Referências Bibliográficas

ALTSCHUL, S.; GISH, W.; MILLER, W.; MYERS, E.; LIPMAN, D. J. **Basic Local Alignment Search Tool**. Journal of Molecular Biology 215, 1990, pp. 403-410.

ALTSCHUL, S.; MADDEN, T.; SCHÄFFER, A.; ZHANG, J.; ZHANG, Z.; MILLER, W.; LIPMAN, D. **Gapped blast and psi-blast: a new generation of protein database search programs**. Nucleic Acids Research 25(17), 1997, pp. 3389-3402.

ASN. **ASN.1 Information Site**. Disponível em: <http://asn1.elibel.tm.fr/xml>. Acesso em: ago. 2006.

BEDELL, J.; KORF, I.; YANDELL, M. **BLAST**. O'Reilly, 2003.

BENSON, D. A.; KARSCH-MIZRACHI, I.; LIPMAN, D. J.; OSTELL, J.; RAPP, B. A.; WHEELER, D. L. **GenBank**. In: Nucleic Acids Research 28(1), pp. 15-18, 2000.

CAMERON, M.; WILLIAMS, H. E.; CANNANE, A. **Improved Gapped Alignment in BLAST.** IEEE/ACM Transactions on Computational Biology and Bioinformatics, 1(3), 2004, pp. 116-129.

CCB. **Center for Computational Biology: BioPostgres**. Disponível em: <http://www.biopostgres.org, 2006>. Acesso em: ago. 2006.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. **Linux Device Drivers**. 3 ed. O'Reilly, 2005.

COSTA, R. L. C.; LIFSCHITZ, S. **Database Allocation Strategies for Parallel BLAST Evaluation on Clusters**. Distributed and Parallel Databases 13(1), 2003, pp. 99-127.

DARLING, A.; CAREY, L.; FENG , W. **The Design, Implementation, and Evaluation of mpiBLAST.** 4th International Conference on Linux Clusters, 2003.

DOOLITTLE, R. **Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences Methods in Enzymology.** vol. 183, Academic Press, 1990.

LEMOS, M.; LIFSCHITZ, S. **A Study of a Multi-Ring Buffer Management for BLAST**. 1st International Workshop on Biological Data Management, In conjunction with DEXA, 2003, pp. 5-9.

LIN, H.; MA, X.; CHANDRAMOHAN, P.; GEIST, A.; SAMATOVA, N. **Efficient Data Access for Parallel BLAST**. IEEE International Parallel & Distributed Symposium, 2005.

MARTIN, C.; NARAYANAN, P. S.; ÖZDEN, B.; RASTOGI, R.; SILBERSCHATZ, A. **The Fellini Multimedia Storage Server**. Multimedia Information Storage and Management, S.M.Chung, Kluwer Academic Publishers, 1996, pp. 117-146.

MAURO, R.; LIFSCHITZ, S. **An I/O Device Driver for Bioinformatics Tools: the case for BLAST**. III Brazilian Workshop on Bioinformatics, 2004.

NCBI. **NCBI BLAST.** Disponível em: <http://www.ncbi.nlm.nih.gov/BLAST/>. Acesso em: ago. 2006.

_____. **NCBI Data in XML**. Disponível em: <http://www.ncbi.nlm.nih.gov/data_specs/NCBI_data_in_XML.html>. Acesso em: ago. 2006.

PEARSON, W. **Searching Protein Sequence Libraries: Comparison of the Sensitivity and Selectivity of the Smith-Waterman and FASTA algorithms.** Genomics 11, 1991, pp.635-650.

RED HAT. **Red Hat.** Disponível em: <www.redhat.com>. Acesso em: ago. 2006.

_____. **Fedora Project.** Disponível em: <http://fedora.redhat.com>. Acesso em: ago. 2006.

ROSA, J. **Um Estudo de Compactação de Dados para Biosseqüências**, dissertação de mestrado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2006.

SAKAMOTO, N.; USHIJIMA, K. **Designing and Integrating Human Genome Databases with Object-Oriented Technology**. DEXA, 1994, pp.145-152.

SALZMAN, P.; BURIAN, M.; POMERANTZ, O. **Linux Kernel Module Programming Guide**, v. 2.6.3. Disponível em: <http://www.tldp.org/guides.html>. Acesso em: ago. 2006.

SETÚBAL, J.; MEIDANIS J. **Introduction to Computational Molecular Biology**, PWS Publishing Company, 1997.

SIB. **Swiss-Prot and TrEMBL**. Disponível em: <http://bo.expasy.org/sprot/>. Acesso em: ago. 2006.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts**, McGraw-Hill Companies, Inc, 1997.

STEPHENS, S. M.; CHEN, J. Y.; DAVIDSON, M. G.; THOMAS, S.; TRUTE, B. M. **Oracle Database 10g: a platform for BLAST search and regular expression pattern matching in life sciences**, Nucleic Acids Research, 33. 2005, pp. D675-D679.

UBIC. **UBC Bioinformatics Centre: Formatdb**. Disponível em: <http://bioinformatics.ubc.ca/resources/tools/formatdb>. Acesso em: ago. 2006.

WANG, J.; MU, Q. **Soap-HT-BLAST: high throughput BLAST based on Web services**, Bioinformatics, vol. 19, no. 14, 2003, pp. 1863-1864.

WASHINGTON UNIVERSITY. **WU-BLAST**. Disponível em: <http://blast.wustl.edu>. Acesso em: ago. 2006.

A
Funcionamento do BLAST

O programa do BLAST implementa em uma heurística para o alinhamento local de biosseqüências. Estas consistem em seqüências de nucleotídeos, formadas por repetições das letras *a*, *t*, *c* e *g*, e seqüências de aminoácidos, formadas por repetições das 20 letras respectivas a cada um dos 20 aminoácidos existentes. Exemplos de pequenas seqüências de nucleotídeos e aminoácidos seriam, respectivamente, "cgtggcgagacct" e "gdiikdpfatinev".

Alinhamentos locais são usados para encontrar regiões nas quais duas seqüências possuem alto grau de similaridade, sendo diferente do alinhamento global, que busca alinhar pares de seqüências em toda a extensão. O BLAST compara seqüências de entrada com todas as seqüências de um banco de dados, realizando alinhamentos locais.

A idéia do algoritmo é permitir que buscas semi-ótimas em bancos de dados de seqüências sejam realizadas com desempenho aceitável (Altschul et al., 1990), já que os algoritmos conhecidos para o cálculo de alinhamentos locais ótimos de biosseqüências apresentam desempenhos muito ruins (Setúbal e Meidanis, 1997). O BLAST não busca encontrar alinhamentos locais ótimos entre uma dada seqüência de consulta e todas as seqüências de um banco de dados, e sim HSPs (*High Scoring Pairs*), que são pares de segmentos de seqüências de alta pontuação quando alinhados. O cálculo da pontuação se difere para nucleotídeos e proteínas. Para determinar a pontuação de um par de segmentos de nucleotídeos, basta contar em quantas posições dos segmentos as letras de ambos são iguais e em quantas posições estas são diferentes. Já no caso em que o par de segmentos é de aminoácidos, são usadas matrizes de pontuação, como a PAM e a BLOSUM (Bedell et al., 2003), as quais associam pares de aminoácidos com a pontuação do alinhamento do par em questão, levando em conta semelhanças funcionais e evolutivas existentes entre os aminoácidos.

O BLAST se baseia no fato de que um alinhamento local de alta pontuação entre duas seqüências possui diversos pequenos trechos de casamento dos caracteres das mesmas. Assim, para encontrar os HSPs, um método de alinhamento por palavras é utilizado, através do qual palavras de um tamanho fixo extraídas da seqüência de consulta são procuradas na outra seqüência sendo alinhada.

O algoritmo básico do BLAST possui três etapas, que são descritas a seguir (Altschul, 1990):

1. Construção da lista de palavras candidatas – no caso dos nucleotídeos, são geradas todas as $L-w+1$ palavras de tamanho w presentes na seqüência de consulta de tamanho L. No caso dos aminoácidos, são enumeradas todas as 20^w palavras possíveis de um tamanho w fixo, e selecionadas as que possuem pontuação no mínimo igual a um limite T quando alinhadas, sem buracos, com alguma palavra (também de tamanho w) da seqüência de consulta.
2. Determinação dos *hits* no banco de dados – são encontrados todos os casamentos exatos (*hits*) das palavras candidatas com as seqüências do banco de dados.
3. Extensão dos *hits* – cada casamento exato encontrado no passo anterior é estendido até que a sua pontuação caia um valor X abaixo da melhor pontuação encontrada para extensões menores.

O BLAST não garante que o alinhamento local ótimo será encontrado, porém são utilizadas técnicas estatísticas para apresentar medidas de qualidade dos resultados encontrados nas buscas. Alguns parâmetros podem ser definidos pelo usuário para refinar a busca do BLAST, como o tamanho das palavras candidatas e a máxima queda de pontuação durante as extensões. Dependendo dos parâmetros selecionados, o BLAST pode apresentar no seu resultado mais seqüências consideradas similares à de entrada ou limitar as mesmas.

No passo 2 do algoritmo, a implementação do NCBI optou por se utilizar de uma tabela de busca. Um esquema eficiente para o cálculo incremental dos endereços das palavras do banco de dados é utilizado na implementação. Com isto, cada letra do banco de dados é lida apenas uma vez durante a etapa de busca.

O WU-BLAST, até onde pôde ser verificado na implementação de sua versão 1.4, utiliza um autômato finito determinístico para a execução do passo 2 da estratégia BLAST.

Uma característica comum nas implementações é a possibilidade de filtragem de uma seqüência de consulta. Uma das características de biosseqüências é apresentar longas regiões de baixa complexidade, ou seja, repetições de pequenos motivos. Num alinhamento, estas regiões possuem grande relevância estatística, mas pouco significado biológico. Existem ferramentas computacionais que podem ser chamadas a partir do BLAST para realizar a filtragem de regiões de baixa complexidade.

B
Gerência de Memória em Bancos de Dados

Um banco de dados é, de modo geral, armazenado em diversos arquivos mantidos pelo sistema operacional. Cada arquivo é dividido em blocos, que são as unidades de alocação de armazenamento físico (no disco) e de transferência de dados. A CPU, que é a unidade central de processamento, trabalha diretamente com a memória, e não com discos. Como o tempo gasto durante a transferência de um bloco para a memória é imensamente maior que o tempo gasto na leitura de dados que já estão disponíveis na memória, um dos principais objetivos de um SGBD é minimizar o número de transferências de blocos do disco para a memória. Isto é feito maximizando-se a chance de que os blocos acessados já estejam na memória principal no momento da requisição, caso contrário será necessário acessar o disco (Silberschatz et al., 1997).

Como normalmente não é possível manter todos os blocos de um banco de dados na memória principal, o SGBD deve gerenciar a alocação do espaço disponível na memória para o armazenamento destes. A parte da memória principal disponível para o armazenamento de cópias dos blocos de disco é denominada *buffer* (Martin et al., 1996). Os dados em um *buffer* são organizados como um conjunto de páginas.

O gerenciamento da memória consiste na decisão de como transferir os dados para as páginas do *buffer* e quais páginas devem ser substituídas. Uma estratégia muito comum para a transferência de dados para a memória é a previsão de quais blocos do disco serão acessados no futuro próximo e a cópia destes para a memória antes mesmo de serem requisitados. Esta estratégia é denominada *prefetching* (Silberschatz et al., 1997). Já as políticas de substituição de páginas mais usadas são a LRU (*Least Recently Used*), a MRU (*Most Recently Used*) e a FIFO (*First-In-First-Out*). A política LRU escolhe para substituição a página menos recentemente usada, sendo o oposto da MRU, que escolhe a página mais

recentemente usada. Já a política FIFO elege para substituição a página mais antiga na memória, independente da sua utilização. Como os SGBDs têm conhecimento a priori dos padrões de acesso de páginas usados pelas aplicações, estes podem utilizar diversas políticas de substituição de páginas diferentes.

C
Drivers de Dispositivos para o Linux

Um *driver* de dispositivo é uma camada de software dependente de máquina que interage diretamente com o *hardware*. Para cada periférico conectado a uma máquina deve haver um *driver* de dispositivo associado. Os *drivers* utilizam funções do dispositivo controlado e do sistema operacional e oferecem, em troca, os recursos do dispositivo controlado. Por sua vez, o sistema operacional pode controlar o acesso das aplicações de mais alto nível a estes recursos.

No Linux, um *driver* pode ser implementado de duas formas: estaticamente ligado ao núcleo ou dinamicamente carregado como módulo. A primeira forma consiste em modificar o núcleo do sistema operacional, acrescentando-lhe código, e compilando-o para que o *driver* seja carregado junto com o núcleo. A segunda alternativa consiste em criar o *driver* separadamente e carregá-lo, em um núcleo já em execução, apenas quando necessário. A vantagem da utilização de *drivers* carregados como módulos dinâmicos é que o núcleo pode possuir apenas os *drivers* mais elementares ligados estaticamente (aqueles indispensáveis ao funcionamento do sistema operacional, como memória, CPU, etc), e possuir uma grande quantidade de *drivers* de dispositivo carregados na medida em que sejam necessários (placas de rede, placas de som, controladores, etc).

Os dispositivos são normalmente representados no sistema de arquivos do Linux como arquivos especiais, os quais têm localização padrão no diretório */dev*. Os dispositivos e os seus arquivos especiais são classificados em três tipos:

- **Dispositivos de caractere:** são aqueles que podem ser acessados como uma seqüência de *bytes*, do mesmo modo que um arquivo comum. O *driver* é acessado por meio dos arquivos especiais associados ao mesmo, reimplementando as funções que realizarão operações nestes arquivos. As funções reimplementadas são *open*, *close*, *read*, *write*, *llseek*, etc. Exemplos de dispositivos de caractere são o console e as portas seriais.

- **Dispositivos de bloco:** da mesma forma que um dispositivo de caractere representa um arquivo, um dispositivo de bloco representa um disco ou um sistema de arquivos. O dispositivo de bloco é normalmente acessado em blocos, embora funções de acesso seqüencial (a caractere) possam ser implementadas. Nesse caso, a diferença será apenas na forma como os dados são tratados internamente. As funções reimplementadas pelos *drivers* de dispositivos de bloco são *open*, *release*, *ioctl*, *check_media_change*, etc. Exemplos de dispositivos de blocos são os discos rígidos e discos flexíveis.
- **Dispositivos de rede:** são os dispositivos acessados em pacotes. Esses dispositivos não possuem uma entrada no sistema de arquivos e são acessadas por funções especiais de transmissão de pacotes. Um exemplo de dispositivo de rede é a interface Ethernet.

C.1.
Números Maior e Menor

Os dispositivos e seus arquivos especiais são classificados no Linux por um número chamado "número maior". Como o arquivo especial é apenas uma interface para o *driver* associado a um dispositivo, o número maior identifica a que *driver* o arquivo especial está associado. A lista dos nomes dos dispositivos associados aos números maiores pode ser consultada no arquivo */proc/devices*. Quando um acesso é realizado a um arquivo especial, essa lista é acessada para identificar o *driver* correspondente. O *driver* das unidades de disco flexível, por exemplo, está associado ao número maior 2. Números maiores nas faixas de 60 a 63, 120 a 127 e 240 a 254, são reservados para uso local e experimental.

Além de um número maior, um arquivo especial é associado a um número menor. Este é utilizado pelo *driver* para distinguir os dispositivos associados ao mesmo *driver*. As unidades de disco flexível */dev/fd0* e */dev/fd1*, por exemplo, possuem números menores 0 e 1, respectivamente, e número maior 2. A criação de arquivos especiais é feita através do comando *mknod*.

C.2.
Operações em um Arquivo Especial

Um arquivo aberto pelo sistema operacional, seja ele um arquivo comum ou especial, possui uma estrutura do tipo *file* associada. Esta guarda um apontador para uma lista de funções que manipulam os diversos tipos de acesso ao arquivo em questão. Para o caso de um arquivo especial a caractere, as funções que realizarão operações no arquivo, e conseqüentemente no dispositivo, são definidas pelo *driver* daquele dispositivo.

A definição das operações é realizada através de uma estrutura do tipo *file_operations*, detalhada na figura 28. Esta estrutura possui um conjunto de apontadores que mapeiam chamadas de sistema como *open* e *read* em funções específicas do dispositivo, implementadas pelo *driver*.

Quando o arquivo é aberto, o sistema operacional identifica o *driver* responsável e associa ao arquivo aberto à estrutura *file_operations* implementada pelo *driver*. Entretanto, um *driver* pode ter diferentes funcionalidades de acordo com o número menor associado ao arquivo especial.

Desta forma, quando a função *open* implementada pelo *driver* é chamada, o mesmo pode desejar mudar as operações associadas de acordo com o número menor do arquivo especial aberto. Este procedimento é possível pois, além da estrutura do tipo file, um arquivo aberto possui uma estrutura do tipo *inode*. Esta estrutura fornece informações sobre a entrada no sistema de arquivos que está sendo aberta e pode ser utilizada para extrair o número menor do arquivo especial em questão.

```
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
   int (*readdir) (struct file *, void *, filldir_t);
   unsigned int (*poll) (struct file *, struct poll_table_struct *);
   int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *);
   int (*release) (struct inode *, struct file *);
   int (*fsync) (struct file *, struct dentry *, int datasync);
   int (*fasync) (int, struct file *, int);
   int (*lock) (struct file *, int, struct file_lock *);
   ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
loff_t *);
   ssize_t (*writev) (struct file *, const struct iovec *, unsigned
long, loff_t *);
   ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
loff_t *, int);
   unsigned long (*get_unmapped_area)(struct file *, unsigned long,
unsigned long, unsigned long, unsigned long);
};
```

Figura 1: Estrutura file_operations

Os arquivos especiais de bloco apresentam um comportamento um pouco diferente. Isso ocorre porque dispositivos de bloco ainda estão intimamente ligados ao núcleo do Linux. A estrutura que contém os apontadores para as funções que manipulam as operações em arquivos especiais de bloco está ilustrada na figura 29. Além das funções listadas na figura, os arquivos especiais de bloco necessitam de alguns manipuladores que serão cadastrados diretamente no núcleo e estão fora desta estrutura.

```
struct block_device_operations {
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   int (*ioctl) (struct inode *, struct file *, unsigned, unsigned
long);
   int (*check_media_change) (kdev_t);
   int (*revalidate) (kdev_t);
   struct module *owner;
};
```

Figura 2: Estrutura block_device_operations

C.3.
Alocação de Memória

De forma semelhante à alocação de memória no nível de usuário, a alocação dentro do núcleo do Linux é feita através da função *kmalloc()*, a qual recebe dois parâmetros: o tamanho em bytes do bloco de memória a ser alocado e um valor de prioridade que indica o tipo e a urgência da alocação.

Uma observação importante é que o núcleo aloca a memória procurando um bloco de memória com tamanho adequado em um *pool* de objetos de memória. Desta forma, quando um usuário solicita a alocação de uma determinada quantidade de memória, provavelmente receberá um bloco maior do que o solicitado. Maiores detalhes sobre os blocos de memória podem ser obtidos no arquivo *mm/slab.c*. Por outro lado, a liberação de memória acontece utilizando a função *kfree()*.

C.4.
Execução de Tarefas

Um *driver* pode precisar executar tarefas em três situações: em resposta a uma solicitação de um processo, em resposta a um evento no *hardware* (*i.e.* interrupções de hardware) e espontaneamente.

C.4.1.
Execução em resposta a uma solicitação de um processo

Esta primeira situação acontece quando algum processo ou *thread* chama uma função do *driver* ou acessa um arquivo especial associado ao *driver*. Neste caso, a operação de acesso ao arquivo especial executará diretamente a função do *driver* associada àquela operação. A execução acontece portanto sob o contexto do processo que realizou a operação de acesso.

C.4.2.
Execução em resposta a interrupções

A segunda situação acontece quando o *driver* captura uma linha de interrupção, conforme ilustrado na seção anterior. Desta forma, quando algum evento importante acontece no dispositivo controlado, este sinaliza o evento através de uma interrupção. Caso a interrupção não esteja mascarada, o sistema operacional consulta imediatamente a tabela de interrupções e executa a rotina associada àquela interrupção. Durante a execução desta rotina, o *driver* pode realizar apenas algumas tarefas simples, por exemplo, no caso de uma placa de rede ela poderia copiar os dados da memória do dispositivo para a memória RAM. Entretanto, a rotina de tratamento de interrupção não pode realizar operações complexas ou demoradas, por exemplo, ela não pode escrever ou ler de arquivos. Uma forma de resolver este problema é dividindo o tratamento das interrupções em duas partes, a parte rápida, chamada de *top half* e a parte lenta, chamada de *bottom half*. A parte rápida deve realizar o mínimo de operações possível e depois agendar a execução da parte lenta. Quando todas as partes rápidas das interrupções terminarem de executar e antes que qualquer processo ou *thread* seja escalonado, as partes lentas cujas execuções foram agendadas serão executadas.

A forma mais comum de se construir a parte lenta de uma interrupção é através de *tasklets*. Uma *tasklet* é uma estrutura que contém um apontador para uma função que realiza as tarefas complexas que não poderiam ser realizadas durante a parte rápida do tratamento da interrupção.

C.4.3.
Execução espontânea

No terceiro caso, um *driver* pode desejar executar tarefas independentemente do acontecimento de eventos (interrupções ou chamadas do usuário). Para isso o *driver* pode criar uma *thread* do núcleo (*Kernel Thread*). Essa abordagem pode ser necessária, por exemplo, caso o *driver* não use interrupções e, por isso, precise acessar periodicamente o dispositivo para verificar se algum evento relevante aconteceu. Em execução, a *thread* pode realizar operações de E/S, inclusive operações bloqueantes.

A criação de *threads* do núcleo é realizada através da função *kernel_thread()*. É importante observar porém que quando a *thread* é criada, ela é criada sob o contexto do programa que executou a operação que resultou na sua criação. Por exemplo, se o *driver* cria uma *thread* no momento do seu carregamento, o que é bastante freqüente, a *thread* será criada sob o contexto do processo *insmod* ou *modprobe*. Deste modo, após criada a *thread* deve ser desacoplada do processo que estava executando durante sua criação.

C.5.
Controle de Concorrência

Enquanto um aplicativo comum raramente precisa considerar aspectos de execução concorrente de suas funções, um *driver* de dispositivo precisa estar preparado para que suas funções sejam executadas simultaneamente. Por exemplo, um *driver* pode estar associado a um arquivo especial e várias aplicações podem, simultaneamente, estar acessando esse arquivo. Estas condições de corrida podem acontecer tipicamente em duas situações: quando um processo é bloqueado por uma das funções do *driver*, liberando o processador para um novo processo que pode acessar a mesma função, ou quando aplicações em uma máquina multi-processada executam a mesma função do *driver* concorrentemente, em diferentes processadores. Desta forma, o controle de concorrência deve ser feito de modo que as estruturas utilizadas em diferentes fluxos de execução sejam mantidas

separadas e que as áreas compartilhadas (*i.e.* variável globais compartilhadas) sejam acessadas de forma a não corromper os dados.

Existem diversas formas de se controlar a concorrência no núcleo, sendo a utilização de semáforos uma das mais importantes. Semáforos são bastante parecidos com *spinlocks*, não permitindo que um fluxo de execução prossiga, a não ser que a trava esteja aberta. A grande diferença é que, se por um lado o *spinlock* executa uma espera ocupada enquanto a chave não é liberada, o semáforo bloqueia o processo até a liberação da mesma. Desta forma, se grandes esperas estão envolvidas, semáforos são mais adequados.

C.6.
Bloqueio de Processos

Quando uma aplicação acessa um arquivo especial tentando recuperar uma informação, é possível que o *driver* não possua essa informação pronta imediatamente e pode ser necessário buscá-la no dispositivo controlado, por exemplo. Esse é o caso dos discos de armazenamento: quando uma aplicação lê de um disco, ela pode permanecer bloqueada enquanto os dados são recuperados do disco físico; mais tarde, quando o *driver* dispõe dessas informações, ele desbloqueia o processo. O bloqueio de processos pode ser feito de várias formas, por exemplo através de semáforos. No entanto, todas as formas são baseadas em um mecanismo chamado “fila de espera”, nas quais os processos são bloqueados para serem acordados posteriormente.

C.7.
Estrutura Básica de um Módulo do Linux

Um módulo é um pedaço de código compilado que pode ser dinamicamente acoplado ao núcleo do Linux. Este pedaço de código previamente compilado pode possuir referências a funções e estruturas exportadas pelo núcleo. Essas referências serão resolvidas no momento do carregamento e, caso alguma das funções utilizadas pelo módulo não seja encontrada, uma mensagem de erro é

exibida e o módulo não é carregado. As funções exportadas pelo núcleo são funções do próprio núcleo ou funções exportadas por outros módulos carregados anteriormente. A lista de funções exportadas é dinâmica e pode ser encontrada no arquivo */proc/ksyms*.

Os módulos são formas flexíveis para implementação de funcionalidades acessórias do sistema operacional e não apenas *drivers* de dispositivos. Dois exemplos bastante freqüentes de funcionalidades são sistemas de arquivos (por exemplo, suporte a FAT e NTFS) e protocolos de rede (por exemplo, suporte a IPv6 ou IPX). Os módulos que implementam suporte a sistemas de arquivos (ou protocolos de rede) funcionam então como software *drivers*, transformando estruturas de alto nível em estruturas de baixo nível que serão repassadas para o *driver* do dispositivo de disco (ou de rede).

Um módulo é um programa, normalmente em linguagem C, com duas funções principais: as funções *init_module()* e *cleanup_module()*. A função *init_module()* é a função chamada no momento que o módulo é carregado (por exemplo, por um comando *modprobe* ou *insmod*). Esta função deve realizar as operações necessárias para a iniciação do módulo, alocando os recursos necessários e iniciando o dispositivo de hardware controlado. Caso esta função retorne 0, o módulo será mantido na memória. Caso contrário, o sistema operacional exibe uma mensagem de erro e remove o módulo da memória. Já a função *cleanup_module()* é aquela chamada no momento de remoção de um módulo, tendo a responsabilidade de liberar os recursos alocados pelo mesmo.

Referências mais detalhadas sobre *drivers* de dispositivos, módulos e suas implementações podem ser obtidas em Corbet, et al. (2005) e Salzman et al. (2005).