# KERNEL NEUTRINO

**DOCUMENTAÇÃO v0.01**

**Autor: Alisson Linhares de Carvalho**

**Aracaju**
**2011**

# LISTA DE FIGURAS

# SUMÁRIO

# 1. INTRODUÇÃO

O projeto neutrino consiste em um agrupamento de conceitos sobre algo pouco explorado nas universidades brasileiras: o desenvolvimento de sistemas operacionais. Trata-se de um kernel multiprogramado, monothread, monolítico, preemptivo, com suporte a modo protegido e interface gráfica. Suportado pelas principais famílias de processadores da arquitetura x86, com ênfase nos processadores superiores à família 486 da Intel, o mesmo visa uma alternativa aos padrões de softwares existentes. Busca obter um maior poder de personalização e um ganho de desempenho em determinadas aplicações com o uso de técnicas avançadas de programação em assembly.

As abordagens utilizadas na construção do kernel possibilitarão que programas possam executar serviços diretamente sem o intermédio do núcleo e com mínimo esforço por parte dos usuários.

O kernel é distribuído sob a licença GNUv3 (General Public License version III). Desta forma, programadores poderão alterar e redistribuir de forma gratuita o núcleo, possibilitando a melhoria contínua desse software.

As bibliotecas de integração e interfaces são distribuídas sob a licença Lesser GNUv3. Permite que desenvolvedores proprietários possam alterar a licença original dessas bibliotecas para uso comercial. Desta forma será possível fechar o código das aplicações caso esses desenvolvedores, assim, decidam.

# 2. ARQUITETURA NEUTRINO

Um sistema operacional consiste em um conjunto de camadas de *software* inter-relacionadas. Praticamente, todos os sistemas analisados antes da elaboração deste documento, possuíam separações claras de suas estruturas, apresentavam camadas de *drivers*, bibliotecas, aplicativos etc. Na arquitetura idealizada para o neutrino não poderia ser diferente.

Na Figura 1 podemos visualizar a estrutura necessária para o funcionamento de um sistema operacional que faça uso do núcleo neutrino. Este núcleo desempenha o papel mais importante dessa arquitetura. Funciona como elo entre as demais camadas de *software*.


**Figura 1: Arquitetura necessária ao funcionamento do kernel neutrino.**

Para o *kernel* neutrino funcionar, é necessário que cada camada de software siga uma estrutura hierárquica bem definida. Os programas devem se comunicar com uma biblioteca de integração. Essa, por sua vez, com o *kernel* e uma biblioteca estendida.

Um dos objetivos deste documento consiste em descrever o inter-relacionamento entre as camadas, *kernel*, biblioteca estendida, biblioteca de integração e o *hardware*. Cada uma

destas quatro camadas apresentam certas peculiaridades, necessitando de uma explicação mais aprofundada.

## 2.1. Biblioteca de integração

A biblioteca de integração consiste em um conjunto de interfaces. Tem como objetivo tornar as aplicações independentes da versão do *kernel*. Esta biblioteca possui um papel importante nesta arquitetura, fornece os subsídios necessários para tornar o desenvolvimento de *software* mais produtivo. Foi completamente escrita em C++, possibilitando um maior reaproveitamento de código.

Esta camada fornece uma abstração das estruturas básicas do núcleo. Todo programa necessita ser compilado juntamente a essa biblioteca, caso contrario, o sistema não será capaz de executa-lo. A biblioteca de integração foi projetada para consumir o menor espaço possível da aplicação e possui em suas chamadas um segundo nível de validação dos dados que trafegam entre o *kernel* e o programa.

## 2.2. Biblioteca estendida

A biblioteca estendida atua como *drivers* projetados por terceiros. Tem como objetivo deter codificações que não estão presente no *kernel*. Depende diretamente da estrutura e serviços desempenhados pelo sistema operacional e pode se comunicar diretamente com o *hardware* sem o intermédio do núcleo. Suas principais metas são o aumento da robustez e versatilidade da plataforma.

Esta camada é similar à biblioteca de integração, porém o núcleo não possui conhecimento das suas estruturas. Deve ser usada somente em situações em que o *kernel* não for capaz de atender os requisitos da aplicação.

Um grande benefício trazido por esta camada é o fato que um sistema operacional pode apresentar varias características diferentes entre suas aplicações, pois esta biblioteca é auto gerenciável. Em contra partida, o maior beneficio se torna o maior defeito, o sistema pode ser completamente comprometido em caso de um erro nesta biblioteca.

## 2.3. Camada Kernel

O núcleo foi construído de forma parcial sobre o paradigma *monolítico*, usado em uma grande variedade de sistemas operacionais. O *kernel* tem como objetivo ser o mais genérico e simples possível. Oferece apenas o essencial ao gerenciamento de programas.

No *kernel* neutrino, todos os programas possuem uma abstração de computador real, porém, de forma personalizável. Cada programa pode determinar em qual nível de abstração o seu processo trabalhará. O programador possui o direito de escolher entre níveis mais elevados, com o uso das bibliotecas de integração ou mais baixos, escrevendo códigos diretamente. Essa característica traz como principal vantagem o ganho de personalização por parte do sistema. Torna possível a construção de sistemas especialistas que necessitem de máximo desempenho do hardware, pois é possível incorporar *drivers* específicos ao funcionamento dos programas.

Infelizmente, erros fatais podem ser gerados com acessos descoordenados ao *hardware*. Para solucionar este problema idéias retiradas da arquitetura *exokernel* foram

incorporadas a plataforma. Apesar da possibilidade das escolhas dos níveis de abstração, o sistema oferece um conjunto de bibliotecas de integração, que são recomendadas para o correto funcionamento de todo o sistema.

Outra característica marcante da plataforma é o fato dos programas possuírem modelos de arquivos *flat binaries*. Cada programa assemelha-se a um executável *bootável* com área de inicialização posterior aos primeiros 4096 bytes. A comunicação com o hardware é feita por interrupções de forma análoga a uma BIOS (*Basic Input Output System*) de computador, aparentando um sistema mais simples, porém, virtualizado.

Os programas se comunicam com as interfaces de chamadas de sistemas, cujas únicas funções, são mudar a base de endereçamento de memória e efetuar as chamadas procedurais às funções dos gerentes do núcleo. Tudo é feito de forma hierárquica. Cada procedimento pode apenas se comunicar com funções do mesmo nível de endereçamento. As interrupções controlam as trocas de informações entre os programas e os módulos. Porém, em virtude da liberdade de programação oferecida pelo sistema, programas podem forçar essas chamadas sem o intermédio do mesmo.

Pode-se dizer que o neutrino possui algumas características similares ao paradigma modular. O *kernel* é minimalista e é responsável por trocar informações entre os módulos projetados. O núcleo é compilado de forma única. Entretanto, funções que não são oferecidas podem ser compiladas de forma separada e executadas, oferecendo assim, novos serviços.

O núcleo se subdivide em seis módulos distintos, denominados gerentes. Cada gerente possui um conjunto específico de funções básicas que podem ser usadas por programas. Os gerentes podem ser visualizados na Figura 2.

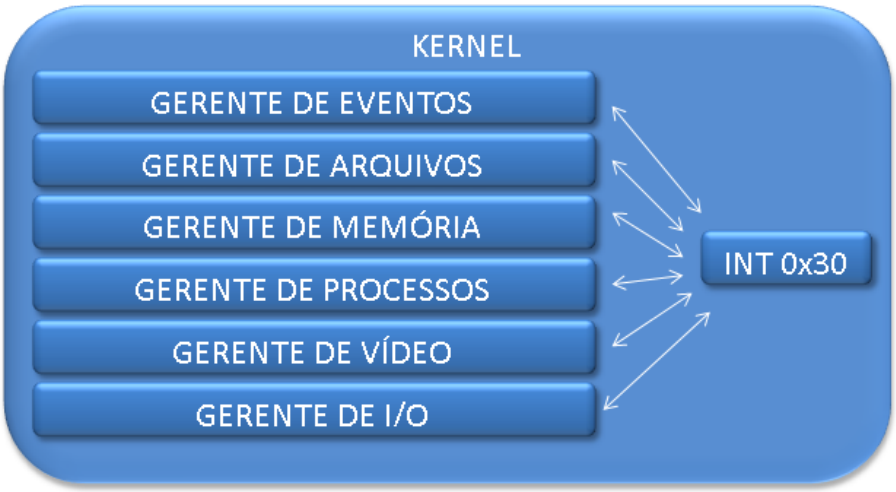


**Figura 2: Gerentes do kernel**

Todas as chamadas aos gerentes são feitas por software com o auxílio da interrupção 0x30. O processamento de eventos e determinadas tarefas são executados por programas em níveis de acesso diferentes. Todo o sistema possui um conjunto de drivers genéricos projetados para possuir 100% de integração com os gerentes.

O neutrino trabalha em modo protegido. Contudo, para um maior poder de personalização do sistema, a plataforma funciona em *Ring*0. É possível recompilar o módulo principal para que programas trabalhem em *Ring*3 apenas mudando algumas diretivas. Contudo, toda a capacidade de personalização dos programas é perdida e os mesmos poderão apenas utilizar funções oferecidas pelo núcleo.

### 2.3.1. Gerente de processo

Na arquitetura neutrino o gerente de processos consiste em uma lista circular duplamente encadeada. A lista é formada pelos BCPs (Blocos de Controle de Processo), usados para manter estatísticas, estados dos registradores e as tabelas de alocação de recursos

de cada programa. Cada bloco apresenta duas entradas constituídas por ponteiros: Uma contendo o endereço do próximo BCP da lista e outra a do anterior.

Cada processo está obrigatoriamente associado ao seu BCP, que se encontra em uma região especial da pilha de cada programa. Na Figura 3, podemos visualizar a estrutura deste gerente.

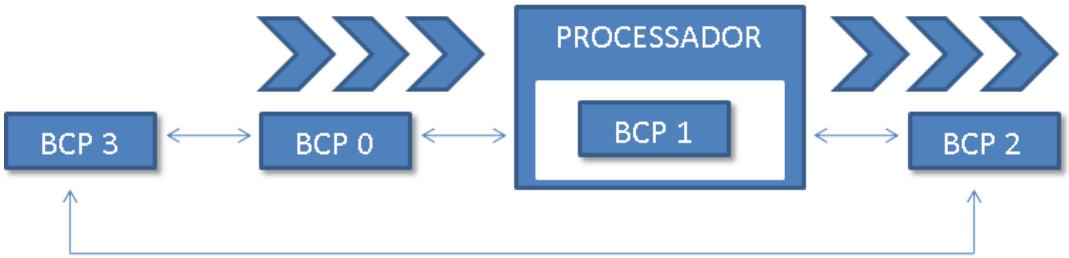


**Figura 3: Lista duplamente encadeada circular**

Cada processo pode executar dentro de um tempo limite de 18.2ms. Ao término do período de execução do processo é realizada uma ação denominada troca contexto, responsável por entregar a CPU (*Central Processing Unit*), ou UCP (Unidade Central de Processamento) a outro processo.

A política adotada para o escalonador é a mais justa dentre as bibliografias pesquisadas. Cada processo deve compartilhar a CPU de forma proporcional a necessidade de processamento. Processos *cpu-bound* tendem a usar 100% do *quantum*, enquanto processos *io-bound* tendem a executar durante uma ínfima fatia deste período de 18.2ms. Estas medidas beneficiam os processos *cpu-bound*, que tendem a receber a posse do processador mais rapidamente.

O escalonador é *preemptivo*, permite que um processo possa ser interrompido durante sua execução, possibilitando a entrega do processador a outro programa. O processo pode perder a posse do processador de duas formas distintas:

1. Se o processo fizer uma chamada a uma interrupção do *kernel* que requisite atividades interativas, *sleep*, entrada ou saída de dados e o núcleo não esteja apto a satisfazer a requisição, o programa entrará em estado de espera, perdendo assim, a posse do processador. Na Figura 4 podemos visualizar um trecho de código onde este conceito é aplicado.

```
1  #include "../../NLIB/console.h"
2
3  int main() {
4      Console *cmd = new Console(50,50,"Exemplo\0");
5      cmd->readkey(); //Processo dorme até que a entrada de dados seja realizada.
6      return 0;
7  }
```

**Figura 4: Exemplo de perda do processador**

2. Se o tempo do *quantum* destinado ao processo terminar é feita automaticamente uma troca de contexto.

Na arquitetura neutrino, existem basicamente três estados possíveis para um processo, similar ao encontrado nas principais bibliografias do assunto. Podemos visualizá-los detalhadamente na Figura 5.

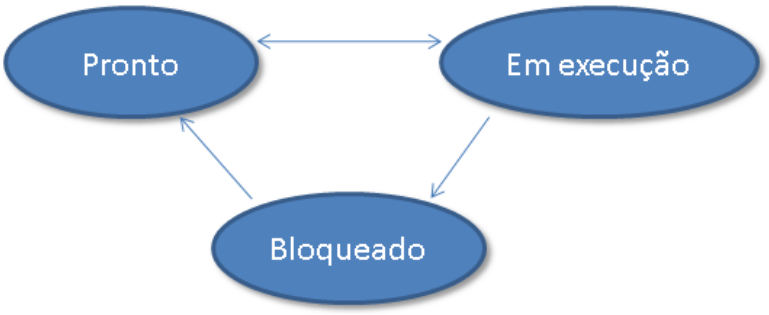


**Figura 5: Estados de um processo**

Quando um processo é iniciado, este é automaticamente inserido na lista circular de prontos e aguarda a posse do processador. Assim que o processador lhe é entregue, o processo entra em execução. Caso ocorra uma condição de bloqueio, o processo perderá a posse do processador durante o seu *quantum* e entrará em estado de espera.

O neutrino possui uma característica diferenciada: Um processo nunca é removido da lista circular de prontos. Quando um processo se encontra bloqueado, ao entrar em execução são efetuados pequenos testes e na sequência a posse do processador é perdida novamente. Somente uma porcentagem ínfima do *quantum* é executada. Essas medidas trazem como vantagem a redução do *starvation* e minimização do tempo de resposta a eventos.

Um ponto forte do escalonador usado está no fato do mesmo trabalhar de forma híbrida, funcionando tanto *monotarefa* como *multitarefa*. Quando existe apenas um processo em operação, o mesmo é executado na máxima capacidade de processamento. Isso se deve ao fato do sistema processar eventos durante o tempo de execução de cada processo.

O *kernel* neutrino não possui um processo. O escalonador e todos os outros gerentes são executados como serviços internos dos programas.

A abordagem utilizada desperdiça uma quantidade substancial de processamento com trocas de contexto, todavia, processos que necessitem de uma grande quantidade de eventos tendem a ser beneficiados com esta abordagem.

### 2.3.2. Gerente de memória

Na arquitetura neutrino existem duas camadas de gerência de memória, uma de aplicação e outra de núcleo. Ambas fazem parte do *kernel e* trazem como principais benefícios maior segurança e desempenho das aplicações.

A camada de gerenciamento de memória do núcleo é a camada mais baixa do sistema. Ela é responsável por verificar a disponibilidade de blocos consecutivos de memória e entregá-los aos programas. Usada por uma grande variedade de funções, tem como principal meta alocar grandes quantidades de blocos de memória.

A gerência de memória oferecida pelo *kernel* consiste em uma lista duplamente encadeada de blocos de memórias livres. Cada bloco pode apresentar o tamanho mínimo de uma página. Na arquitetura neutrino, uma página representa o espaço mínimo de armazenamento usado para gerenciamento de memória[1].

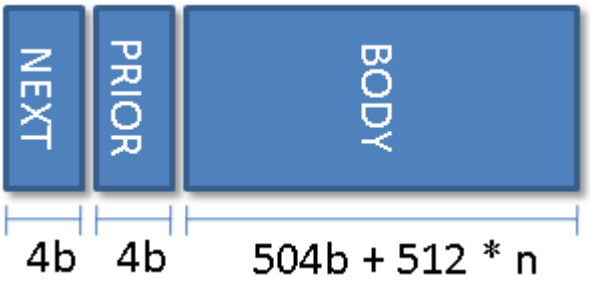


**Figura 6: Bloco de páginas**

Cada bloco de páginas é composto por três campos (Figura 6). O *next* consiste de um ponteiro de 32bits que aponta para o próximo bloco de páginas. O *size* é o número de páginas ocupadas pelo *body* mais 8bytes. O *body* consiste no espaço físico ocupado pela área de memória livre consecutiva.

[1] Atualmente este valor é de 512 bytes.

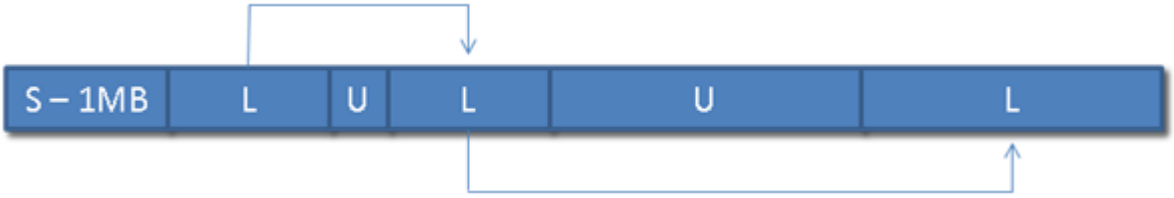


**Figura 7: Exemplo da memória durante o uso do sistema.**

Na Figura 7 é apresentado um exemplo da memória durante o uso da plataforma. O sistema possui três tipos de estados distintos para memória. O estado estático consiste da área de memória ocupada pelo núcleo. Esta região é imutável e consiste em sua grande maioria de códigos reentrantes. Possui o tamanho de 1mb e é compartilhado por todas as funções do *kernel*. O estado livre consiste de áreas de memória que não foram alocadas, essas áreas são agrupadas formando blocos de páginas consecutivas. Por fim, o estado usado representa as áreas de memória ocupadas por programas e seus respectivos dados.

As principais funções do gerente de memória de núcleo são as de alocação, liberação e expansão. Sempre que um processo necessita alocar uma página, o programa faz uma chamada ao gerente de memória passando como parâmetro o total de páginas que devem ser alocadas pelo sistema. Ao receber a solicitação o gerente efetua uma busca sequencial dos blocos de memória livres consecutivos como o objetivo de encontrar espaço suficiente para satisfazer a chamada. Ao final do processo, o sistema recalcula o tamanho do bloco onde foram removidas as páginas e verifica se o mesmo continua a existir.

Na chamada por liberação deve-se passar como parâmetros o total de páginas que serão liberadas e a posição de memória do bloco de páginas que será removido. Ao receber a solicitação, o sistema insere na lista de livres o espaço que antes estava ocupado, recalcula o tamanho dos blocos livres e verifica se é necessário reagrupar áreas de memória.

A expansão é um dos algoritmos de maior complexidade do sistema. Esta função é pouco usada. Porém, é vital para o gerente de memória de nível de aplicação, pois sempre que um processo não consegue alocar memória sequencialmente, é necessário fazer uma movimentação de dados entre áreas de memória. O funcionamento consiste de uma procurar por uma área suficiente para armazenar o processo mais o tamanho da necessidade da alocação. Ao final do processo é necessário mover todos os dados para a nova área.

O modelo de gerência de memória do núcleo possui as vantagens de consumir apenas 12 bytes para gerenciar toda a memória do sistema. Apresenta velocidades muito superiores a alguns dos principais modelos de gerenciamento de memória quanto à busca por páginas consecutivas. A adoção traz como principal benefício uma baixa fragmentação interna.

A camada de gerência de memória no nível de aplicação tem como objetivo coordenar a utilização da memória fornecida pelo núcleo. Tem como principal meta gerenciar níveis muito baixos de memória sendo utilizadas em procedimento como *malloc* e *free*.

O principal motivo de esta área estar presente no *kernel*, ao invés da aplicação, é a busca por maior desempenho e segurança de todo o sistema. Programadores modernos tendem a fazer grande quantidade de chamadas a funções como *new* ou *malloc*. A existência dessas funções no núcleo traz como benefício maior permanência desses códigos em *cache* e um menor tamanho dos programas. Como prejuízo, aplicações que necessitem alocar e liberar quantidades variáveis de memória com grande velocidade tendem a sofrer com queda de desempenho pelas chamadas de interrupção.

Os algoritmos de alocação e liberação são similares aos usados anteriormente e grande parte do código é compartilhado. A principal diferença está no tamanho da página, que no

gerente de memória de nível de aplicação possui oito bytes, buscando diminuir a fragmentação interna.


**Figura 8: Programa em execução.**

No nível de aplicação ainda existe o controle da pilha dos processos. Na arquitetura neutrino a pilha possui tamanho estático definido por padrão 3840 bytes e a área dinâmica pode crescer até ocupar toda memória disponível. Na Figura 8 podemos visualizar a organização da memória para programas.

### 2.3.3. Gerente de vídeo

O gerente de vídeo é o módulo responsável por controlar o funcionamento da interface gráfica do sistema neutrino. Geralmente essa camada é negligenciada pelas bibliografias relacionadas a sistemas operacionais e sua codificação é atribuída ao nível de bibliotecas fora do escopo do *kernel*. Na arquitetura apresentada, esta camada foi inserida no núcleo, oferecendo uma maior portabilidade entre aplicações que rodem em possíveis distribuições e maior controle sobre as ações do usuário.

A principal característica desse gerente é a possibilidade evolutiva das janelas. O gerente é completamente isolado das aplicações com o uso das bibliotecas de integração. Caso

o sistema e suas janelas evoluam, as aplicações acompanharão esta evolução e não necessitarão de recompilações.

As bibliotecas de integração exercem uma grande importância sobre o funcionamento deste gerente. Trabalhar com gráficos se torna mais simples, pois o programa necessita apenas efetuar chamadas correspondentes aos recursos que devem ser instanciados. Na Figura 9 podemos visualizar um trecho de código retirado de um dos programas de demonstração do sistema. Neste exemplo é criada uma janela capaz de *renderizar* vídeos, imagens e possivelmente jogos com o uso da *GraphicArea*. Todas as funções estão presentes nas bibliotecas de integração e fazem apenas as configurações e chamadas ao núcleo.

A codificação em alto nível pode ser observada na Figura 9. A organização foi inspirada na biblioteca *swing* do *Java*. A interface é minimalista e apresenta somente o fundamental ao gerenciamento de telas. O código entre a linha 43 a 56 é responsável por criar e configurar os componentes que serão renderizados. Da linha 59 em diante é criado um painel e adicionados os componentes criados anteriormente.

```
42      //Criando um botão "prior" e inserindo um tratador;
43      Button *prior = new Button(2,18,38,20,"prior\0");
44      prior->addEventHandler( priorAnim, Component::ON_MOUSE_DOWN_L );
45
46      //Criando um botão "next" e inserindo um tratador;
47      Button *next = new Button(42,18,34,20,"next\0");
48      next->addEventHandler(nextAnim,Component::ON_MOUSE_DOWN_L);
49
50      //Criando um label para renderizar o total de fps;
51      Label *fpsLabel = new Label(90,20,100,20);
52      fpsLabel->setName(fpsStr);
53
54      //Criando uma tela gráfica;
55      GraphicArea *g = new GraphicArea(2,40,VIDEO_AREA_W,VIDEO_AREA_H);
56      g->clear(0);
57
58      //Criando um painel e inserindo todos os componentes;
59      Panel *panel = new Panel(10,10,284,322, "C++ GraphicArea Exemplo");
60      panel->add(prior);
61      panel->add(next);
62      panel->add(fpsLabel);
63      panel->add(g);
64      panel->open();
65      panel->draw();
```

**Figura 9: Exemplo da criação de uma janela em C++.**

Conceitualmente, este gerente está dividido em três outras subcamadas. Na Figura 9 pudemos visualizar dois desses conceitos, os de painéis e componentes. Abaixo, uma descrição superficial dos elementos básicos que compõe esse gerente:

1. *Graphic* é a camada mais básica do gerente de vídeo. É desta subcamada a responsabilidade por criar figuras simples como ponto, reta, quadrado, contorno etc. Tem integração direta com o *driver* de vídeo e é usado por todos os outros componentes gráficos do sistema.
2. *Panel* é uma abstração mais elevada da camada *Graphic*. É responsável por controlar o agrupamento de componentes e determinar quem pode usufruir dos eventos gerados pelo usuário. É constituído por uma lista duplamente encadeada que determina qual componente está em foco durante a execução do sistema.

3. *Component* é um elemento gráfico gerado sobre a camada *Graphic*. É responsável por concentrar funcionalidades, eventos e formas. Possui uma grade variedades de subtipos, tais como: botões, caixas de texto, consoles, áreas gráficas etc.

Em suma, o funcionamento dessa camada consiste em gerenciar a organização da lista de renderização. O sistema possui um ponteiro para uma lista de painéis e gerencia a inserção, remoção e ordenação desta lista. O núcleo ao receber uma chamada de sistema que invoque uma inserção, o painel passado por parâmetro é validado e inserido ao final da lista. Em caso de chamada por remoção é passado por parâmetro o painel que deve ser removido, o sistema efetua as validações necessárias e o remover.

A lista de renderização é similar à usada na gerência de processo, porém não é circular e sua ordenação está diretamente ligada às ações interativas desempenhadas pelo usuário. A lista sofre constantes alterações na ordem dos elementos. Quando um painel é interagido pelo usuário, o mesmo é colocado em foco e é trazido para o início da lista. Os elementos que sofrem maior influência interativa tendem a estarem na frente dessa lista.

Como dito anteriormente, o *kernel* não possui nenhum processo. Todo o trabalho da *renderização e* ordenação são efetuados por programas. Quando um programa requisita a *renderização*, o *kernel* entra em atividade e é feita a validação da chamada. O sistema só permite a *renderização* de telas em foco. Processos podem requisitar o foco, porém o usuário possui o privilégio de determinar qual tela realmente possui o direito de estar na frente da lista.

Em resumo, a *renderização* consiste de uma varredura de todos os elementos da lista, do fim ao início, desenhado suas diretivas gráficas básicas. Desta forma, os painéis são

dispostos em ordem de interação, como ocorre na maioria dos sistemas que possuem interfaces gráficas.

O gerente gráfico é um desafio à parte na elaboração deste sistema. Os benefícios obtidos superaram as dificuldades encontradas. Em virtude da inserção desta camada no *kernel*, o sistema adquiriu um maior controle sobre as ações do usuário, o processamento de eventos se tornou mais simples de ser projetado e surgiu a possibilidade de uma portabilidade das aplicações entre sistemas que implementem o padrão neutrino.

O driver do gerente de vídeo possui três modos de operação, 24bpp, 16bpp e o texto. Todos os painéis e componentes são projetados para trabalhar nos três modos. Infelizmente o modo texto é o mais restritivo, apresentando apenas suporte para um único tipo de componente: o console (os outros componentes são desprezados neste modo). Os programas não necessitam saber a respeito do modo de operação do sistema, com exceção os que fizerem uso exclusivo das diretivas básicas de vídeo, como resolução e total de bits por *pixel*.

### 2.3.4. Gerente de I/O

A gerência de entrada e saída está intimamente relacionada com os aspectos de *hardware*. Na arquitetura neutrino esta camada tem por objetivo controlar o funcionamento de todos os módulos. Sempre que uma ação relacionada ao *hardware* ou *software* ocorre, essa camada é executada automaticamente.

Todos os serviços de entrada e saída são executados pelos gerentes correspondentes. Na prática, esta camada não existe, sendo apenas uma representação teórica de como o sistema funciona. Os *drivers*, *interrupções* e configurações são executados em suas camadas

correspondentes. A adição de novos dispositivos ou funcionalidades são destinadas a camadas posteriores de softwares tais como bibliotecas e programas.

Tudo no neutrino é controlado via interrupções. Existem basicamente dois tipos; interrupções de software e interrupções de hardware.

Existe apenas uma interrupção de software, a 0x30. Todos os gerentes possuem suas funções instaladas nessa interrupção. Sempre que a interrupção 0x30 é chamada, o programa deve informar qual função deve ser executada. Isso é feito através da passagem de um parâmetro via o registrador acumulador. Na Figura 10 é demonstrado um exemplo de uma chamada de interrupção via software, retirada de um dos exemplos da plataforma.

```
34  desktop_Main:
35      MOV EAX,53                  ;
36      INT 30h                     ;Carregando informações do sistema
```

**Figura 10: Executando uma chamada de sistema.**

O sistema possui suporte para até 255 interrupções. Destas, 48 são de uso exclusivo do sistema.

As interrupções de *hardware* são as nativas. Existem basicamente 16 tratadores, todos destinados aos gerentes. Sempre que uma interrupção de *hardware* é gerada, a plataforma ativa áreas de código específicas desses gerentes. Das 48 interrupções usadas, 47 são de *hardware*[2]. Se uma falha de *hardware* ou *software* ocorre, automaticamente os tratadores apropriados são executados e medidas drásticas podem ser tomadas.

Podemos incluir nesse gerente a teoria de independência de dispositivo. Para o usuário não existe diferença entre unidades de armazenamento. Unidades PATA[3], PATAPI[4], SATA[5] e

[2] Incluindo as interrupções destinadas às exceções.
[3] PATA (*Parallel Advanced Technology Attachment*)

STAPI[6] são tratadas igualmente. Os *drivers* são usados nesse sentido, o de incluir codificações especifica de dispositivos escondendo a real interação com o hardware, diga-se de passagem, a parte mais complexa desse sistema.

No estágio atual de desenvolvimento, os *drivers* de disco não possuem suporte a DMA (*Direct Memory Access*). A leitura e escrita são feitas via PIO (*Programmed input/output*). Não existem *drivers* para controladoras PCI (*Peripheral Component Interconnect*) e USB (*Universal Serial Bus*). Alguns dispositivos são emulados via controladoras alternativas reduzindo drasticamente o desempenho do sistema. Em futuras versões, os *drivers* terão maior atenção.

### 2.3.5. Gerente de Eventos

O gerente de eventos é o módulo responsável por controlar as atividades interativas executadas pelo usuário. Na arquitetura neutrino, essa camada recebe uma atenção especial, pois raramente é retratada em bibliografias do assunto. Funciona como complemento ao gerente de vídeo e I/O e tem a responsabilidade de controlar a execução de programas orientados a evento.

Todo programa interativo necessita executar trechos específicos de código em resposta a eventos. É bem verdade que alguns programas são bem simples e apresentam apenas leitura e saída de dados. Para esses programas, este gerente se torna desnecessário. Na prática, existe uma necessidade de fabricar aplicações mais elegantes, com outros tipos de interação e

---

[4] PATAPI (*Parallel Advanced Technology Attachment Packet Interface*)
[5] SATA (*Serial Advanced Technology Attachment*)
[6] SATAPI (*Serial Advanced Technology Attachment Packet Interface*)

fazendo uso de telas gráficas mais amigáveis. Possuir um gerente de eventos no *kernel* torna possível reduzir a complexidade na construção desses aplicativos.

Sempre que um evento assíncrono ocorre, o driver de dispositivo é responsável por tratá-lo. Posteriormente, o gerente de eventos entra em ação. Com o auxílio do gerente de vídeo, o gerente de eventos determina qual aplicativo deverá receber o foco da entrada de dados. Uma vez determinado qual programa tem o direito de executar um evento, o sistema verifica se o aplicativo é capaz de tratá-lo. O gerente de eventos *bufferiza* todos os eventos gerados e sempre que o processo entra em execução um evento do *buffer* é removido e o sistema passa a executar o tratador correspondente.

Existem duas camadas de gerência de eventos: uma na aplicação, contida no CRT0 e outra contida no gerente de eventos. A ideia básica é o do funcionamento cliente e servidor. O gerente de eventos envia mensagens; e o host as recebe e as trata.

O programa deve inicialmente registrar um evento para que o *kernel* possa executá-lo quando o sistema entrar em ação. O registro é bastante simples, graças à biblioteca de integração. Na Figura 11 podemos visualizar um tratador de eventos sendo registrado. Entre as linhas 31 e 33 foi definido um tratador de eventos; seu registro ocorre na linha 44.

```
30    //Procedimento para trocar de animação;
31    void priorAnim() {
32        anim <= 0 ? anim = TOTAL_ANIME - 1 : anim--; //Escolhendo a animação anterior;
33    }

43        Button *prior = new Button(2,18,38,20,"prior\0");
44        prior->addEventHandler( priorAnim, Component::ON_MOUSE_DOWN_L );
```

**Figura 11: Registrando um tratador de eventos.**

A estrutura de dados usada consiste de uma fila simples, porém, para uma maior segurança do sistema, os eventos são cadastrados em uma área de espaço fixo de 10 posições,

na pilha dos processos. Assim, o *kernel* só permite um número limitado de eventos concorrentes por processo. Sempre que o evento é executado o sistema libera o espaço usado e aguarda a entrada de novos eventos.

### 2.3.6. Gerente de Arquivo

A gerência de arquivos, sem dúvida alguma, é a parte mais complexa desse sistema operacional. Poderia ser escrito um livro inteiro sobre como construir um sistema de arquivos. São centenas de problemas e detalhes relacionados com concorrência, segurança, integridade, usabilidade, velocidade e espaço que podem ser encontrados. Um bom gerente de arquivos deve ser capaz de atender, pelo menos em parte, alguns dos problemas citados. Esta é uma opinião pessoal.

O gerenciamento de arquivos escolhido para o neutrino foi baseado em tabelas, pois se trata de um dos mecanismos mais simples de gerência. Entretanto, o sistema é completamente diferente em relação aos principais sistemas de arquivos que adotam essa filosofia. Esse sistema foi intitulado NFS. (*Neutrino File System*).

A construção do NFS foi dividida em duas etapas. A primeira ficou responsável por alocar dados em disco. A segunda possui a tarefa mais complexa, oferecendo uma perspectiva de pastas e diretórios que podem ser usados por programas, servindo como abstração da primeira camada.

Os dados em disco são alocados conforme o gerente de memória, anteriormente apresentado. Todavia, o sistema é especializado em armazenamento em massa. A grande diferença na codificação é o fato de os dados estarem sempre salvos em uma mídia, exigindo

um controle adicional para que os dados trafeguem da memória principal para os dispositivos de armazenamento.

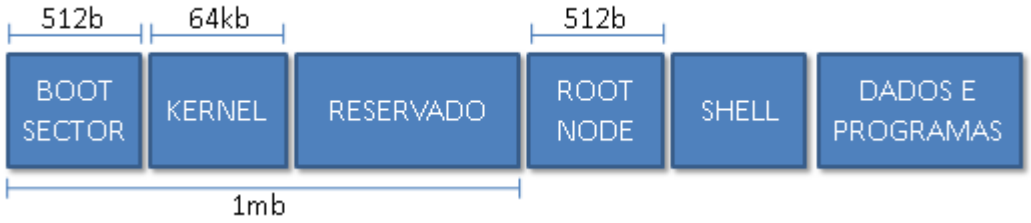


**Figura 12: Organização dos dados em disco.**

Na Figura 12 é apresentada a organização dos dados em disco. Podemos classifica-la em cinco categorias distintas, descritas logo abaixo:

1. O *boot sector* está localizado nos primeiros 512b da unidade de armazenamento, contém o código de *boot,* responsável por carregar o sistema operacional para memória principal e possui o registro responsável por gerenciar o sistema de arquivos.
2. O *kernel* é instalado logo após o *boot sector*, possui o tamanho de aproximadamente 64kb. Em sequência, existe uma área reservado para a expansão.
3. O *root node* é a raiz do diretório principal, possui um total de sete pastas e não pode ser modificado. É criado logo após o primeiro 1mb da unidade de armazenamento. Na Figura 13 podemos visualizar a listagem de diretório. Das pastas apresentadas, a boot e prog são as mais importantes. A pasta boot é responsável por conter os programas que são inicializados junto ao sistema, foi projetada para conter o *shell*. A pasta "prog" contém todos os programas e serviços não inicializáveis.

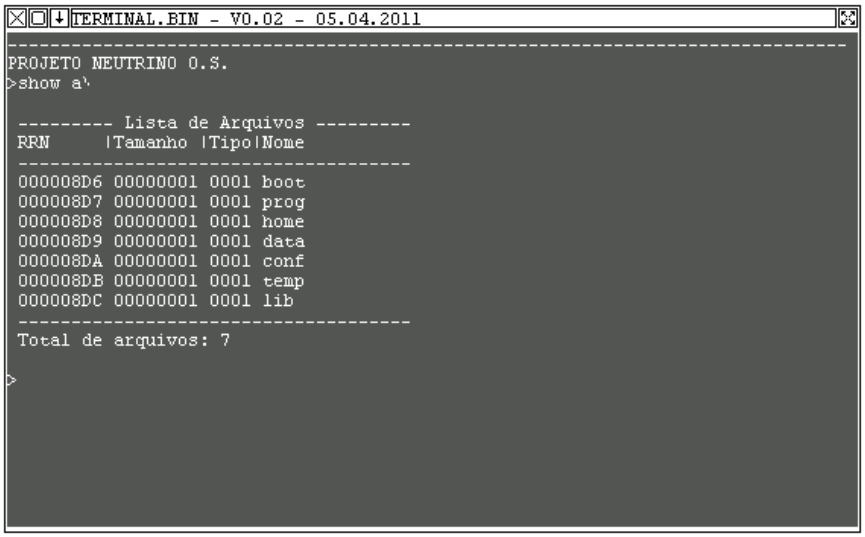


**Figura 13: Estrutura do diretório raiz.**

4. O s*hell* inicia logo em seguida, posteriormente ao diretório principal, apresentando todos os programas responsáveis por oferecer uma interface homem máquina. Geralmente, programas carregados posteriormente a inicialização do núcleo. Todos os dados são agrupados de forma contínua para aumentar a velocidade de inicialização[7].
5. Os dados, programas e tabelas de arquivos são armazenadas logo após o *shell*. Podem ocupar todo o disco. O usuário deve decidir como os programas e seus dados serão armazenados.

O sistema de arquivos é alocado conforme uma árvore. Cada nó pode apresentar infinitos subnós ou folhas. As folhas representam os dados do usuário e o subnós às pastas. O único nó imutável é o nó principal. Este apresenta apenas sete nós, demonstrados na Figura 13.

O sistema de arquivos é gerenciado pelo núcleo através de interrupções. Existem sete funções básicas oferecidas pelo núcleo no estágio atual de desenvolvimento: criar, remover,

[7] Atualmente o tempo de inicialização é da ordem de micro segundos.

procurar, abrir, ler, escrever, fechar. Os algoritmos são demorados e complexos de serem explicados. Porem, na prática consiste em detectar a unidade de armazenamento, procurar um caminho até um nó específico e efetuar uma das sete operações básicas. O programador deve informar sempre uma *string* contendo o caminho completo até um nó.

O NFS possui algumas características, que diferem da maioria dos sistemas de arquivos. Os dados são armazenados de forma sequencial. Como benefício, os dados não sofrem com fragmentação. Ideal para armazenamento de programas, imagens e vídeos, pois são tipos de dados que são beneficiados por estarem de forma contínua. Como prejuízo, dados que necessitem alterações em seus tamanhos tendem a perder desempenho em virtude de constantes reposicionamentos dos dados em disco.

O projeto do sistema de arquivos, assim como todo o *kernel*, tem como objetivo aplicações de uso acadêmico. O NFS cumpre o seu papel quanto a isso.

## 2.4. Camada Hardware

O hardware é a camada mais baixa da arquitetura neutrino. O sistema foi projetado para trabalhar sobre a arquitetura x86, utilizando processadores posteriores à geração Intel 486.

Pelo fato de todo o código ter sido escrito completamente em *assembly* (Intel 32bits), o sistema perdeu a portabilidade entre computadores. Em virtude disso, o núcleo foi exclusivamente projetado para a plataforma Intel, porém é possível executá-lo em processadores que possuem *sets* de instruções compatíveis. Como exemplo, o sistema possui suporte para quase todos os processadores AMD.

O objetivo central da plataforma está no uso de computadores mais modestos e antigos. Sets de instruções mais modernos não são suportados pela versão atual do sistema. Futuras atualizações poderão conter comandos mais novos.

A plataforma consome uma baixa quantidade de memória e ao mesmo tempo oferece recursos básicos para execução de programas. Ao todo, o sistema utiliza 4mb de memória RAM (*Random Access Memory*), sendo necessário apenas 1mb para que o mesmo possa rodar em modo texto, isto é, com o modo de gerência de vídeo desabilitado. Levando em conta que maior parte da memória gasta pela plataforma está relacionada a *buffers* do sistema e dados gráficos, em soluções mais críticas o sistema pode ser recompilado para ocupar 64kb de RAM, perdendo em parte o desempenho obtido com o uso de 1mb.

O uso de espaço em disco é equivalente ao de memória. O sistema necessita de apenas 1mb de disco para carregar todas as suas funcionalidades. Uma vez carregado, o sistema não precisa mais de uma unidade de disco. As operações básicas podem ser executadas diretamente em memória, com exceção das funções executadas por programas que referenciem o disco, como leitura e escrita.

O vídeo funciona sobre o padrão VESA[8] (*Video Electronics Standards Association*) 1.2 ou superior. Sempre que o sistema inicia é feita uma verificação de *hardware* que determina qual a modalidade de vídeo será usada. Existem ao todo nove modos distintos suportados pelo *kernel*: quatro para 16bits de cor, outros quatro para 24bits e uma modalidade especial compatível com todos os computadores x86, o modo texto.

O sistema não necessita de drivers para componentes básicos como mouse, teclado, vídeo e dispositivos de armazenamento. Todos os gerentes são construídos sobre os padrões

[8] Fornece padrões universais para construção de placas de vídeo.

gerais impostos pelos fabricantes de *hardware*. Muitos dispositivos funcionam abaixo da velocidade ideal em virtude da generalização dos algoritmos de acesso aos dispositivos. Para contornar o problema o sistema foi construído com alguns conceitos do *exokernel*, possibilitando que processos acessem diretamente o *hardware*. Programas que necessitam de maior desempenho podem ser compilados com *drivers* específicos para dispositivos.

O sistema é capaz de rodar nas principais máquinas virtuais do mercado. Dentre elas o sistema apresenta compatibilidade com o *Virtual PC*, *VMware*, *Virtual Box*, *Bochs*, *Parallels* e *Qemu*. Quanto ao *hardware* real, o sistema apresenta compatibilidade crescente, porém no estágio atual, alguns equipamentos mais modernos podem não funcionar de forma adequada, forçando o sistema a executar em modo texto.

# GLOSSÁRIO

Bootável- um neologismo. Consistem na capacidade de um software ser inicializado por um computador durante a etapa de boot.

Complexidade- consiste na quantidade de trabalho necessária para a execução de determinado algoritmo.

CPU-bound- termo utilizado para processos que executam grande carga de processamento.

Driver- tipo de software que permite ao sistema operacional se comunicar com algum dispositivo de hardware.

Exo- é uma arquitetura para sistemas operacionais, criada pelo MIT. Possui um conceito de emulação de computadores reais para provimento de funcionalidades. Seu trabalho é atribuir recursos a máquinas virtuais e, então, verificar tentativas de utilizá-los assegurando que nenhuma máquina está tentando usar recursos de outra.

Fragmentação Interna- é o não aproveitamento ou perda de espaço da memória dentro de uma área de tamanho fixo.

Free- função para liberar memória em programas C.

Flat binaries- formato de arquivo binário que não possui header com informações pertinentes ao sistema operacional. Apresenta na sua estrutura apenas código máquina e seus respectivos dados.

Hardware- parte física de um computador. É um conjunto de componentes eletrônicos.

Interrupção- é um evento externo ou interno que causa o desvio da execução para um bloco de código denominado ISR (*Interrupt Service Routine*).

I/O-bound- termo utilizado para designar os processos que fazem uso intensivo de entrada e saída.

Kernel- camada de software que contém os componentes centrais de um sistema operacional.

Malloc- função usada para alocar memória em programas C.

Modular- é uma arquitetura em que as camadas do núcleo são divididas em módulos. O kernel é minimalista e se encarrega basicamente de coordenar a troca de mensagens e dados entre os diferentes componentes do sistema.

Monolithic- é uma arquitetura em que todas as funções do sistema operacional trabalham dentro do espaço do kernel. Não possui um modo supervisor.

Monotarefa- sistema que permite a realização de apenas uma tarefa de cada vez.

Multitarefa- sistemas que permitem repartir a utilização do processador entre várias tarefas simultaneamente.

New- função em alto nível, usada para alocar memória em programas. Geralmente uma sobrecarga da função malloc.

Página- unidade de alocação de memória usada na arquitetura neutrino. Consiste em um tamanho de 512byte.

Preempção- capacidade de alterar a ordem da execução de um processo em detrimento de outro.

Processo- consiste em um programa em execução. Contém dados, pilha, memória dinâmica e recursos especiais controlados pelo sistema operacional.

Quantum- unidade de tempo usada para geração de interrupção de troca de contexto. No neutrino este tempo é de 18.2ms.

Reentrância- é a qualidade de uma subrotina ser executada concorrentemente de forma segura, isto é, a subrotina pode ser invocada enquanto está em execução.

Renderização- ação que consiste em efetuar atualizações gráficas de elementos em vídeo.

Ring- camada de acesso usada em sistemas operacionais. Quanto menor o Ring, maior controle sobre o equipamento e menor tolerância a falhas.

Starvation- sinônimo de inanição. Em programação concorrente, ocorre quando um processo nunca é executado.