

Arquitetura do Kernel Linux

Alan Godoy S. Mello

Instituto de Computação

7 de maio de 2009

1 Introdução

- O que é um kernel?
- Funções do kernel
- Arquiteturas de kernel

2 Divisão do kernel Linux

- Sistema de arquivos
- Gerenciamento de processos
- Gerenciamento de memória
- Rede
- Drivers de dispositivos

3 QEMU

- Virtualização
- QEMU
- Usando o QEMU

O que é um kernel?

- Em uma divisão do sistema operacional em camadas, kernel é o nome dado à porção de software mais próxima da máquina.
- Centro da maioria dos sistemas operacionais.
- Responsável por prover as funcionalidades básicas às aplicações executando no sistema.

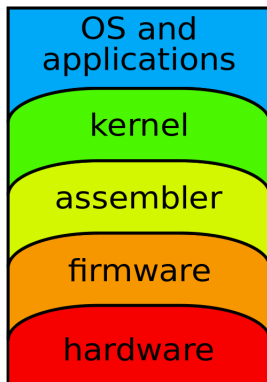


Figura: Visão em camadas de uma arquitetura de computador.

Funções do kernel

- Abstrair o funcionamento do *hardware*, fornecendo um meio de desenvolver aplicações que funcionem em diversas configurações de máquinas e melhorando a eficiência de algumas atividades junto ao hardware.
- Garantir a segurança da máquina e dos processos em execução, ao controlar quem e como os recursos do sistema podem ser acessados.
- Realizar uma divisão adequada dos recursos entre os processos, garantindo o isolamento de cada processo.
- Prover ao usuário o acesso aos serviços disponibilizados, através de chamadas de sistema.

1 Introdução

- O que é um kernel?
- Funções do kernel
- Arquiteturas de kernel

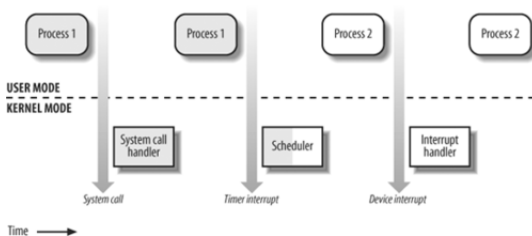
2 Divisão do kernel Linux

- Sistema de arquivos
- Gerenciamento de processos
- Gerenciamento de memória
- Rede
- Drivers de dispositivos

3 QEMU

- Virtualização
- QEMU
- Usando o QEMU

- A glibc permite a transição entre o modo de usuário e o modo kernel; ao realizar uma chamada de sistema, um processo de usuário torna-se um processo do kernel.
- O kernel Linux, em si, não é um processo, mas um gerenciador de processos, executado nas seguintes situações: em resposta a uma chamada de sistema, exceção ou interrupção ou, mais raramente, em alguma *thread* do kernel, que seja responsável por executar algumas funções periodicamente.



- Em uma visão mais rápida, o Linux pode ser dividido em alguns subsistemas principais:

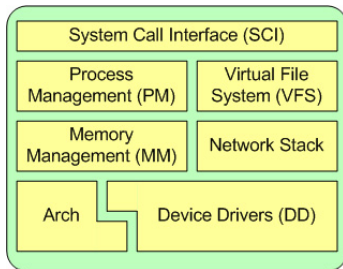


Figura: Principais subsistemas do kernel Linux.

- Apesar de essa ser uma divisão comum, ela não é única, assim como diversas partes do kernel relacionam-se entre si, independentemente dos subsistemas exibidos acima.

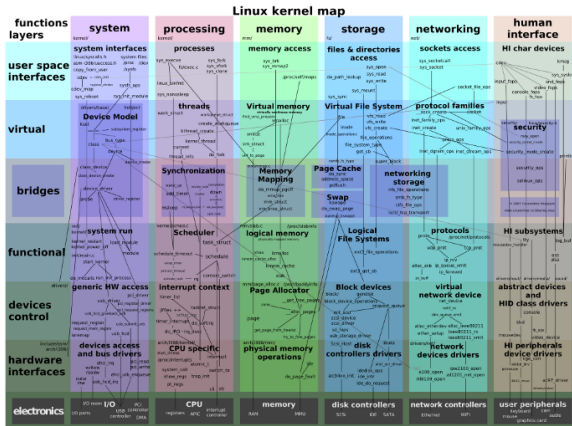


Figura: Mapa detalhado do kernel Linux.

- Sistemas de arquivo são modelos acerca da organização lógica dos arquivos, podendo não lidar com dispositivos de armazenamento (como o PROCFS, gerado durante o carregamento do sistema e que permite obter informações do kernel, o DEVFS, que interagir com dispositivos através de chamadas de sistema de E/S).
- Como exemplos de sistemas de arquivos implementados para o Linux, temos EXT3, JFS, NFS, GMailFS e WikipediaFS, sendo os dois últimos implementados em espaço de usuário, através do FUSE.

- Cada sistema de arquivos deve lidar com algumas estruturas básicas: o `superblock`, o `inode` e o `dentry`.
 - `superblock`: responsável por armazenar informações como o tipo do sistema de arquivos, tamanho, dispositivo, lista de `inodes` e suas operações básicas, como criação e remoção de `inodes`.
 - `inode`: representa um objeto do sistema de arquivos (um arquivo ou um diretório), contendo informações como dono, grupo, permissões, blocos e operações (como leitura e escrita no arquivo).
 - `dentry`: responsável por estabelecer a relação entre nomes de arquivos e `inodes`.
- Ao separar os arquivos de seus nomes, é possível atribuir vários nomes a um mesmo arquivo (*hard link*).


```
struct inode {
    unsigned long      i_ino;
    umode_t            i_mode;
    uid_t              i_uid;
    struct timespec     i_atime;
    struct timespec     i_mtime;
    struct timespec     i_ctime;
    unsigned short      i_bytes;
    struct inode_operations *i_op;
    struct file_operations *i_fop;
    struct super_block  *i_sb;
    ...
}

struct inode_operations {
    int (*create)(struct inode *, struct dentry *,
                  struct nameidata *);
    struct dentry *(*lookup)(struct inode *,
                             struct dentry *,
                             struct nameidata *);
    int (*mkdir)(struct inode *, struct dentry *, int);
    int (*rename)(struct inode *, struct dentry *,
                  struct inode *, struct dentry *);
    ...
}

struct file_operations {
    struct module *owner;
    ssize_t (*read)(struct file *, char __user *,
                    size_t, loff_t *);
    ssize_t (*write)(struct file *, const char __user *,
                     size_t, loff_t *);
    int (*open)(struct inode *, struct file *);
    ...
}
```

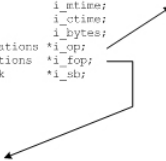


Figura: A estrutura do inode e suas operações.

Gerenciamento de processos

- Um processo pode ser usualmente definido como uma instância de um programa em execução, podendo haver vários processos de um mesmo programa rodando ao mesmo tempo.
- O Linux diferencia claramente um processo de um programa: um processo é criado através da função `do_fork()` do kernel – a qual pode ser acionada pelo usuário através da chamada de sistema `fork()` –, enquanto que um programa é carregado através da família de chamadas de sistema `exec()`.

- Inicialmente o kernel Linux não oferecia suporte a aplicações com múltiplas *threads*, sendo que elas deveriam ser implementadas em modo usuário; isso causava problemas pois quaisquer chamadas que levassem ao bloqueio de uma das *threads* do processo em nível de kernel fazia com que todas as outras *threads* daquele processo acabassem bloqueadas.
- Atualmente o Linux oferece esse suporte no kernel, sendo que tanto processos como *threads* são implementados utilizando a idéia de processos leves (*lightweight process*).
- Dois processos leves podem compartilhar alguns recursos, como espaço de endereçamento e arquivos abertos, de tal forma que sempre que um dos processos modifica um dos recursos, o outro pode imediatamente perceber a alteração.

```
struct task_struct {  
  
    volatile long state;  
    void *stack;  
    unsigned int flags;  
  
    int prio, static_prio;  
  
    struct list_head tasks;  
  
    struct mm_struct *mm, *active_mm;  
  
    pid_t pid;  
    pid_t tgid;  
  
    struct task_struct *real_parent;  
  
    char comm[TASK_COMM_LEN];  
  
    struct thread_struct thread;  
  
    struct files_struct *files;  
  
    ...  
  
};
```

Figura: A representação de um processo no kernel.

- No kernel cada processo leve é representado por uma estrutura chamada `task_struct`.
- Esta estrutura armazena informações completas sobre a execução do processo, como identificadores, prioridades de execução, estado da execução da tarefa, processos relacionados (pais e filhos), apontador para a pilha de execução e o estado da tarefa (dependente de arquitetura), utilizado para permitir a troca de tarefas em execução.
- Cada processo leve possui um identificador único, chamado `pid`, sendo que *threads* de um mesmo processo formam um grupo, identificado pelo valor de `tgid`, igual ao `pid` da primeira tarefa no grupo; o valor de `tgid` é o valor retornado pela chamada de sistema `get_pid()`.

Criação de processos

- Exceto pelo processo `init`, criado na inicialização e que existe durante toda a execução do sistema, todos os novos processos e *threads*, seja em espaço de usuário ou de kernel, são resultado de alguma chamada a `do_fork()` a partir de um outro processo (processo pai).
- Ao ser invocada, a chamada `do_fork()` cria um novo `pid` para a tarefa, verifica as `flags` do processo pai e copia a `task_struct` do pai; caso esteja sendo criado um novo processo são duplicadas as estruturas de memória, de gerência de arquivos e de sinais do pai, caso seja uma nova *thread* essas estruturas apenas apontarão para as mesmas estruturas que o pai; finalizado esse processo, o novo processo é, então, acordado.

Finalização de processos

- Quando um processo é finalizado é realizada a função `do_exit()`, responsável por remover do sistema todas as referências ao processo em questão, exceto para o caso de recursos compartilhados, e por emitir notificações sobre a finalização do processo.
- Se é requerida uma sinalização ao pai, por exemplo, a tarefa entra em estado zumbi, mantendo seu estado na memória até que uma chamada do tipo `wait()` seja executada; caso contrário, a memória ocupada pelo processo é liberada.

Escalonamento

- Criados os processos, o escalonador (função `schedule()`) deve garantir que cada processo seja executado por um determinado tempo, criando a ilusão de execução simultânea dos processos; para isso ele se valhe de critérios que buscam o máximo aproveitamento do tempo de processador, ao mesmo tempo que atribui prioridades diferentes a cada processo, segundo as necessidades do usuário.
- Contabilizando as necessidades da aplicação como prioridade estática, atribuída pelo usuário, nível de interatividade e demanda por execução em tempo real, os processos do Linux podem receber diferentes valores de prioridade dinâmica.

- O controle de tempo é feito através de *ticks* do *clock* do computador, que executa chamadas periódicas ao escalonador; dependendo da forma como a tarefa é escalonada ela pode ter um tempo limite para se manter no processador, o qual depende de sua prioridade estática.
- Do kernel 2.4 para o 2.6 foram feitas grandes mudanças no escalonador, de forma que atualmente ele executa em tempo $O(1)$, organizando as tarefas segundo suas prioridades; a partir do kernel 2.6 também é possível um melhor uso de múltiplos processadores, de forma que cada processador mantém uma fila de execução própria, sendo feito um balanceamento periódico (200ms) para evitar que alguns processadores fiquem ociosos enquanto outros encontrem-se sobrecarregados.

Gerenciamento de memória

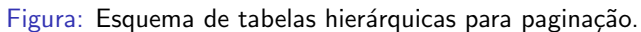
- O gerenciamento de memória no Linux é feito através de um esquema de memória virtual, o qual implementa uma camada de abstração que faz com que os endereços vistos pelos programas de usuários não correspondam diretamente aos endereços em *hardware*.
- O uso de memória virtual busca atingir alguns objetivos, como:
 - Relocação: um programa deve ser capaz de executar em diferentes pontos da memória.
 - Proteção e compartilhamento: processos co-existent não devem interferir uns nos outros (exceto quanto explicitado).
 - Transparência: uma aplicação não precisa de informação sobre o tamanho da memória, uma vez que não precisa restringir-se à memória fisicamente disponível.

Segmentação

- Segmentação divide a memória em pedaços de tamanhos variáveis e com diferentes tipos de permissões, sendo os endereços calculados com base no início do segmento atual.
- O Linux utiliza a idéia de segmentação de forma limitada, preferindo esquemas de paginação, o que permite maior simplificação no gerenciamento de memória e maior portabilidade.
- São utilizados apenas seis segmentos: segmento de código do kernel, de dados do kernel, de código de usuário, de dados de usuário, de TSS e de LDT; esses segmentos são usados para definir proteção em nível de *hardware* para essas áreas.

Paginação

- Paginação divide a memória em blocos contínuos e de tamanho fixo, também tendo níveis de permissões; o espaço de uma página na RAM é chamado de *page frame*, sendo que páginas são mapeadas em *page frames* através de *page tables*.
- Caso seja necessário mais memória do que a disponível fisicamente, o kernel pode mover páginas pouco usadas da memória para o disco; dessa forma, as páginas devem ser marcadas como presentes ou ausentes na RAM.
- As páginas são organizadas em tabelas hierárquicas que podem ser acessadas dividindo o endereço em partes, sendo a porção mais significativa responsável por indicar a posição da página nessas tabelas, enquanto que a porção menos significativa indica o *offset* dentro da página.



- O Linux utiliza 4 níveis de tabelas hierárquicas: *Page Global Directory*, *Page Upper Directory*, *Page Middle Directory* e *Page Table Entry*.
- Essa implementação busca compatibilidade com diferentes arquiteturas, sendo esses 4 níveis utilizados apenas em arquiteturas 64 bits; em outras arquiteturas os níveis intermediários são eliminados para maior eficiência.
- Cada processo possui suas próprias tabelas de páginas, as quais contém apenas os endereços que o processo pode utilizar.

- O uso de 32 bits possibilita o endereçamento de até 4GB, sendo que o kernel divide esse espaço virtual em espaço de usuário e espaço de kernel (usualmente 3GB e 1GB, respectivamente).
- Como o kernel pode manipular apenas memória que esteja mapeada em seu espaço de endereçamento, há alguns anos o Linux em um x86 poderia manipular pouco menos de 1GB de memória física.
- Atualmente quando o kernel precisa trabalhar com alguma porção da *high memory* ele cria um mapeamento durante a execução, o que causa deterioração de desempenho.

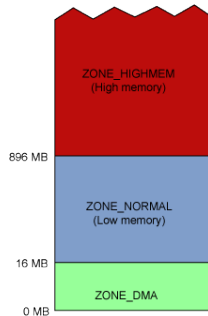


Figura: Regiões da memória.

- Camada de enlace: relação direta com o *hardware*, responsável pelo roteamento de pacotes; exemplo de protocolo: Ethernet.
- Camada de rede: responsável pelo endereçamento dos pacotes ao seu destino final; exemplo de protocolo: IP.
- Camada de transporte: responsável pelo redirecionamento interno aos servidores (portas, por exemplo) e implementação de mecanismos de confiabilidade, quando necessário; exemplos de protocolo: TCP e UDP.
- Camada de aplicação: camada na qual os dados transportados apresentam valor semântico; exemplos de protocolo: HTTP, FTP e SSH.

- Interface agnóstica de protocolo: camada de abstração e protocolos; os protocolos devem se registrar junto a essa camada, definindo suas operações básicas.

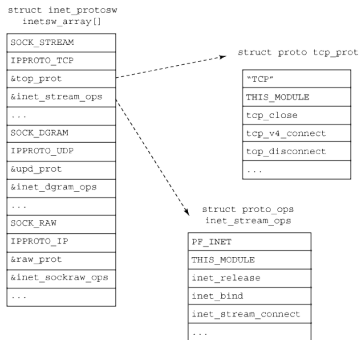


Figura: Estrutura do vetor de protocolos.

- Protocolos de rede: implementação específica de protocolos; alguns protocolos são incluídos por padrão, como IP e TCP, mas novos protocolos podem ser registrados.
 - As movimentações de dados através da rede são feitas através de *socket buffers*, os quais contém o pacote em questão e informações incluídas pelas várias camadas, como dispositivo utilizado, *socket*, *timestamp* e cabeçalhos de protocolos, podendo ser organizados em filas para cada conexão específica.

- Interface agnóstica de dispositivo: outra camada de abstração, porém relacionando protocolos a dispositivos, fornecendo, por exemplo, métodos genéricos para enfileirar *buffers* de entrada e de saída; os dispositivos disponíveis devem registrar-se nessa camada.
- Drivers de dispositivos: implementações específicas de drivers de dispositivos; nessa camada são definidas as funções responsáveis por, de fato, enfileirar *buffers* para transmissão (dependente de *hardware*).

Drivers de dispositivos

- A implementação separada de drivers de dispositivos permite maior transparência na relação entre kernel e *hardware*.
- Drivers podem ser visto como “caixas pretas”, que fazem com que um determinado *hardware* interaja com o sistema operacional através de uma interface bem definida.
- O Linux implementa drivers através de módulos, os quais podem ser dinamicamente carregados e descarregados quando necessário, permitindo que uma imagem muito menor do kernel.

Módulos

- Módulos não relacionam-se apenas a drivers, podendo implementar funções internas ao kernel ou novas camadas de abstração, por exemplo.
- Ao contrário de aplicações de usuário, módulos do kernel são orientados a eventos, como carregamento e descarregamento.
- Módulos registram em seu código informações sobre autor, descrição e licença, a qual pode acarretar no bloqueio do módulo.
- Algumas ferramentas para trabalhar com módulos: `modprobe`, `depmod` e `modinfo`.

Tipos de dispositivos

- Vários recursos no Linux são mapeados no sistema de arquivos, o que facilita sua manipulação, sendo que a maior parte dos dispositivos são encontrados no diretório `/dev`.
- São distinguidos três tipos básicos de dispositivos:
 - Dispositivos de caractere: orientado a fluxos de dados permitindo, normalmente, apenas acesso sequencial; exemplos de dispositivos: `/dev/console`, `/dev/random`, `/dev/lp0`.
 - Dispositivos de bloco: dispositivo que pode alojar um sistema de arquivos; exemplos de dispositivos: `/dev/fd0` e `/dev/sda`.
 - Interfaces de rede: capaz de trocar dados com outros servidores, não sendo necessariamente um dispositivo; ao contrário dos outros tipos não é mapeado no sistema de arquivos e é tratado pelo kernel através de pacotes; exemplo de dispositivo: `eth0`.

Números de dispositivos

- Cada dispositivo possui um número único, sendo formado por duas partes:
 - *Major number*: identifica ao kernel o driver respectivo ao dispositivo.
 - *Minor number*: identifica ao driver o dispositivo em questão (um mesmo driver pode gerenciar múltiplos dispositivos).

- Ao listar arquivos (comando `ls -l`) de dispositivo são listados algumas dessas informações.
 - Na região destacada à esquerda é indicado o tipo de dispositivo (b para dispositivos de bloco e c para dispositivos de caractere).
 - Na outra região destacada são indicados os números do dispositivo no formato “*major, minor*”.

b	r	w	-	r	w	-	-	-	+	1	root	floppy	2, 0	2009-05-06	11:28	/dev/fd0
c	r	w	-	-	-	-	-	-	-	1	root	root	108, 0	2009-05-05	09:53	/dev/ppp
c	r	w	-	r	w	-	r	w	-	1	root	root	1, 8	2009-05-06	11:28	/dev/random
b	r	w	-	r	w	-	-	-	-	1	root	disk	8, 0	2009-05-06	11:28	/dev/sda
c	r	w	-	-	-	-	-	-	-	1	root	root	4, 1	2009-05-06	14:28	/dev/tty1
c	r	w	-	r	w	-	r	w	-	1	root	root	1, 5	2009-05-06	11:28	/dev/zero

Figura: Listagem da representação de alguns arquivos no sistema de arquivos.

1 Introdução

- O que é um kernel?
- Funções do kernel
- Arquiteturas de kernel

2 Divisão do kernel Linux

- Sistema de arquivos
- Gerenciamento de processos
- Gerenciamento de memória
- Rede
- Drivers de dispositivos

3 QEMU

- Virtualização
- QEMU
- Usando o QEMU

Virtualização

- Virtualização de plataforma (ou apenas virtualização) consiste na implementação de um programa de controle, criando uma máquina virtual para o sistema convidado (que pode ser um sistema operacional).
- Ao criar intermediar a relação entre o sistema convidado e a máquina real, o programa de controle pode gerenciar recursos de maneira diferenciada, dividindo a máquina e múltiplas máquinas virtuais, criando processadores ou emulando dispositivos que não se encontram disponíveis, por exemplo.
- Alguns mecanismos de virtualização permitem que uma máquina execute *softwares* com conjuntos de instruções de outras máquinas.

- Exemplos de possíveis benefícios do uso de virtualização:
 - Isolamento do sistema convidado com relação ao hospedeiro, reduzindo possíveis danos.
 - Uma máquina virtual pode ser facilmente copiada para outro computador, mesmo que as máquinas tenham configurações diferentes.
 - Múltiplos sistemas operacionais podem ser executados em uma máquina concorrentemente.
 - Controle externo de acesso a recursos e facilidade na inspeção do estado da máquina, o que facilita o desenvolvimento de sistemas operacionais.

QEMU

- O QEMU é um virtualizador e emulador completo de máquinas.
- Ao ser executado como emulador, ele pode executar programas feitos para uma arquitetura em diferentes arquiteturas, através da conversão dinâmica do código de máquina.
- Ao ser usado como virtualizador, o QEMU é capaz de executar o código do sistema emulado diretamente na máquina hospedeira (através de um módulo, chamado KQEMU).

Usando o QEMU

- O uso do QEMU é muito similar ao uso de uma máquina real de forma que, caso não haja um “disco” com um SO instalado, é necessário instalá-lo:

- Como disco pode ser utilizado um dispositivo real ou uma imagem do disco, que pode ser criada com o comando `qemu-img`:

```
$ qemu-img create -f qcow disco.img 512M
```

- Deve ser, então “inserido” o CD-ROM de instalação na sua nova máquina, indicando na “BIOS” que o *boot* deve ser feito a partir do CD-ROM:

```
$ qemu -hda disco.img -cdrom /dev/cdrom -boot d
```

- Agora a instalação segue o formato usual de uma instalação de um sistema operacional.

- Terminada a instalação, você agora pode rodar sua nova máquina.

```
$ qemu -hda disco.img
```

- Caso você queira rodar uma imagem específica de kernel que não esteja no dispositivo especificado, pode ser usado o parâmetro `-kernel`:

```
$ qemu -hda disco.img -kernel bzImage
```

- Por padrão, o QEMU abre uma janela SDL para saída, caso você queira inibir essa janela (usando SSH, por exemplo), use o parâmetro `-nographic`.

Referências I



J. Corbet, A. Rubini, and G. Kroah-Hartman.
Linux Device Drivers, 3rd Edition.
O'Reilly Media, Inc., 2005.



A. D'Assumpção.
Introdução aos linux device drivers.
[http://www.adassumpcao.net/
introducao-aos-linux-device-drivers](http://www.adassumpcao.net/introducao-aos-linux-device-drivers), 2007.
[Online; acessado em 6-maio-2009].

Referências II



M. T. Jones.

Inside the linux scheduler.

<http://www.ibm.com/developerworks/linux/library/l-scheduler/>, 2006.

[Online; acessado em 5-maio-2009].



M. T. Jones.

Anatomy of the linux file system.

<http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/>, 2007.

[Online; acessado em 4-maio-2009].

Referências III



M. T. Jones.

Anatomy of the linux kernel.

<http://www.ibm.com/developerworks/linux/library/l-linux-kernel/>, 2007.

[Online; acessado em 4-maio-2009].



M. T. Jones.

Anatomy of the linux networking stack.

<http://www.ibm.com/developerworks/linux/library/l-linux-networking-stack/>, 2007.

[Online; acessado em 6-maio-2009].

Referências IV



M. T. Jones.

System emulation with qemu.

<http://www.ibm.com/developerworks/linux/library/l-qemu/>, 2007.

[Online; acessado em 6-maio-2009].



M. T. Jones.

Anatomy of linux process management.

<http://www.ibm.com/developerworks/linux/library/l-linux-process-management/>, 2008.

[Online; acessado em 4-maio-2009].

Referências V



P. Larson.

Kernel comparison: Improved memory management in the 2.6 kernel.

[http:](http://www.ibm.com/developerworks/library/l-mem26/)

[//www.ibm.com/developerworks/library/l-mem26/](http://www.ibm.com/developerworks/library/l-mem26/),
2004.

[Online; acessado em 5-maio-2009].



V. Shukla.

Explore the linux memory model.

[http://www.google.com/search?q=cache:dDH_PeA02TEJ:](http://www.google.com/search?q=cache:dDH_PeA02TEJ:www.ibm.com/developerworks/library/l-memmod/)
www.ibm.com/developerworks/library/l-memmod/, 2006.

[Online; acessado em 5-maio-2009].

Referências VI



Wikipedia.

Microkernel — wikipedia, the free encyclopedia.

<http://en.wikipedia.org/w/index.php?title=Microkernel&oldid=287802400>, 2009.

[Online; acessado em 4-maio-2009].



Wikipédia.

Kernel — wikipédia, a enciclopédia livre.

<http://pt.wikipedia.org/w/index.php?title=Kernel&oldid=14952227>, 2009.

[Online; acessado em 4-maio-2009].