagem de mensagens como a forma principal de comunicação entre processos. Muito da

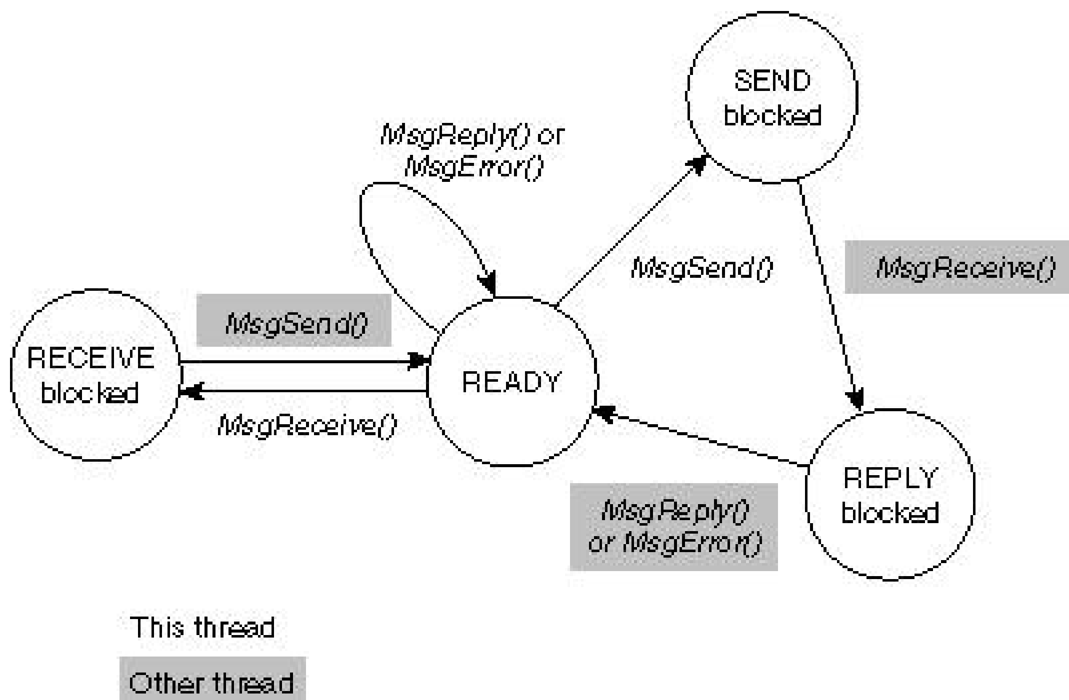

Figura 3.2: Estados de um fluxo de controle usando passagem de mensagens

flexibilidade, simplicidade e elegância se devem a completa integração desse método por todas as partes do sistema.

O conteúdo de uma mensagem no QNX não tem nenhum significado para o micronúcleo em si. O conteúdo da mensagem tem um significado somente para quem a envia e a recebe.

Além de servir como um meio de comunicação, a passagem de mensagens também serve como um meio de sincronização entre processos. Como a figura 3.2 mostra, a medida que uma mensagem é enviada, recebida e respondida, o processo passa por várias mudanças de estado que afetam quando e por quanto tempo um processo executa.

O micronúcleo do QNX é bem pequeno como são os sistemas de tempo real e fornece os seguintes serviços:

- fluxos de controle (threads) - o micronúcleo fornece primitivas POSIX para fluxos de controle;

- sinais - o micronúcleo fornece primitivas POSIX para sinais;

- passagem de mensagens - o micronúcleo faz o roteamento das mensagens para os fluxos de controle;
- sincronização - o micronúcleo fornece primitivas POSIX para sincronização;
- escalonamento - o micronúcleo utiliza os diversos algoritmos de tempo real POSIX para escalonar os fluxos de controle;
- temporização - o micronúcleo fornece primitivas POSIX de temporização;
- gerência de processos - o micronúcleo e o gerenciador de processos formam uma unidade (chamada de procnto) que gerencia os processos, memória e o espaço de nomes.

Todos os demais serviços são fornecidos por processos convencionais. Uma possível configuração do sistema poderia incluir:

- gerenciadores de sistemas de arquivos (FAT, QNX, cd-rom, etc.);
- dispositivos orientados a caracter (porta paralela, porta serial, etc.);
- interface gráfica (Photon);
- TCP/IP.

Essa arquitetura fornece uma estensibilidade muito grande combinada com a simplicidade para estender o sistema: basta escrever novos programas para estender o sistema.

Ao contrário de muito sistemas operacionais onde os controladores de dispositivos estão intimamente ligados ao sistema, no QNX os controladores de dispositivos também são processos convencionais que podem ser iniciados e parados a qualquer momento. Com isso, incluir um novo controlador de dispositivo não perturba o resto do sistema e o seu desenvolvimento e depuração pode ser feito como outro programa qualquer.

3.3 Iniciação

O processo de iniciação do QNX não é muito diferente do processo de iniciação de um sistema monolítico. No QNX, um arquivo chamado de imagem do sistema é carregado para a memória e executado. Esse arquivo contém tudo que é necessário para o sistema

iniciar (o micronúcleo, os diversos programas como, por exemplo, os sistemas de arquivos, o interpretador de comandos, etc.).

Várias configurações do sistema podem ser feitas através da utilização de diferentes imagens do sistema. Para criar essas imagens do sistema, um arquivo contendo as regras de criação da imagem do sistema é usado. Esse arquivo contém três partes:

- um *script* para iniciação da máquina;
- um *script* para a iniciação do QNX;
- uma seção com uma lista de arquivos.

Como diferentes máquinas fazem o *boot* de maneira diferente, um *script* de iniciação da máquina é necessário para especificar as ações a serem tomadas para iniciar o sistema. Esta parte descreve o programa de iniciação a ser usado, incluindo qualquer ação específica para uma operação correta da máquina como, por exemplo, desabilitar a instrução HLT em processadores Pentium com defeito, reservar alguma área de memória física necessária para um controlador de dispositivo, entre outros. Um caso típico seria:

```
[virtual=x86,bios +compress] .bootstrap = {
    startup-bios -s 64k -A -vv
    PATH=/proc/boot procnto
}
```

Nesse exemplo, vários parâmetros de configuração são usados. **“virtual”** diz para usar um espaço de endereçamento virtual, **“x86,bios”** especifica o processador e o tipo de *boot* (fazer o *boot* via BIOS) e **“+compress”** diz para compactar a imagem do sistema.

O *script* em si inicia o programa **startup-bios** que é responsável por descompactar a imagem, fazer algumas configurações e executar o micronúcleo e **procnto** que é o micronúcleo junto com o gerenciador de processos.

Neste ponto, o micronúcleo está pronto para iniciar processos que vão colocar o sistema num estado que permita a sua utilização. O script para iniciação do QNX é executado neste momento e se parece com:

```
[+script] .script = {
    PATH=/proc/boot:/usr/bin:/bin
    LD_LIBRARY_PATH=/proc/boot:/usr/lib:/lib
```

```
        # start pci server
        pci-bios &
        # wait till pci-bios probes system
        waitfor /dev/pci
        # start console driver with two virtual consoles
        devc-con -n2 &
        # start shell on the consoles
        reopen /dev/con1
        [+session] uesh &
        reopen /dev/con2
        [+session] uesh &
    }
```

Como se nota, esta configuração do sistema é bem básica iniciando o servidor do barramento PCI e o servidor do console e criando duas sessões com interpretadores de comandos. Este *script* é criado especificamente para cada necessidade e pode ser muito maior iniciando, por exemplo, a rede e um servidor de janelas.

Finalmente, a última parte do arquivo contém os arquivos necessários para criar a imagem do sistema:

```
libc.so.1
[type=link] /usr/lib/libc.so.1=/proc/boot/libc.so.1
[data=copy]
[perms=+r,+x]
devc-con
uesh
pci-bios
```

São incluídos nesta parte bibliotecas necessárias para executar os programas e os próprios programas utilizados. Além disso são especificadas as permissões dos arquivos executáveis e algumas ligações simbólicas que devem ser criadas.

Com esse arquivo de configuração pronto, o arquivo com a imagem do sistema deve ser criado com o comando **mkifs**:

```
mkifs -v arquivo_conf imagem
```

e colocado no disquete com o comando **dinit**:

```
dinit -f imagem /dev/fd0
```

3.4 Finalização

A finalização do QNX segue o mesmo padrão de finalização dos sistemas Unixes, onde a maior parte do processo de finalização é feita por programas. No caso do QNX o processo **proc32** é o responsável por enviar o sinal SIGPWR para todos os processos quando o sistema vai ser reiniciado ou desligado.

Esse sinal, na maioria dos casos, faz o processo finalizar. Em certos casos o processo trata esse sinal para fazer outras finalizações como, por exemplo, escrever em disco as páginas de memória modificadas.

Como no QNX a maioria das funcionalidades do sistema é fornecida por processos convencionais, o micronúcleo não tem um papel muito importante na finalização do sistema. Ao contrário, por exemplo, dos sistema monolíticos onde todos os subsistemas (memória, discos, etc.) devem ser finalizados pelo núcleo do sistema.

Depois de enviar esse sinal, o processo **proc32** utiliza diferentes mecanismos específicos de cada arquitetura de computador para reiniciar ou desligar a máquina.

3.5 Desenvolvimento do sistema

O desenvolvimento de novas funcionalidades ou drivers de dispositivos para o QNX é como o desenvolvimento de um programa convencional já que essas funcionalidades ou drivers de dispositivos vão executar como processos convencionais. Isso facilita muito o desenvolvimento de novas funcionalidades já que as ferramentas existentes para desenvolvimento de programas como, por exemplo, o depurador podem ser utilizadas.

Outra vantagem é que esses serviços ou drivers de dispositivos podem ser parados e iniciados como qualquer outro processo, não necessitando que o sistema seja reiniciado. Isso diminui muito o tempo de desenvolvimento de um driver de dispositivo, principalmente a parte de testes onde várias versões do programa podem ser necessárias.

Assim como os sistemas modulares, o desenvolvimento de novas funcionalidades para o sistema não precisa do código fonte dos outros programas e do micronúcleo utilizado.

Como o principal meio de comunicação entre os processos no QNX é a passagem de mensagens, o modelo mais adequado para o fornecimento de serviços pelos processos que estendem o sistema é o modelo cliente/servidor. Neste modelo, o servidor que fornece os serviços fica no aguardo de requisições dos clientes. Ao receber as requisições elas são processadas e uma resposta é enviada de volta para o cliente que requisitou o serviço.

Na página da internet do QNX (http://www.qnx.com) existem vários DDKs (Driver Development Kits). Entre eles o de áudio, vídeo, entrada, rede, entre outros. Esses DDKs contém diversas informações sobre as APIs a serem utilizadas e até exemplos de como desenvolver um driver de dispositivo.

3.6 Estudo de caso: HURD

A história do desenvolvimento do sistema operacional Hurd começa com o advento do projeto GNU que pretende criar um sistema operacional totalmente livre. Não apenas o núcleo do sistema operacional, mas todo o sistema com seus diversos programas.

Depois de muita discussão sobre qual núcleo deveria ser usado para o sistema ou se um totalmente novo deveria ser feito, foi decidido em 1990 que um sistema baseado no micronúcleo Mach seria utilizado. A decisão de usar o Mach demorou por causa de problemas com a licença e somente depois dessa licença ter sido alterada o micronúcleo Mach foi adotado.

O Hurd é um sistema operacional multi-servidor compatível com POSIX que utiliza um micronúcleo baseado no Mach 4.0 chamado de GNU Mach. O projeto Hurd mantém o micronúcleo GNU Mach porque ele foi modificado para utilizar vários controladores de dispositivos de disco e rede do núcleo do Linux versão 2.0.

A arquitetura utilizada pelo Hurd é muito parecida com a usada pelo QNX. Existem diversos processos servidores que fornecem serviços para o restante do sistema. Assim como no QNX, a comunicação entre os processos é feita através da passagem de mensagens e, para isso, os serviços do micronúcleo são utilizados.

O sistema de arquivos no Hurd tem um papel muito importante pois é através dele que os clientes encontram os servidores. Os servidores disponibilizam os seus serviços através de um arquivo especial criado para isso.

3.7 Conclusões

Os sistemas micronúcleo foram concebidos para tentar resolver os problemas encontrados nos sistemas monolíticos. Tiveram muito sucesso no problema do forte acoplamento necessário nos sistemas monolíticos e também na grande flexibilidade conseguida. Porém esse sucesso teve um custo muito alto no desempenho do sistema que inviabilizou o seu uso fora do meio acadêmico por vários anos.

Devido a esse problema de desempenho, alguns sistemas operacionais adotaram uma forma híbrida. Nesses sistemas o micronúcleo tem uma responsabilidade maior fornecendo muitos serviços que seriam providos por processos convencionais num sistema micronúcleo original. Dentre os sistemas híbridos podemos destacar as primeiras versões do Windows NT. Mesmo assim, as primeiras versões do Windows NT são conhecidas por serem muito lentas, o que levou a Microsoft a diminuir ao máximo o uso da arquitetura micronúcleo nos sucessores Windows 2000 e Windows XP.

Somente recentemente com o enorme aumento da potência dos computadores pessoais foi possível utilizar essa arquitetura de sistema operacional mais amplamente. Mesmo assim, ainda existem muito poucos sistemas operacionais baseados nessa arquitetura e, os que existem, são muitas vezes para propósitos específicos como é o caso do QNX.

Capítulo 4

Sistemas modulares

4.1 Introdução

Os sistemas operacionais modulares são um meio termo entre os sistemas monolíticos e os sistemas micronúcleo. Eles foram desenvolvidos para resolver o problema de falta de flexibilidade dos sistemas monolíticos, que dificulta a expansão e atualização desses sistemas, sem ter o baixo desempenho dos sistemas micronúcleo, que impedia, e talvez ainda impeça, a sua utilização por um número maior de usuários.

Isso é conseguido fazendo com que as diversas partes do sistema se comuniquem usando o mecanismo de chamada de funções do próprio processador como nos sistemas monolíticos, mas permitindo que certas partes do sistema fiquem em arquivos separados que são carregados quando necessário, lembrando os sistemas micronúcleo.

Embora na maioria dos sistemas modulares esse esquema não seja tão flexível quanto o micronúcleo, ele é muito mais flexível e amigável para um usuário final que os sistemas monolíticos, pois basta acrescentar ou atualizar um arquivo e reiniciar o sistema para ter novas funcionalidades ou correções do sistema em uso.

Nos sistemas modulares, várias funcionalidades do sistema são fornecidas por arquivos binários chamados módulos ou bibliotecas de ligação dinâmica, que são carregados somente quando são utilizados. Esses módulos têm funções conhecidas por quem os usa, o que permite, por exemplo, que empresas desenvolvam drivers de dispositivo para esses sistemas sem que, necessariamente, o código fonte tenha de ser distribuído também.

Neste capítulo o foco será o sistema operacional Windows NT e suas variantes

(Windows 2000 e XP) já que esses são os sistemas modulares mais conhecidos e usados no mundo.

O Windows NT começou a ser desenvolvido no ano de 1988 e a primeira versão (Windows NT 3.1) saiu em 1993. O sistema seguia o modelo micronúcleo com um escalonador preemptivo, suporte a multiprocessadores e uma nova API (Win32).

Ao longo dos anos seguintes e das sucessivas versões (3.5 em 1994, 3.51 em 1995 e 4.0 em 1996) o sistema sofreu várias modificações, passando cada vez mais de um sistema micronúcleo para um sistema modular, principalmente para aumentar o desempenho do sistema que era conhecido por ser extremamente lento. E essa mudança se acentuou ainda mais com o Windows 2000 e XP.

4.2 Iniciação

Sem dúvida, uma das partes mais difíceis de serem desenvolvidas num sistema modular é a sua iniciação. Como fazer com que as diversas partes do sistema sejam carregadas, iniciadas e estejam prontas para uso sem o auxílio de um sistema operacional? Muitas vezes, para que isso seja possível, vários mecanismos, alguns não muito elegantes, são usados.

A iniciação dos sistemas modulares tem diversas dificuldades inerentes ao modelo adotado. Uma delas é como carregar para a memória os diversos módulos do sistema se para isso alguns módulos do próprio sistema são necessários (gerenciador de memória, leitor do sistema de arquivos, leitor de disco, etc.). Outra é como saber a ordem de iniciação dos diversos módulos, muitos feitos por outras empresas, não planejados na hora de desenvolver o sistema, etc.

A solução para esses e outros problemas é feita de uma forma muito elegante pelo Windows 2000 e começa na hora da instalação do sistema numa máquina. O processo de iniciação do Windows 2000 é divido nas seguintes etapas:

- **Setor de boot** - Responsável por ler o arquivo Ntldr do diretório raiz
- **Ntldr** - Lê os arquivos boot.ini, ntoskrnl.exe, bootvid.dll, hal.dll e os drivers de dispositivos que são iniciados no *boot*
- **Ntoskrnl.exe** - Inicia os subsistemas, inicia os drivers de dispositivos carregados anteriormente e prepara o sistema para rodar aplicativos

Na instalação do Windows 2000 numa máquina, é escolhido o tipo de sistema de arquivos que vai ser usado e dependendo dessa escolha um setor de boot específico para esse sistema de arquivos é gravado em disco. Esse setor de boot contém código apenas capaz de ler do sistema de arquivos e acessar os arquivos no diretório raiz.

Depois do setor de boot ter lido o arquivo Ntldr, o controle é passado para o ponto de entrada desse arquivo. Esse pequeno programa opera nas máquinas da arquitetura Intel nos modos 16 bits e 32 bits. Isso porque para usar a BIOS da máquina o processador tem que estar no modo 16 bits, mas para acessar toda a memória o processador tem que estar em 32 bits.

O Ntldr começa limpando a tela e lendo o arquivo boot.ini. Esse arquivo contém diversas diretivas que controlam como o sistema é iniciado. Por exemplo, /3GB para usar 3 GB de espaço de memória para os programas, /BOOTLOG para escrever um arquivo de log do boot, entre outras.

Antes de outros arquivos serem lidos, o programa Ntdetect.com é carregado e executado. Esse é um programa de 16 bits que consulta a BIOS para obter vários parâmetros do computador, incluindo:

- A hora e data do sistema
- Os tipos de barramentos (ISA, PCI, MCA, etc.) e a identificação dos dispositivos ligados a eles
- O número, tamanho e tipo dos discos no sistema
- O tipo de mouse presente
- O número e tipo das portas paralelas configuradas no sistema

Essas informações são armazenadas em estruturas internas que mais tarde são passadas para o núcleo do sistema carregado.

Após executar essas etapas, começa o carregamento dos arquivos necessários à iniciação do sistema, entre eles: o núcleo do sistema (Ntoskrnl.exe), o HAL (Hardware Abstract Layer), o arquivo com as configurações do sistema e os drivers de dispositivos indicados nas configurações do sistema.

Feito esse carregamento, o controle é passado para o núcleo do sistema (Ntoskrnl.exe) para terminar a iniciação do sistema.

O Ntoskrnl recebe de Ntldr uma estrutura que contém parâmetros do arquivo boot.ini, um ponteiro para as tabelas de páginas construídas por Ntldr para descrever a memória física do sistema, um ponteiro para o arquivo de configurações e um ponteiro para a lista de drivers carregados.

A iniciação feita por Ntoskrnl é dividida em duas fases: fase 0 e fase 1. Para que os diversos subsistemas e drivers possam saber em que fase da iniciação estão executando, um parâmetro indicando a fase é passado para a rotina de iniciação de cada subsistema ou driver de dispositivo.

O propósito da fase 0 é construir as estruturas mínimas necessárias para que a fase 1 possa executar. Durante a fase 0, as interrupções estão desabilitadas.

Depois do HAL ter iniciado o controlador de interrupções de cada processador do sistema e ter configurado a interrupção responsável pelo "clock tick", as seguintes iniciações são feitas:

- O gerenciador de memória constrói as tabelas de páginas e as estruturas internas para prover serviços básicos de memória.
- O gerenciador de objetos do sistema constrói o espaço de nomes e alguns objetos pré-definidos.
- O monitor de segurança inicia os objetos de segurança.
- O gerenciador de processos faz a maior parte de sua iniciação na fase 0, criando os objetos processo e linha de execução (thread) e construindo listas dos processos ativos. Nesta fase, as estruturas para o primeiro processo, o processo Idle, são criadas. Por fim, o gerenciador de processos cria o processo System (que contém as linhas de execução do tipo sistema) e cria uma linha de execução do tipo sistema que vai executar a fase 1. Essa linha de execução não começa a executar imediatamente porque as interrupções estão desabilitadas.
- O gerenciador Plug and Play faz a sua iniciação criando um sincronizador para os recursos dos barramentos.

Após essas iniciações, as interrupções são habilitadas e a linha de execução responsável pela fase 1 começa a executar os seguintes passos:

1. O driver de vídeo somente utilizado na iniciação do sistema (bootvid.dll) é iniciado e mostra a tela de iniciação do Windows 2000

2. O gerenciador de energia é iniciado
3. A hora do sistema é iniciada
4. Se existirem mais processadores no sistemas eles são iniciados e começam a executar
5. O gerenciador de objetos cria mais alguns tipos de objetos
6. Os objetos de sincronização (semáforos, mutex, eventos, etc.) são criados
7. As estruturas de dados do escalonador são criadas
8. O gerenciador de memória é chamado para criar as linhas de execução que fazem vários tipos de tarefas relacionadas com o mecanismo de paginação
9. O gerenciador de cache inicia as suas estruturas de dados e cria linhas de execução para fazer a sua manutenção
10. O gerenciador de configuração é iniciado e copia os dados passados pelo Ntldr para o registro do Windows
11. O gerenciador de entrada/saída inicia as suas estruturas internas e depois os drivers de dispositivos (tanto os carregados pelo Ntldr quanto os outros registrados no sistema). Este é um passo grande e responsável pela maior parte do tempo gasto na iniciação.
12. Finalmente, o primeiro programa é executado

Esses são os passos executados pelo núcleo do sistema, o resto da iniciação é feito pelo primeiro programa e inclui iniciar os programas que são serviços, abrir as DLLs conhecidas, criar as variáveis de ambiente, entre outras coisas.

4.2.1 Arquivo de log da iniciação

Quando o parâmetro de iniciação /BOOTLOG é passado para o núcleo do sistema, todos os drivers de dispositivos carregados e os que não foram carregados são listados num arquivo de log.

Como o núcleo do sistema evita modificar o disco antes que o programa Chkdsk execute, todas as informações que serão escritas no arquivo de log ficam armazenadas

temporariamente em memória. Sendo o primeiro programa a executar em modo-usuário, o Session Manager executa o Chkdsk para verificar os discos e chama uma função do sistema para indicar que tudo está bem. Quando essa função é chamada o núcleo do sistema sabe que pode criar um arquivo de log e escrever tudo que está na memória para ele. Após essa função ser chamada, tudo o que for para o arquivo de log é escrito imediatamente.

4.2.2 Modo de segurança

Como [Solo 00] afirma, a razão mais comum para o Windows 2000 não poder iniciar é porque um driver de dispositivo derruba o sistema enquanto ele inicia. Para tentar resolver esses casos em que o sistema não consegue iniciar, o Windows 2000 tem um modo de iniciação chamado *Modo de Segurança* que inicia o mínimo necessário de drivers de dispositivos e de serviços.

Fazer isso não significa que o sistema vai iniciar corretamente, mas minimiza a possibilidade de erro já que muitos drivers de dispositivos fornecidos por terceiros e muitos drivers não essenciais não são iniciados.

Para saber quais drivers de dispositivo carregar e iniciar, o Windows 2000 consulta o registro do sistema. A ordem de iniciação dos drivers é feita através de grupos de drivers. Existe o grupo Boot, o grupo File, entre outros. Quando um driver de dispositivo é instalado no sistema é passado um parâmetro indicando o grupo no qual o driver deve ser inserido.

Um possível problema no processo de iniciação do Windows 2000 em modo de segurança é apontado por [Solo 00]: o programa Ntldr (responsável por carregar os drivers do sistema) carrega todos os drivers de dispositivos marcados para serem carregados na iniciação do sistema ao invés de carregar apenas os drivers de dispositivos que devem ser iniciados no modo de segurança. Assim, se o carregamento de um desses drivers estiver causando problemas na iniciação, iniciar em modo de segurança não resolverá.

Além dessa parte da iniciação feita no núcleo do sistema, existe outra que deve ser feita por programas executando em modo-usuário. Como os serviços do Windows 2000 são iniciados em modo-usuário, o programa que inicia os serviços verifica no registro do sistema se o Windows 2000 está em modo de segurança e, se estiver, inicia apenas os serviços essenciais.

4.3 Finalização

A finalização do Windows 2000 segue o mesmo esquema geral dos outros sistemas operacionais, ou seja, termina todos os processos, finaliza os subsistemas e reinicia ou pára o computador.

Quando um processo inicia a finalização do sistema, um processo especial chamado Csrss percorre todos os processos de cada usuário enviando uma mensagem do tipo WM_QUERYENDSESSION para cada linha de execução do processo. Se a linha de execução retornar TRUE, a finalização pode continuar e nesse caso o Csrss envia outra mensagem agora do tipo WM_ENDSESSION para a linha de execução finalizar. O processo Csrss espera alguns segundos por uma resposta e, se ela não chegar, mostra uma janela informando o usuário que o processo não está respondendo e permite matar o processo ou cancelar a finalização.

Após terminar a finalização dos processos dos usuários do sistema, Csrss inicia a finalização dos processos do sistema. Essa finalização segue o mesmo esquema anterior de envio de mensagens e espera de resposta, mas desta vez nenhuma janela informando que o processo não está respondendo é mostrada e nenhum processo é encerrado. Assim, quando o sistema finaliza muitos processos do sistema ainda estão executando, entre eles: Smss (Session Manager), Winlogon, SCM (Service Control Manager) e Lsass (Local Security Authentication Server).

Finalmente, uma função do núcleo do sistema é chamada para fazer a finalização dos drivers de dispositivos e dos subsistemas (gerenciador Plug and Play, gerenciador de energia, gerenciador de entrada/saída, gerenciador de configuração, gerenciador de memória, etc.).

Por exemplo, o gerenciador de entrada/saída envia um pedido para os drivers de dispositivos avisando que o sistema está finalizando, o gerenciador de configuração grava em disco todos as configurações que estão em memória, o gerenciador de memória grava as páginas de memória modificadas.

O último subsistema a ser finalizado é o gerenciador de energia que reinicia ou desliga a máquina.

4.4 Desenvolvimento do sistema

Ao contrário dos sistemas monolíticos, no desenvolvimento dos sistemas modulares o código fonte de todo o sistema não é necessário. O que é necessário são as definições das funções e das estruturas do sistema.

Com isso, o desenvolvimento de novas funções ou drivers de dispositivos fica mais acessível e pode ser feito por pessoas sem um relacionamento direto com o fabricante do sistema operacional.

Outro fator facilitador no desenvolvimento dos sistemas modulares é a existência de uma interface bem conhecida e documentada das funções disponíveis para os drivers de dispositivos e das funções que os mesmos devem implementar. Isso facilita muito o desenvolvimento já que a pessoa que vai implementar o driver de dispositivo não precisa pensar no desenvolvimento dessa interface.

Um aspecto muito positivo dos sistemas modulares é a facilidade de distribuição e instalação de novas funções e/ou drivers de dispositivos, afinal é nessa etapa que um usuário final, muitas vezes sem conhecimentos técnicos sobre o sistema, participa. A facilidade de desenvolver e de testar as novas funções e/ou drivers de dispositivo é muito importante para os **desenvolvedores** e não para os **usuários finais**.

Nos sistemas modulares, a distribuição de novas funções e/ou drivers de dispositivos pode ser através de uma arquivo executável num processo automatizado sem a intervenção do usuário, já que a instalação nada mais é do que gravar em disco e registrar no sistema o driver de dispositivo. O máximo que o usuário final precisa fazer é apertar o botão OK para finalizar o processo de instalação ou para reiniciar o computador em alguns casos.

No caso do Windows 2000 que é um sistema operacional proprietário de código fechado, o desenvolvimento de novas funções para o sistema é feito exclusivamente pela fabricante, a Microsoft, ou por empresas ligadas à Microsoft. Existem várias empresas que desenvolvem componentes diferenciados para o Windows 2000, como um núcleo do sistema mais adequado para sistemas de tempo real, por exemplo. Nesses casos, as empresas têm acesso ao código fonte do Windows 2000 para poder fazer um bom produto.

Já no caso dos drivers de dispositivo o cenário é diferente. Qualquer pessoa com o compilador e o DDK (Device Driver Kit) da Microsoft é capaz de desenvolver novos drivers. Inclusive, o próprio DDK tem vários exemplos e esqueletos de drivers de diversos dispositivos como vídeo, teclado, modem, discos, entre outros.

Essa estratégia é particularmente importante para a Microsoft, já que assim ela

mantém o controle das principais funcionalidades do sistema mas, ao mesmo tempo, permite que uma ampla gama de dispositivos seja suportada.

4.5 Estudo de caso: Solaris

As primeiras versões do Unix da Sun eram baseadas no BSD e se chamavam SunOS. Com a mudança para uma versão baseada no System V Release 4 (SVR4), a Sun mudou o nome para Solaris mantendo, porém, o nome SunOS nos componentes do núcleo do sistema.

O SunOS 1.0 era baseado no BSD 4.1 de 1982 dos laboratórios Berkeley e foi implementado para os processadores Motorola 68000. Nesse tempo o sistema era pequeno e compacto rodando com apenas 1 megabyte de memória.

Com a crescente demanda por sistemas Unixes e por sistemas com suporte a redes, a Sun investiu pesado no SunOS. A versão 2.0 de 1985 oferecia vários componentes de rede avançados como o suporte a *remote procedure call* (RPC), o *Network Information Service* (NIS) e um novo modelo de sistema de arquivos (*virtual file system*), com suporte ao NFS.

Com esses novos componentes, a base de usuários do SunOS cresceu muito e mais aplicativos foram desenvolvidos. Esses novos aplicativos tinham requisitos muito maiores, como memória compartilhada, arquivos mapeados em memória, entre outros. Essa pressão dos aplicativos fez surgir uma nova fase de inovações no sistema que culminou com o lançamento do SunOS 4.0 em 1988. Essa versão tinha um novo sistema de memória virtual para acomodar as necessidades crescentes dos aplicativos.

Durante os anos de 1980, a demanda por capacidade de processamento aumento muito e como apenas um processador não conseguia mais prover todas as necessidades, os sistema multiprocessados começaram a surgir. Esse avanço nos computadores impulsionou, novamente, o desenvolvimento de um novo núcleo para o sistema SunOS: o SunOS 4.1 de 1990. Essa versão suportava asymetric multiprocessing (ASMP) de uma forma muito limitada, onde o núcleo só podia executar num processador por vez. Esse modelo de multiprocessamento foi um grande passo a frente, mas quando se aumentava o número de processadores o ganho não crescia na mesma proporção. Uma implementação melhor de multiprocessamento era necessária.

Nessa época a Sun estava participando num desenvolvimento conjunto com a AT&T e o SVR4 incorporou muitos dos avanços do SunOS, do BSD e do Xenix. Prevendo

o crescimento dos sistema multiprocessados, a Sun investiu pesado num novo sistema com foco principal na escalabilidade em multiprocessadores. Esse novo sistema oferecia linhas de execução (threads) que junto com uma sincronização fina no núcleo do sistema, é a base do sistema Solaris de hoje.

Com a adoção do novo sistema baseado no SVR4, o sistema como um todo passou a se chamar Solaris, sendo o Solaris 2.0 de 1992 a primeira versão, e os componentes do núcleo do sistema passaram para a versão SunOS 5.0.

Esse novo sistema foi estruturado de uma forma modular, o que permitiu que novas arquiteturas fossem suportadas (Intel x86 e o novo processador de 64 bits Itanium). Além disso, a versão Solaris 7 de 1998 introduziu uma implementação de 64 bits.

Num primeiro momento pode parecer um engano que um sistema Unix possa ser modular. Mas o sistema operacional da Sun, o Solaris, é um contra-exemplo, inclusive possui muitas semelhanças com o Windows 2000 no que se refere à divisão do sistema em módulos, cada um distribuído na forma de um arquivo binário pronto para uso.

O núcleo do sistema Solaris é agrupado em diversos componentes chaves e é implementado de uma forma modular. Os componentes chaves são os seguintes:

- **Interface das chamadas de sistema** - As chamadas de sistema permitem que programas tenham acesso às funções providas pelo núcleo do sistema. Esta camada direciona cada chamada de sistema para o módulo do núcleo apropriado.

- **Escalonamento e execução de processos** - A gerência de processos permite criar, executar, gerenciar e terminar processos e/ou linhas de execução. O escalonador permite que várias classes de escalonadores sejam carregadas para atender a requisitos de escalonamento diferentes.

- **Gerência de memória** - A gerência de memória é dividida em duas partes: uma parte comum à todas as arquiteturas e outra específica. Este modelo permite que novas arquiteturas sejam suportadas facilmente.

- **Sistemas de arquivos** - O Solaris permite que diversos tipos de sistemas de arquivos sejam utilizados pelo sistema. Sistemas de arquivos locais, de rede e pseudo sistemas de arquivos são implementados nesta camada.

- **Gerência de dispositivos e barramento de entrada/saída** - O modelo adotado pelo sistema Solaris para entrada/saída representa os dispositivos e os barramentos

de entrada/saída numa hierarquia de módulos refletindo o que acontece na parte física do computador.

- **Funcionalidades do núcleo do sistema** - Funcionalidades centrais do núcleo do sistema como interrupções, primitivas de sincronização e carregamento de módulos do sistema.

- **Rede** - Suporte aos protocolos de rede.

O núcleo do sistema Solaris é implementado como um grupo de funções essenciais, com serviços e subsistemas adicionais implementados como módulos do sistema. Isso é possível devido a existência de uma infraestrutura de carregamento e uso de módulos, que permite que módulos sejam adicionados ao sistema durante a iniciação do sistema ou por demanda enquanto o sistema está executando.

Como [Maur 01] afirma, o Solaris 7 suporta sete tipos de módulos do sistema: classes de escalonador, sistemas de arquivos, chamadas de sistema, carregadores de arquivos executáveis, fluxos (*streams*), drivers de dispositivos e de barramento e outros tipos de módulos. A tabela 4.1 mostra o que faz parte do núcleo do sistema e alguns exemplos de módulos do Solaris.

A iniciação do Solaris é muito parecida com a iniciação do Windows 2000 e pode ser resumida nas seguintes etapas:

1. O bloco de iniciação é lido e carregado em memória

2. O bloco de iniciação localiza e carrega o programa de *boot* secundário, **ufsboot**, e passa o controle para ele

3. **ufsboot** localiza e carrega o núcleo do sistema e passa o controle para o núcleo

4. O núcleo do sistema localiza e carrega os módulos do sistema e executa o código de iniciação do núcleo

5. O código de iniciação do núcleo cria e inicia as estruturas, os recursos e os componentes usados pelo núcleo

6. O sistema executa os *scripts* para terminar de iniciar o sistema como descrito no arquivo /etc/inittab

<table>
<tr><th>Núcleo do sistema</th><th>Tipo de Módulos</th><th>Exemplos</th></tr>
<tr><td rowspan="23">Chamadas de sistema
Escalonador
Gerência de memória
Gerência de processos
Virtual File System
Sincronização
Relógio e Timers
Gerência de interrupções
Iniciação
Gerência de CPU</td><td rowspan="3">Classes de escalonador</td><td>TS - Time share</td></tr>
<tr><td>RT - Real Time</td></tr>
<tr><td>Etc...</td></tr>
<tr><td rowspan="4">Sistemas de Arquivos</td><td>UFS - Unix File System</td></tr>
<tr><td>NFS - Network File System</td></tr>
<tr><td>PROCFS - Process File System</td></tr>
<tr><td>Etc...</td></tr>
<tr><td rowspan="4">Chamadas de sistema</td><td>shmsys - System V Shared Memory</td></tr>
<tr><td>semsys - Semaphrores</td></tr>
<tr><td>msgsys - Messages</td></tr>
<tr><td>Etc...</td></tr>
<tr><td rowspan="2">Carregadores de
arquivos executáveis</td><td>ELF - SVR4 Binary Format</td></tr>
<tr><td>COFF - BSD Binary Format</td></tr>
<tr><td rowspan="3">Fluxos (Streams)</td><td>pipemod - Streams Pipes</td></tr>
<tr><td>ldterm - Terminal Line Disciplines</td></tr>
<tr><td>Etc...</td></tr>
<tr><td rowspan="3">Outros</td><td>NFSSRV - NFS Server</td></tr>
<tr><td>IPC - Interprocess Communication</td></tr>
<tr><td>Etc...</td></tr>
<tr><td rowspan="4">Drivers de dispositivos
e de barramento</td><td>SBus - SBus Bus Controller</td></tr>
<tr><td>PCI - PCI Bus Controller</td></tr>
<tr><td>sd - SCSI I/O Devices</td></tr>
<tr><td>Etc...</td></tr>
</table>

Tabela 4.1: Funcionalidades do núcleo do sistema e exemplos de módulos

4.6 Conclusões

Os sistemas modulares são um meio termo entre os sistemas monolíticos e os sistemas micronúcleo, tendo a vantagem dos sistemas monolíticos, a velocidade, e a vantagem dos sistemas micronúcleo, a flexibilidade. Isso é possível porque o sistema é dividido em partes, os módulos, que, quando carregados em memória, se tornam parte do sistema e se comunicam com o resto do sistema através de simples chamadas de função.

Essa arquitetura mostrou ser viável e foi adotada por um dos sistemas operacionais mais populares do mundo: o Windows NT e seus sucessores Windows 2000 e Windows XP da Microsoft. Mesmo no ambiente dos sistemas operacionais tipicamente monolíticos (os sistemas Unix), um sistema modular foi construído e também é um caso de sucesso: o Solaris.

Percebendo o sucesso desses sistemas modulares, muitos sistemas monolíticos começaram a ser modificados para tirar proveito dessa arquitetura. Entre eles podemos citar o Linux e o FreeBSD.

Com a crescente utilização de dispositivos de *hardware* que podem ser inseridos ou retirados do sistema sem que o computador tenha que ser reiniciado, cresce também a necessidade por sistemas operacionais mais flexíveis e que possam ser reconfigurados sem a necessidade de reiniciar. Essa é uma tendência atual e os sistemas operacionais que não a seguirem vão perder mercado e usuários.

Capítulo 5

ModulOS

5.1 Introdução

As primeiras idéias que levaram ao desenvolvimento do ModulOS surgiram no final de 1997 quando eu era um estudante de graduação do curso de ciências da computação e estava terminando o meu primeiro ano na universidade. O interesse no desenvolvimento de um sistema operacional novo a partir do zero apareceu com a frustração que era utilizar o sistema operacional Windows 95 com seus inúmeros defeitos (sem que eu pudesse alterar alguma coisa) e, mais tarde, com a mesma frustração de usar o Linux (naquele tempo um Red Hat 4.2), um sistema com núcleo monolítico difícil de entender e modificar.

Tendo os dois sistemas operacionais que conhecia me decepcionado e sem um sistema operacional que tivesse o que eu queria (naquele tempo eu não investiguei a fundo se existiam outros sistemas operacionais), decidi que seria uma boa idéia fazer um sistema operacional diferente dos que eu conhecia.

Nesse tempo os meus conhecimentos sobre sistemas operacionais eram muito limitados, mas mesmo isso não me intimidou e eu pensava em fazer o melhor sistema operacional do mundo! Foram tempos de muitas idéias e algoritmos, mas nada de código.

Após vários anos experimentando diferentes desenhos para o sistema, muita experiência e conhecimento adquiridos, a primeira versão capaz de ser iniciada de um disquete estava funcionando em 2000. Nesse tempo eu ainda não tinha uma visão muito clara do sistema e tudo que tinha feito não era baseado nas idéias de outros sistemas

operacionais. Todos os algoritmos eram feitos do nada com o conhecimento adquirido nos livros sobre o processador Intel e na internet.

Com o passar do tempo e dos vários livros sobre sistemas operacionais lidos, uma visão mais clara do sistema e de suas diferenças foi surgindo. Com essa visão mais clara do sistema e sabendo o que o sistema era e o que ele precisava, muitas idéias de outros sistemas operacionais, especialmente Linux e Windows 2000, foram utilizadas. Entre essas idéias estão o modo de organizar as áreas de memória de um processo inspirado no Linux, o processo de iniciação do sistema inspirado no Windows 2000 e a utilização de asserções para verificar o sistema enquanto ele está executando inspiradas no FreeBSD.

Com essa nova filosofia de usar as boas idéias dos outros sistemas operacionais, o ModulOS avançou muito e passou a suportar muitas funções existentes em outros sistemas operacionais, permitindo que programas avançados pudessem ser feitos ou portados para ele.

Esses avanços foram possíveis depois que a grande diferença do ModulOS para os outros sistemas operacionais ficou clara: a extrema modularidade e a forma de relacionamento entre os módulos com a utilização de interfaces. Consciente dessas diferenças, o desenvolvimento do ModulOS ficou muito mais fácil e todas as boas idéias dos outros sistemas operacionais puderam ser adaptadas para tirar proveito dessas diferenças.

Uma mudança que trouxe um grande ganho em produtividade e que permitiu organizar muito melhor o que tinha que ser feito foi a utilização do SourceForge (http://sourceforge.net/), um site especializado em manter projetos de programas, fornecendo uma infraestrutura muito boa (listas de discussão, fóruns, CVS, tarefas a serem feitas, etc.).

Além disso, um programa que foi muito importante para o desenvolvimento do ModulOS foi o emulador da arquitetura Intel chamado Bochs (http://bochs.sourceforge.net/). Esse programa emula um computador compatível com os PCs, permitindo que um sistema operacional seja executado como se fosse um programa. Possibilita ainda que o estado do processador e da memória desse computador virtual seja analisado, fornecendo informações indispensáveis para a descoberta de certos problemas, principalmente no gerenciamento de memória.

No ModulOS não existe um núcleo do sistema que fornece as primitivas básicas do sistema. Existe, sim, alguns módulos que são essenciais para o funcionamento do sistema, mas que nem por isso deixam de ser módulos. Alguém poderia dizer que esses módulos

são o núcleo do sistema, mas esse termo "núcleo do sistema" passa uma idéia de sistema monolítico sem flexibilidade e, por isso, não é usado.

Atualmente o ModulOS possui 40 interfaces e 22 módulos, suportando, entre outras coisas, o sistema de arquivos ext2, o teclado padrão do PC com suporte a diferentes *layouts* e podendo executar diversos programas.

A figura 5.1 mostra uma visão geral do sistema, com os módulos do usuário (programas e bibliotecas) e os muitos módulos do sistema.

5.2 Iniciação

Como foi visto na seção 4.2, a iniciação de um sistema modular é complexa e sujeita a falhas. Por isso, tem que ser planejada com muito cuidado para suportar os diversos cenários e módulos, já que muitos módulos que ainda não existem na hora desse planejamento podem ser usados no futuro.

Sabendo da complexidade e da dificuldade que é criar um processo de iniciação, foi decidido que no início seria utilizado um processo de iniciação simples e rápido de ser desenvolvido. Por isso, a versão atual do ModulOS utiliza um processo de iniciação muito parecido com os sistemas monolíticos, ou seja, um arquivo binário, chamado de arquivo de iniciação, com todos os módulos necessários para a iniciação do sistema é carregado para a memória e executado.

Antes que o sistema possa ser iniciado um arquivo chamado de arquivo de iniciação deve ser criado. Esse arquivo é um arquivo binário com todos os módulos necessários para que o ModulOS possa ler do disco e com algumas configurações necessárias para que certos módulos possam ser iniciados. Esse arquivo é uma imagem do sistema na memória, ou seja, com as estruturas de gerenciamento de memória já criadas, com as relocações dos módulos já feitas, enfim com tudo pronto para o sistema executar.

Os passos do processo de iniciação são os seguintes:

1. O setor de iniciação (*boot sector*) carrega e passa o controle para o arquivo de iniciação

2. O arquivo de iniciação inicia o processador, cria um contexto de execução em 32 bits e passa o controle para a função que inicia os módulos

3. A função que inicia os módulos faz alguns malabarismos para possibilitar a iniciação dos módulos e, então, inicia os módulos

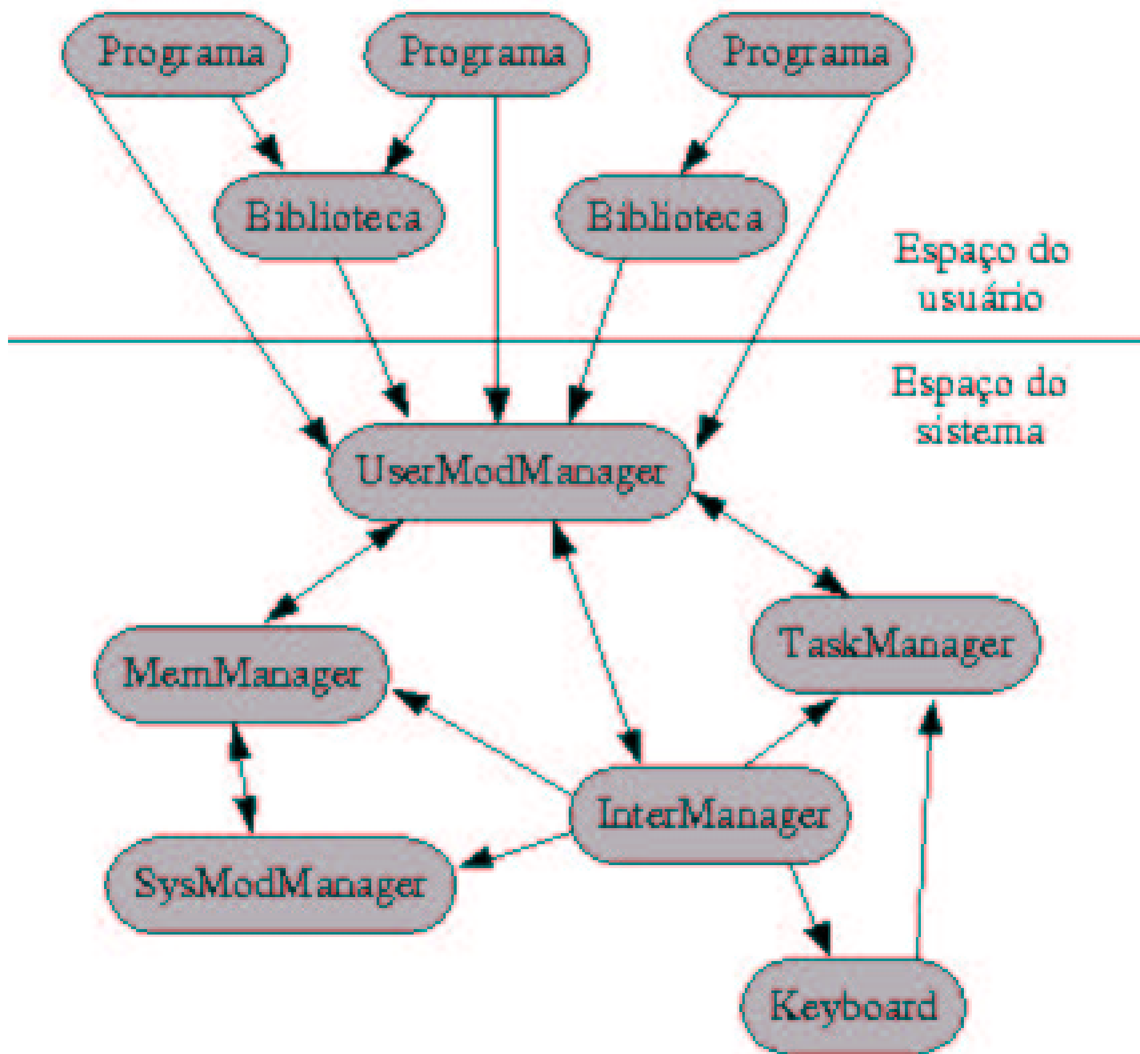

Figura 5.1: Visão geral do ModulOS

Esse processo de iniciação apesar de ser simples de implementar é muito difícil de manter. Cada vez que um módulo que deve estar no arquivo de iniciação muda, um novo arquivo de iniciação deve ser feito. Esse cenário piora porque não existe um processo automático para saber quais módulos devem ser incluídos no arquivo, o que exige um conhecimento profundo do sistema para o processo de iniciação funcionar.

Além disso, certas características muito específicas, como a localização de algumas variáveis na área de dados de um módulo, são necessárias na construção do arquivo de iniciação, o que torna o processo de iniciação muito complicado de manter e de ser feito por um usuário sem conhecimento de programação.

Tendo aprendido mais sobre os diversos processos de iniciação dos sistemas operacionais, principalmente dos sistemas modulares, um novo processo de iniciação inspirado no processo de iniciação do Windows 2000 e do Solaris está sendo planejado e implementado para o ModulOS. Junto com esse novo processo de iniciação, um novo módulo para centralizar e padronizar o acesso às configurações do sistema está sendo feito também, já que como foi visto no processo de iniciação do Windows 2000, as configurações do sistema são necessárias para saber quais módulos carregar.

Com esse novo processo de iniciação, o ModulOS ficará mais fácil de manter e de desenvolver.

5.3 Finalização

Por estar num estado inicial de desenvolvimento onde as mudanças na arquitetura do sistema ocorrem com muita frequência, o desenvolvimento da finalização do sistema não foi feito.

Atualmente os sistemas de arquivos somente aceitam operações que não modifiquem os dados do disco. Portanto, não existe o problema de perda de dados se o sistema não for finalizado corretamente e, assim, a finalização do sistema não se mostra muito importante.

No estado atual do desenvolvimento, o esforço está sendo direcionado para fazer o sistema iniciar e funcionar corretamente, não existindo função para finalizar ou reiniciar o sistema.

5.4 Desenvolvimento do sistema

O sistema operacional ModulOS é organizado em módulos que se inter-relacionam para fazer o sistema funcionar. Fazendo uma analogia com o corpo humano, cada módulo do sistema seria um órgão do corpo humana realizando tarefas e provendo serviços para os demais órgãos. Assim como os órgãos do corpo humano têm uma interface para prover esses serviços, os módulos do sistema também disponibilizam os seus serviços através de uma interface padronizada e documentada. Por utilizar uma interface padronizada, os módulos dependem **do que** é disponibilizado pelo módulo e não de **como** o módulo é implementado.

Essa característica é muito importante porque torna o sistema muito mais flexível, já que permite a substituição ou atualização de um módulo sem que necessariamente todo o sistema precise ser atualizado. Por exemplo, um módulo pode ser substituído por outro que disponibilize os mesmos serviços só que com uma implementação melhor e isso não implicará na atualização de todo o sistema.

Essa dependência de implementação é muito comum nos sistemas monolíticos, onde a mudança de uma biblioteca força todos os programas que utilizam essa biblioteca a serem recompilados ou, em alguns casos, força a instalação de uma biblioteca de compatibilidade, mesmo que as funções utilizadas não tenham mudado, isto é, mesmo que a interface utilizada não tenha mudado.

Além da flexibilidade conseguida com a utilização de uma interface padrão, outra tarefa muito difícil que é o trabalho de depuração e testes dos módulos se torna mais fácil e rápido de ser realizado, já que os pontos de comunicação entre os módulos são bem conhecidos e documentados.

Com a utilização de interfaces padronizadas e documentadas, o desenvolvimento de módulos para o sistema se torna muito mais fácil, já que todas as funções que o módulo deve disponibilizar já estão definidas e documentadas na interface, não existindo o trabalho de desenvolvimento e planejamento das funções.

Além dessa, existe outra vantagem oferecida pelo modelo adotado que é o desenvolvimento de módulos paralelamente. Tendo as interfaces já definidas e documentadas, o desenvolvimento dos módulos que vão implementar essas interfaces pode seguir paralela e independentemente dos demais módulos, trazendo um aumento na velocidade de desenvolvimento.

5.4.1 Interfaces

Uma interface nada mais é do que uma especificação das estruturas de dados, funções e do comportamento do módulo que implementar essa interface. Na especificação das funções da interface é detalhado o que cada uma faz, as suas restrições, os seus parâmetros, o que retorna e os possíveis erros. Note que é especificado **o que** a função deve fazer e não **como** ela deve fazer.

A idéia por trás do uso de interfaces foi porque elas permitem que um módulo seja utilizado mesmo que o módulo não seja conhecido por quem o use. Basta saber se o módulo implementa ou não a interface necessária.

Essa idéia também é usada na linguagem de programação Java. Nessa linguagem, se uma classe especifica que implementa uma determinada interface, isso significa que vários métodos definidos na interface são implementados pela classe. Com isso, uma função pode especificar que um determinado parâmetro tem que ser um objeto de uma classe que implemente a interface List, por exemplo, não importando que classe seja essa.

O primeiro passo no desenvolvimento de um módulo para o sistema é saber se já existe uma interface para o serviço que o módulo vai disponibilizar. Se a interface já existe, basta seguir a especificação e implementar o módulo.

Não existindo uma interface para os serviços do módulo, uma nova tem que ser criada e, para isso, um nome para a interface tem que ser definido. Por exemplo, SysModManager, TaskManager, etc. Isso porque os módulos usam o nome da interface para especificar qual interface estão usando.

A escolha de nomes de interfaces e não números, por exemplo, foi porque isso facilita o desenvolvimento de novas interfaces e torna mais difícil a existência de um conflito de nomes. Se fosse usado números, um cuidado muito grande teria que ser tomado para evitar que interfaces diferentes tivessem o mesmo número. Isso é feito, por exemplo, com os dispositivos PCI que têm que registrar o seu número de identificação numa organização para então poder ser usado.

Conhecido o nome da interface, as funções dessa interface devem ser especificadas. Ao contrário das interfaces que possuem um nome e é através desse nome que os módulos especificam qual interface estão usando, as funções das interfaces são especificadas através de números já que elas são conhecidas pelos módulos que as utilizam e, por isso, não existe risco de conflito.

Este é um exemplo de especificação de uma função da interface TextVideo que

especifica uma interface utilizada para mostrar informações numa tela no modo texto (onde só é possível mostrar texto):

`0x02` **- GetModeDesc**

Sintaxe

```
int TextVideo_GetModeDesc(unsigned int mode,
TextVideo_ModeDesc *desc);
```

Propriedades

Do sistema.

Descrição

Esta função obtém as características do modo *mode*.
Em caso de sucesso *desc* é preenchido com as características do modo.

Valor de retorno

* E_OK - Sucesso
* E_BAD_VALUE - Parâmetro inválido
* E_BAD_INDEX - Modo inválido

Após feita a especificação da interface, um arquivo .h utilizado pelos módulos que utilizam a interface deve ser criado. Os arquivos .h são utilizados na linguagem de programação C/C++ para armazenar especificações de estruturas, classes e protótipos de funções.

Por exemplo, este é o arquivo .h da interface TextVideo:

```
/*
*****************************************
* Interface TextVideo
*
* Author: Luiz Henrique Shigunov
* E-mail: shigunov@inf.ufsc.br
```

```
* WEB: http://www.inf.ufsc.br/~shigunov
* Last change: 06/03/2002
*
* See the interface specification to know what each
function does.
*
* Copyright (C) 2000-2002, Luiz Henrique Shigunov
*****************************************
*/
#if !defined(__TEXTVIDEO_H)
#define __TEXTVIDEO_H
#define TextVideo__CLEAR 0x00
#define TextVideo__GETCURRENTMODE 0x01
#define TextVideo__GETMODEDESC 0x02
#define TextVideo__GETSTATUS 0x09
#define TextVideo__SCROLL 0x03
#define TextVideo__SETATTRIB 0x04
#define TextVideo__SETCURSORPOS 0x05
#define TextVideo__SETCURSORTYPE 0x06
#define TextVideo__SETMODE 0x07
#define TextVideo__WRITE 0x08
typedef struct {unsigned int rows;unsigned int cols;
int flags;} TextVideo_ModeDesc;
typedef struct {unsigned int row;unsigned int col;
int attrib;int cursorType;} TextVideo_Status;
void TextVideo_Clear(unsigned int,unsigned int,unsigned
int,unsigned int);
unsigned int TextVideo_GetCurrentMode(void);
int TextVideo_GetModeDesc(unsigned int,
TextVideo_ModeDesc*);
int TextVideo_GetStatus(TextVideo_Status*);
int TextVideo_Scroll(unsigned int,unsigned int,unsigned
int,int);
void TextVideo_SetAttrib(int);
```

```
void TextVideo_SetCursorPos(unsigned int,unsigned int);
void TextVideo_SetCursorType(int);
int TextVideo_SetMode(unsigned int);
int TextVideo_Write(const char*);
#endif  /* __TEXTVIDEO_H */
```

Como a linguagem usada para programar os módulos do sistema é a linguagem C e para que não exista conflitos de funções e tipos de dados, cada função ou tipo de dados de uma interface deve ser precedido do nome da interface. Assim, a função Scroll da interface TextVideo fica TextVideo_Scroll.

5.4.2 Módulos do sistema

O módulo é a unidade básica do sistema e é dividido em código, dados somente para leitura, dados que podem ser modificados, informações sobre as interfaces, implementações e funções utilizadas e informações sobre as interfaces e suas funções implementadas pelo módulo, sendo que cada módulo do sistema tem de implementar uma ou mais interfaces.

Além disso, para diferenciar duas implementações de uma mesma interface, cada implementação tem que ter um nome para identificá-la. Por exemplo, a implementação da interface **UserModManager** poderia se chamar **i386**, fazendo referência a arquitetura para qual foi implementada.

Como um módulo pode implementar mais de uma interface e uma mesma interface pode ter várias implementações, fica fácil agrupar diversas implementações num único módulo. Por exemplo, um único módulo poderia implementar uma interface de criptografia para os diferentes algoritmos existentes.

Como na imensa maioria dos sistemas operacionais existentes, os módulos do sistemas são implementados utilizando-se a linguagem de programação estruturada C. Por não ser uma linhagem baseada em objetos ela oferece um grande controle sobre o que está sendo feito e isso é fundamental na implementação de um sistema operacional. Porém, a utilização dessa linguagem implica no uso de construções não muito elegantes como por exemplo o que foi mostrado para os nomes de funções e tipos de dados que devem ser precedidos pelo nome da interface para evitar conflitos entre interfaces.

Fora esse detalhe dos nomes de funções e tipos de dados, o desenvolvimento de

um módulo não tem mistérios e consiste basicamente em implementar as funções e o comportamento definido na interface.

Qualquer módulo utilizado por um módulo deve ser iniciado explicitamente usando-se a função **StartModule** da interface **SysModManager** antes de ser utilizado. Essa iniciação explícita foi escolhida porque permite ao módulo que solicitou a iniciação receber um código de erro no caso de algo sair errado na iniciação e, com isso, tomar as medidas necessárias.

Como não é possível saber em qual módulo uma interface foi implementada e, assim, saber qual módulo iniciar, a função **StartModule** da interface **SysModManager** recebe como parâmetro o ponteiro para uma função. É por isso que toda interface utilizada por um módulo deve ser usada como parâmetro para essa função.

Este é um exemplo de iniciação dos módulos utilizados pelo módulo que implementa a interface TextVideo:

```
if ((ret = SysModManager_StartModule((unsigned int)
      MemMan_Move))||
     (ret = SysModManager_StartModule((unsigned int)
      MemManager_i386_Catch))) {
     DEBUG_MSG_PRE(ret);
     goto returnEnd;
}
```

Como se nota, os módulos que implementam a interface **MemMan** e a interface **MemManager_i386** são iniciados. Se estas duas interfaces estiverem no mesmo módulo nada de mais acontecerá e o módulo será iniciado apenas uma vez.

Outro aspecto do desenvolvimento dos módulos que merece ser destacado é o suporte à depuração. Como este é um sistema operacional novo, existem poucas ferramentas para depurar os módulos do sistema e, por isso, a depuração é feita basicamente utilizando uma função que imprime na tela dados - o conhecido método de depuração “printf”.

Para facilitar e padronizar essas mensagens de depuração, algumas macros foram criadas: **DEBUG_MSG** - imprime qualquer mensagem passada como argumento e **DEBUG_MSG_PRE** - imprime uma mensagem pré-determinada. Essas macros utilizam a função **DebugMsg** da interface **SysModManager** para imprimir a mensagem de depuração.

Além de facilitar e padronizar as mensagens de depuração, essas macros permitem que essas mensagens sejam inseridas ou retiradas do código facilmente via a utilização de *ifdefs*, que condicionam a inserção de determinado código na compilação se uma condição for satisfeita. A macro **DEBUG_MSG_PRE**, por exemplo, é definida da seguinte maneira:

```
#ifdef DEBUG
#define DEBUG_MSG_PRE(errCode) SysModManager_DebugMsg
(__FILE__ ": debug point at %x: %x caller: %x\n",
__LINE__, errCode, __builtin_return_address(0))
#else
#define DEBUG_MSG_PRE(errCode) do {;} while(0)
#endif
```

Como se nota, se **DEBUG** estiver definida, **DEBUG_MSG_PRE** vai imprimir uma mensagem na tela e, se não estiver, nada será inserido no código que usar **DEBUG_MSG_PRE**. Essa prática de utilizar macros para códigos que podem ser inseridos ou retirados da compilação é muito utilizada nos sistemas operacionais Linux e FreeBSD, pois torna o código mais fácil de ler, não existindo *ifdefs* por todo o código.

Porém, somente a utilização dessas macros para depurar o sistema não foi suficiente e uma ferramenta que se mostrou muito importante para o desenvolvimento do ModulOS foi o emulador Bochs. Sem essa ferramenta, certos erros seriam muito difíceis de descobrir e necessitariam de muitas reiniciações de um computador real.

5.4.3 Módulos do usuário

A utilização de módulos, interfaces e implementações de interfaces não se restringe somente aos módulos do sistema. Esses mesmos conceitos também são utilizados em outra parte do sistema: no código do usuário. O nome código do usuário faz referência ao modo não privilegiado no qual o processador executa esse código.

Nesta outra parte não privilegiada do sistema, os módulos são divididos em módulos executáveis e módulos bibliotecas e são chamados como um todo de módulos do usuário.

Os módulos executáveis são os equivalentes dos arquivos executáveis, os programas utilizados nos outros sistemas operacionais. Esses módulos não implementam interface alguma, ou seja, não fornecem serviços. Na verdade, esses módulos fazem uso dos serviços disponibilizados pelo outros módulos para realizarem o seu trabalho.

Assim como nos arquivos executáveis dos outros sistemas operacionais, os módulos executáveis têm um ponto de entrada que é a primeira função do módulo a ser executada. Essa primeira função normalmente tem a seguinte sintaxe:

```
void main(int argc, char *argv[], char *envp[]);
```

Onde os parâmetros **argc** e **argv** são utilizados para passar os argumentos do programa e o parâmetro **envp** é utilizado para passar as variáveis de ambiente (conjunto de *strings* no formato variável=valor).

Os módulos bibliotecas são os módulos que fornecem serviços para os outros módulos do usuário, sejam eles módulos executáveis ou mesmo outros módulos bibliotecas e são os equivalentes das bibliotecas compartilhadas encontradas na maioria dos sistemas operacionais (*shared libraries* nos Unixes e DLLs no Windows).

O propósito delas é diminuir o tamanho do módulo executável e aumentar a produtividade através da reutilização de código - se um módulo biblioteca já faz o que você precisa, basta usá-lo. Outro ponto que merece destaque é que quando um módulo biblioteca é atualizado todos os módulos que o utilizam automaticamente passam a desfrutar dessa atualização.

Assim como os módulos do sistema, os módulos bibliotecas também têm que implementar uma ou mais interfaces e é somente através dessas interfaces que outros módulos podem usar os serviços do módulo biblioteca. Exemplos de interfaces que os módulos bibliotecas implementam atualmente: **MemManager** - gerência de memória (as funções malloc() e free() utilizadas em programas C), **Semaphore** e **Mutex** - sincronização (semáforos e *mutex*), **Thread** - gerência de fluxos de controle ou *threads* (criar, destruir, etc.), entre outras.

A utilização dos módulos biblioteca pode ser feita de duas maneiras: estática ou dinamicamente. Na utilização estática, a especificação dos módulos bibliotecas usados é feita num arquivo de configuração utilizado pelas ferramentas elf2Xmod (ver a seção 5.5.1) e fica gravada numa seção especial do arquivo do módulo executável ou biblioteca.

Já na utilização dinâmica, várias funções da interface **UserModManager** têm de ser utilizadas para carregar, iniciar e obter as funções do módulo biblioteca, existindo, assim, apenas a utilização de ponteiros para função. Desta maneira, vários módulos podem ser utilizados, bastando que um parâmetro diferente seja passado para as funções da interface **UserModManager**.

A utilização de um módulo biblioteca tem que seguir vários passos. Antes que qualquer módulo biblioteca seja utilizado, ele tem que ser carregado e iniciado explicitamente pelo módulo do usuário que faz uso desse módulo biblioteca. Ou seja, antes que um módulo biblioteca seja usado, a função **ULoadLibrary** (que carrega um módulo biblioteca) e **UGetStartFunction** (que obtém o ponteiro para a função de iniciação do módulo biblioteca) da interface **UserModManager** precisam ser utilizadas.

Por fim, antes do módulo executável terminar, todos os módulos bibliotecas usados devem ser finalizados. Para isso, a função **UGetShutdownFunction** (que obtém o ponteiro para a função de finalização do módulo biblioteca) deve ser chamada.

Este é o código usado pelo interpretador de comandos feito para o ModulOS para iniciar as bibliotecas que ele usa:

```
if (UserModManager_ULoadLibrary((unsigned int)
    User_Exception_SetCallBack)) {
    InterManager_UTV_Write("Error loading Exception
    lib.\n");
    UserModManager_UExit(1);
}
start = UserModManager_UGetStartFunction((unsigned int)
        User_Exception_SetCallBack);
if (start && start()) {
    InterManager_UTV_Write("Error starting Exception
    lib.\n");
    UserModManager_UExit(1);
}
```

Os motivos da utilização deste método são os mesmo que levaram os módulos do sistema a serem explicitamente iniciados: permitir que o módulo executável ou biblioteca trate um possível erro de carregamento, iniciação ou finalização do módulo do usuário, mostrando, por exemplo, uma mensagem de erro.

Como os módulos executáveis ou bibliotecas podem usar tanto as interfaces implementadas por módulos do sistema quanto interfaces implementadas por módulos bibliotecas, mesmo que essas interfaces tenham o mesmo nome, torna-se necessário especificar se a função é de uma ou de outra interface.

Essa e outras informações necessárias para gerar o arquivo com o módulo usuário são especificadas num arquivo de configuração utilizado pelas ferramentas criadas para o projeto.

5.5 Ferramentas criadas para o projeto

Por utilizar alguns conceitos não utilizados nos outros sistemas operacionais (interfaces, implementações e funções de interfaces), algumas ferramentas precisaram ser desenvolvidas para que o ModulOS pudesse existir.

Essas ferramentas são utilizadas para fazer um disquete de iniciação, para gerar o arquivo com todos os módulos necessários para a iniciação do ModulOS e para gerar os diversos formatos de arquivos utilizados no sistema.

5.5.1 Geradores de arquivos de módulos

Tanto os módulos do sistema quanto os módulos do usuário utilizam interfaces, implementações e funções de interfaces, conceitos que não existem nos sistemas operacionais Unix ou Windows, por exemplo. Por isso, formatos de arquivos diferentes dos utilizados nesses sistemas tiveram que ser desenvolvidos para os módulos utilizados no sistema operacional ModulOS.

Como os módulos do sistema, executáveis e bibliotecas têm requisitos e necessidades diferentes, três formatos de arquivos foram criados e para gerar esses arquivos, três ferramentas distintas foram desenvolvidas: elf2mod, elf2emod e elf2lmod.

O nome dessas ferramentas faz referência ao trabalho que elas fazem: converter um arquivo no formato objeto ELF para o arquivo de módulo do sistema, executável ou biblioteca. O arquivo objeto ELF é utilizado por ser o padrão no ambiente de desenvolvimento utilizado (sistema operacional Linux e o compilador GCC). Além disso, esse formato de arquivo objeto é muito bem documentado, o que permitiu o desenvolvimento das ferramentas sem maiores problemas.

Essas três ferramentas têm um comportamento muito semelhante: convertem um arquivo objeto no formato ELF num arquivo de módulo de acordo com um arquivo de configuração que define vários aspectos do arquivo de módulo, entre eles: versão do módulo, funções de iniciação e finalização, interfaces implementadas, interfaces e implementações utilizadas, entre outras. Este é um exemplo de um arquivo de

configuração usado para gerar o módulo do sistema que implementa a interface TextVideo:

```
header {
"AT Text Video - Copyright (C) 2000-2002, Luiz Henrique
Shigunov (shigunov@inf.ufsc.br)"
0x26
0x0
"Start"
"Shutdown"
}
interface {
"TextVideo"
    implementation {
    "AT"
        "Clear"             0x1
        "GetCurrentMode"    0x1
        "GetModeDesc"       0x1
        "Scroll"            0x1
        "SetAttrib"         0x1
        "SetCursorPos"      0x1
        "SetCursorType"     0x1
        "SetMode"           0x1
        "Write"             0x1
        "GetStatus"         0x1
    }
}
usedfunctions {
"MemMan_Move"                    "MemMan"
""  0x03
"MemManager_i386_Catch"          "MemManager_i386"
""  0x03
"MemManager_i386_FreeException" "MemManager_i386"
""  0x07
"MemManager_i386_FreePages"      "MemManager_i386"
```

```
""  0x08
"MemManager_i386_MapPhysPages"  "MemManager_i386"
""  0x0a
"SysModManager_DebugMsg"        "SysModManager"
""  0x02
"SysModManager_StartModule"     "SysModManager"
""  0x0a
"TaskManager_P"                 "TaskManager"
""  0x0f
"TaskManager_V"                 "TaskManager"
""  0x1a
}
```

Como se nota esse arquivo de configuração é dividido num cabeçalho com informações do módulo a ser gerado (versão, funções de iniciação e finalização, propriedades, etc.), numa seção com as interfaces implementadas e, por fim, numa seção com informações sobre as interfaces, implementações e funções das interfaces utilizadas.

5.6 Estado atual do desenvolvimento

Atualmente o ModulOS já se parece bastante com um sistema operacional e não mais um simples carregador que mostra uma mensagem do tipo: Carregado!

Esse tipo de carregador é a primeira coisa que as pessoas interessadas em desenvolver um sistema operacional fazem. Somente depois disso essas pessoas começam a percebem o enorme trabalho que é desenvolver um sistema operacional moderno e completo e a grande maioria desiste logo nos primeiros momentos.

No momento, os principais subsistemas encontrados em sistemas operacionais conhecidos estão implementados: administrador de memória, administrador de processos, administrador de arquivos, administrador de entrada/saída, etc.

5.6.1 Administrador de memória

O modelo de memória não segmentado e paginado com cada processo tendo o seu mapeamento de páginas é utilizado pelo ModulOS. Nesse modelo, o controle de acesso ao espaço de endereços é feito pelo mecanismo de paginação, que permite controlar

quem pode ter acesso à uma página (usuário e sistema ou apenas o sistema) e que tipo de acesso (leitura ou escrita).

O espaço de endereços é dividido em duas partes: uma que muda de acordo com o processo que está executando (espaço do usuário) e outra com a memória do sistema (espaço do sistema) que só pode ser acessada por código privilegiado e que tem sempre o mesmo mapeamento independentemente do processo que executa (figura 5.2).

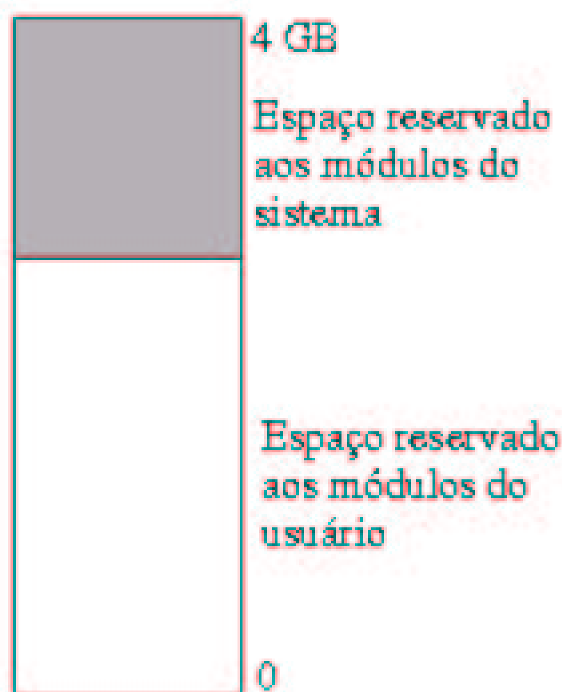

Figura 5.2: Espaço de endereços

Esse é o modelo utilizado pela maior parte dos sistema operacionais atuais porque é mais eficiente já que o mapeamento da memória do sistema sempre existe (uma chamada do sistema não precisa mudar o mapeamento de páginas), os processadores modernos suportam esse esquema (maior portabilidade) e também porque é muito simples de administrar.

A memória lógica utilizada pelos módulos do sistema é administrada utilizando-se o algoritmo *slab* - o mesmo usado pelo sistema Solaris. Já a administração das páginas do sistema utiliza um algoritmo mais simples: o *first fit*.

O administrador de memória também é responsável por administrar a memória lógica

do espaço do usuário, que é feita para cada processo através da alocação/liberação de regiões de memória com certos tipos de acesso (somente leitura, leitura/escrita ou sem acesso).

Esse mecanismo também é utilizado pelo Linux e permite uma grande flexibilidade para o módulo responsável pelo carregamento dos programas, que pode escolher onde e como carregar o programa e também para os próprios programas, que podem utilizar o seu algoritmo de administração de memória.

Além dos serviços de alocação de memória, existem outros tipos de serviços como o de memória compartilhada entre processos, mapeamento de páginas físicas, entre outros.

5.6.2 Administrador de processos

Assim como o administrador de memória, o administrador de processos é um dos módulos mais bem desenvolvidos e, por isso, fornece vários serviços. Isso é muito importante porque o administrador de memória e o administrador de processos, com os seus serviços, determinam como os outros módulos são feitos. Por exemplo, a decisão de utilizar ou não várias linhas de execução (threads) por processo influencia diretamente a arquitetura de vários módulos, obrigando os mesmos a utilizarem mecanismos de sincronização como os semáforos.

Seguindo a tendência dos sistemas operacionais modernos, o ModulOS conhece e utiliza o conceito de linhas de execução. Cada processo possui uma ou mais linhas de execução que compartilham o mesmo espaço de endereços do processo.

Como várias linhas de execução, do mesmo processo ou de processos diferentes, podem utilizar concorrentemente o mesmo serviço de um módulo, mecanismos de sincronização são necessários para evitar problemas de sincronização como, por exemplo, inconsistências nas estruturas de dados.

Atualmente, estão disponíveis semáforos, *spin locks* e sincronizadores do tipo leitor/escritor.

É comum existirem certos tipos de informação relacionados com os processos e/ou as linhas de execução. Por exemplo, os arquivos abertos pelo processo, as áreas de memória compartilhada, entre outras. Nos sistemas com núcleo monolítico, essas informações são definidas na própria estrutura de dados do processo e/ou linha de execução.

Porém, essa solução não é adequada no caso do ModulOS porque certos módulos podem ou não estar sendo usados. Um exemplo é o administrador de arquivos. Pode

ser que numa determinada configuração do sistema o administrador de arquivos não seja necessário e, assim, as suas estruturas relacionadas com os processos também não.

Para solucionar esse problema, o administrador de processos disponibiliza serviços que permitem que módulos armazenem dados relacionados com processos e/ou linhas de execução em suas estruturas.

No momento o administrador de processos utiliza o algoritmo de escalonamento conhecido por *round-robin* com fatia de tempo tanto para os processos quanto para as linhas de execução de cada processo. As fatias de tempo para os processos variam de acordo com sua prioridade, já as fatias das linhas de execução são fixas para todas as linhas de execução.

5.6.3 Administrador de arquivos

O modelo de administrador de arquivos adotado pela maioria dos sistemas Unix modernos é o vnode/vfs (*virtual node/ virtual file system*) desenvolvido pela Sun. Isso se deve em grande parte pelo modo orientado à objetos que o modelo foi feito.

Nesse modelo, um arquivo de qualquer tipo de sistema de arquivos é representado por uma estrutura de dados chamada vnode e um sistema de arquivos é representado por uma estrutura de dados chamada de vfs. Ambas estruturas possuem dados e funções conhecidos pelo administrador de arquivos e dados e funções conhecidos apenas pelos sistemas de arquivos. Ou seja, existe um encapsulamento de informações [Vaha 96].

Por ser um modelo orientado à objetos de muito sucesso, o ModulOS também o utiliza, mas com algumas adaptações.

No ModulOS, existe um módulo que administra os sistemas de arquivos e é responsável por repassar um pedido de serviço para o sistema de arquivos apropriado e os diversos módulos que implementam o acesso aos sistemas de arquivos (figura 5.3).

Assim como nos Unixes, cada sistema de arquivos para ser usado deve ser montado num diretório. Somente depois de estar montado é que os arquivos desse sistema de arquivos ficam disponíveis para serem utilizados.

Para cada arquivo ou diretório aberto existe uma estrutura de dados usada pelo administrador de sistemas de arquivos e outra usada pelo sistema de arquivos. Com isso, detalhes da implementação do sistema de arquivos ficam encapsulados e todos os sistemas de arquivos são utilizados da mesma forma.

Figura 5.3: Administrador de arquivos

5.6.4 Administrador de módulos do usuário

O administrador de módulos do usuário é o responsável pelo carregamento dos programas e das bibliotecas usadas pelos mesmos. Além disso, outra importante tarefa também é de sua responsabilidade: chamar as diversas funções do sistema.

Nos sistemas com núcleo monolítico e até em alguns sistemas modulares, cada chamada do sistema tem um número definido e conhecido. Como esse número e a forma de chamar o sistema muda de um sistema operacional para outro, as bibliotecas são utilizadas, principalmente a biblioteca C (libc) que é responsável por fazer a chamada do sistema.

Isso é possível porque cada função do sistema que um programa pode usar tem um número fixo e conhecido e todas ficam numa tabela chamada de tabela de chamadas do sistema, que associa o número da chamada do sistema com o ponteiro para a função.

Porém, no ModulOS os módulos podem disponibilizar para os programas quantas funções quiserem e podem ser carregados dinamicamente. Isso inviabiliza a utilização de uma tabela fixa com todas as funções do sistema.

Para resolver esse problema, o administrador de módulos do usuário é quem faz a

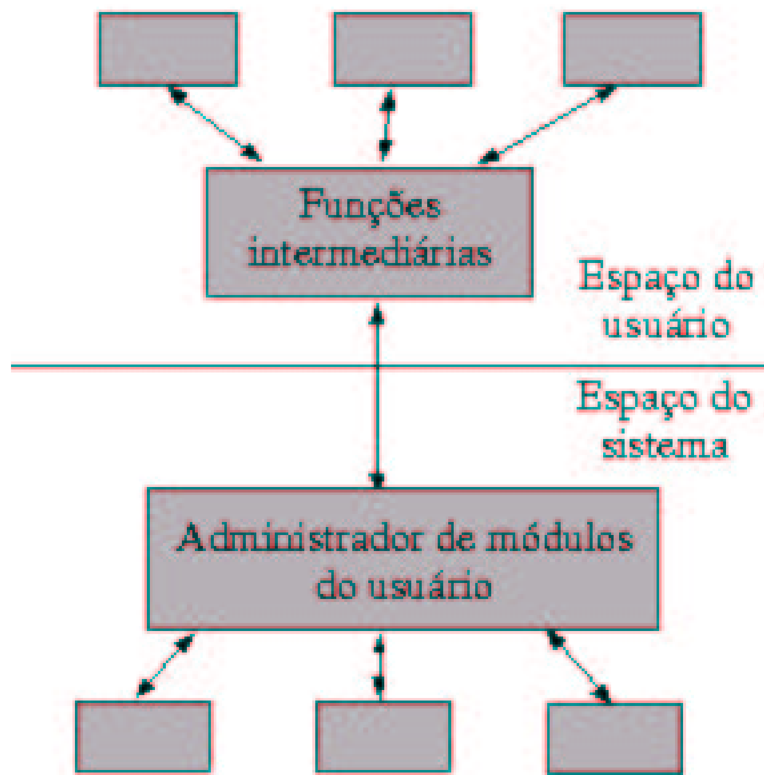

Figura 5.4: Chamada do sistema

chamada do sistema, ou seja, ele serve como um intermediário entre os módulos do usuário e os módulos do sistema.

Sempre que um módulo do usuário é carregado e este utiliza uma função de um módulo do sistema, o administrador de módulos do usuário cria uma entrada numa tabela com as chamadas do sistema e cria também dinamicamente uma função que usa o número dessa entrada para chamar a função do sistema.

Quando o módulo do usuário chama o sistema, a função criada pelo administrador dos módulos do usuário é chamada e esta, por sua vez, chama a função do sistema (figura 5.4).

Essa tabela com as entradas para as funções do sistema e as funções intermediárias criadas são compartilhadas por todos os módulos do usuário, minimizando a quantidade de memória utilizada.

5.7 Trabalhos futuros

Existe **muita** coisa para ser feita no ModulOS. Os primeiros anos de desenvolvimento do sistema foram de muita aprendizagem e somente agora os diversos conceitos usados e o papel de cada módulo no sistema estão ficando mais claros, possibilitando um desenvolvimento mais rápido e consistente.

Atualmente está em curso a mudança no processo de iniciação do ModulOS, o que vai permitir mais facilmente a atualização do sistema e a inclusão de novos módulos. Esse novo processo, por ser totalmente diferente do atual, incluindo novos requisitos para os módulos, implicou na alteração de todos os módulos já implementados. Isso não é um grande problema porque ainda não existem muitos módulos e porque essas mudanças vão trazer um ganho muito grande em termos de facilidade na instalação e atualização do sistema.

Uma das áreas que certamente precisa de melhorias é a parte de sistemas de arquivos. Os módulos e interfaces atuais tratam razoavelmente bem a leitura, mas não suportam escrita. Uma reavaliação e um planejamento é preciso para que a escrita seja suportada sem problemas. Além dessa grande mudança no modelo dos sistemas de arquivos, mais tipos de sistemas de arquivos precisam ser implementados, principalmente o suporte ao sistema de arquivos FAT usado no Windows.

Na área do gerenciamento dos módulos do sistema uma das coisas que precisa ser feita é a especificação e a implementação de quando e como os módulos devem ser retirados da memória quando não estão sendo usados. A questão de quando um módulo tem que sair da memória deve levar em conta que certos módulos podem ser usados por períodos de tempo não muito longos, mas que são usados freqüentemente. Nesse caso, mesmo que o módulo não esteja em uso ele deveria ficar em memória por um certo tempo.

Outra tarefa relacionada com o gerenciamento dos módulos que deve ser feita é o processo de finalização do sistema. Uma especificação de como finalizar o sistema tem que ser feita. Essa especificação tem que levar em conta vários aspectos como a ordem de finalização dos módulos do sistema. No momento não existe nenhum meio para finalizar o ModulOS.

A área da interface usuário/sistema já tem um bom modelo para dispositivos de entrada como o teclado, mas falta permitir que se troque o *layout* do teclado (por exemplo, de ABNT para US, etc.). Outro ponto é o suporte a diferentes código de páginas (para permitir mostrar acentos e até outras línguas como russo, por exemplo) na

interface TextVideo. Atualmente somente o código de página padrão é usado. E mais adiante, claro, suportar modo gráfico.

Um ponto que é um problema não só para o ModulOS como para a maioria dos sistemas operacionais é o desenvolvimento de *drivers* de dispositivos. O único sistema operacional que não tem esse problema é o Windows porque quase a totalidade dos dispositivos têm um *driver* para Windows. Mas muitos dispositivos suportados no Linux, por exemplo, podem e devem ser suportados no ModulOS também.

Uma melhoria que não é essencial, mas que traria alguma facilidade seria unificar todas as ferramentas de conversão de arquivos ELF para módulos (elf2mod, elf2emod e elf2lmod) numa única ferramenta que receberia um parâmetro dizendo qual arquivo de módulo gerar. Isso facilitaria a manutenção dessas ferramentas, já que hoje a correção de algum problema nessas ferramentas tem que ser feita nas três.

Capítulo 6

Conclusões

Ao longo desses anos que tenho trabalhado no ModulOS foi possível aprender muita coisa nova na área de sistemas operacionais e, claro, adquirir muita experiência no desenvolvimento de sistemas.

O desenvolvimento do ModulOS passou por diversas etapas que refletiam a minha experiência naquele momento. Assim, a linguagem de montagem foi usada no início, mas logo em seguida foi substituída pela linguagem C por ser mais fácil e melhor, o modelo de memória que era segmentado passou a ser não-segmentado, as boas idéias dos outros sistemas operacionais passaram a ser utilizadas, etc.

Como desde o início somente eu estou desenvolvendo o sistema, foi necessário aprender muitas coisas diferentes como sistemas de arquivos, gerenciamento de memória, dispositivos de hardware, entrada e saída, entre outras.

Esse aprendizado de coisas tão diferentes somado à minha falta de experiência fizeram com que o tempo de desenvolvimento do ModulOS fosse muito maior e que muitas mudanças na arquitetura do sistema fossem feitas. Porém, esse fato permitiu que hoje eu tenha um conhecimento muito bom sobre muitos sistemas operacionais diferentes.

O objetivo de construir um sistema operacional de propósito geral extremamente modular não foi conseguido na sua completude já que isso demandaria um esforço e tempo muito maior. Contudo, o que já está feito fornece uma base muito boa para que esse objetivo seja alcançado.

Apesar de não ser um sistema operacional completo, todos os objetivos específicos definidos foram alcançados, permitindo que o ModulOS seja utilizado em computadores padrão usando apenas um disquete com todo o sistema.

Referências Bibliográficas

[Bene 02] Beneden, Bart Van. A message-based, microkernel architecture. **Dr. Dobb's Journal**, Junho, 2002.

[Maur 01] Mauro, Jim, McDougall, Richard. **Solaris internals: Core kernel architecture.** California: Sun Microsystems Press, 2001. 657p.

[Maxw 99] Maxwell, Scott. **Linux core kernel commentary.** Estados Unidos: Coriolis, 1999. 576p.

[McKu 96] McKusick, Marshall K., et al. **The design and implementation of the 4.4BSD operating system.** Estados Unidos: Addison Wesley, 1996. 580p.

[Riel 02] Riel, Rik van. http://linux.html.it/articoli/rik_van_riel_en1.htm

[Solo 00] Solomon, David A., Russinovich, Mark. **Inside Microsoft Windows 2000.** Redmond: Microsoft Press, 2000. 903p.

[Vaha 96] Vahalia, Uresh. **Unix internals: The new frontiers.** New Jersey: Prentice-Hall, 1996. 601p.