Revista de Informática Teórica e Aplicada - RITA - Volume VIII, Número 3, dezembro de 2001

Sistemas Operacionais

Rômulo Silva de Oliveira 1

Alexandre da Silva Carissimi 2

Simão Sirineo Toscani 3

Resumo: Ao longo de mais de 40 anos, sistemas operacionais têm sido desenvolvidos com o propósito de tornar a utilização do computador mais eficiente e mais conveniente. Para isso, um enorme número de conceitos, abstrações, mecanismos e algoritmos foram criados e aprimorados. Este artigo é um tutorial a respeito das técnicas fundamentais empregadas nos sistemas operacionais contemporâneos. Também são discutidos aqui os sistemas operacionais distribuídos e de tempo real.

Abstract: For more than 40 years, operating systems have been developed with the goal of making the utilization of computers more efficient and more comfortable. A huge number of concepts, abstractions, mechanisms and algorithms were created and improved. This paper is a tutorial on the fundamental techniques applied in the construction of contemporary operating systems. We also discuss distributed operating systems and real-time operating systems.

1 Dep. de Automação e Sistemas, UFSC, Caixa Postal 476, 88040-900, Florianópolis-SC
romulo@das.ufsc.br
2 Instituto de Informática, UFRGS, Caixa Postal 15064, 91501-970, Porto Alegre-RS
asc@inf.ufrgs.br
3 Instituto de Informática, UFRGS, Caixa Postal 15064, 91501-970, Porto Alegre-RS
simao@inf.ufrgs.br

1 Introdução

O sistema operacional procura tornar a utilização do computador mais eficiente e mais conveniente. A utilização mais eficiente busca um maior retorno no investimento feito no hardware, significa mais trabalho obtido do mesmo hardware. Uma utilização mais conveniente vai diminuir o tempo necessário para a construção e utilização dos programas. Um enorme número de conceitos, abstrações, mecanismos e algoritmos foram criados e aprimorados ao longo dos últimos 40 anos. Este artigo é um tutorial a respeito das técnicas fundamentais empregadas nos sistemas operacionais contemporâneos. O artigo baseia-se, em grande parte, no texto do livro “Sistemas Operacionais”, dos mesmos autores [1].

Para atingir os objetivos propostos, o sistema operacional oferece diversos tipos de **serviços**. Todo sistema operacional oferece meios para que um programa seja carregado na memória principal e executado. Talvez o serviço mais importante oferecido seja o que permite a utilização de arquivos e diretórios. Também o acesso aos periféricos é feito através do sistema operacional. À medida que diversos usuários compartilham o computador, passa a ser interessante saber quanto de quais recursos cada usuário necessita. Diversas informações sobre o estado do sistema são mantidas. Nessa categoria, temos a hora e a data correntes, a lista de usuários utilizando o computador no momento, a versão do sistema operacional em uso. Cabe também ao sistema operacional garantir que cada usuário possa trabalhar sem sofrer interferência danosa dos demais.

Os programas solicitam serviços ao sistema operacional através das **chamadas de sistema**. Elas são semelhantes às chamadas de sub-rotinas. Entretanto, enquanto as chamadas de sub-rotinas são transferências para procedimentos normais do programa, as chamadas de sistema transferem a execução para o sistema operacional. Através de parâmetros, o programa informa exatamente o que necessita. O retorno da chamada de sistema, assim como o retorno de uma sub-rotina, faz com que a execução do programa seja retomada a partir da instrução que segue a chamada. Para o programador *assembly* (linguagem de montagem), as chamadas de sistema são bastante visíveis. Por exemplo, o conhecido "INT 21H" no MS-DOS. Em uma linguagem de alto nível, elas ficam escondidas dentro da biblioteca utilizada pelo compilador. O programador chama sub-rotinas de uma biblioteca, e estas chamam o sistema. Por exemplo, qualquer função da biblioteca que acesse o terminal (como `printf()` na linguagem C) exige uma chamada de sistema.

A parte do sistema operacional responsável por implementar as chamadas de sistema é normalmente chamada de **núcleo** ou ***kernel***. Os principais componentes do *kernel* de qualquer sistema operacional são a **gerência de processador**, a **gerência de memória**, o **sistema de arquivos** e a **gerência de entrada e saída**. Cada um desses componentes será descrito nas próximas seções.

Os **programas de sistema**, algumas vezes chamados de **utilitários**, são programas normais executados fora do *kernel* do sistema operacional. Eles utilizam as mesmas chamadas de sistema disponíveis aos demais programas. Esses programas implementam

tarefas básicas para a utilização do sistema e muitas vezes são confundidos com o próprio sistema operacional. Exemplos são os utilitários para manipulação de arquivos: programas para exibir arquivo, imprimir arquivo, copiar arquivo, trocar o nome de arquivo, listar o conteúdo de diretório, entre outros. O mais importante programa de sistema é o **interpretador de comandos**. Esse programa é ativado pelo sistema operacional sempre que um usuário inicia sua sessão de trabalho. Sua tarefa é receber comandos do usuário e executá-los. Para isso, ele recebe as linhas tecladas pelo usuário, analisa o seu conteúdo e executa o comando teclado. A execução do comando, na maioria das vezes, vai exigir uma ou mais chamadas de sistema. Por exemplo, considere um comando do tipo "lista diretório". Para executá-lo, o interpretador de comandos deve, primeiramente, ler o conteúdo do diretório solicitado pelo usuário. A informação é formatada para facilitar a sua disposição na tela e, finalmente, novas chamadas de sistema serão feitas para listar essas informações na tela. O interpretador de comandos não precisa, obrigatoriamente, ser um programa de sistema. Ele pode fazer parte do sistema operacional. Entretanto, a solução descrita antes é a que oferece a maior flexibilidade. O que foi dito sobre o interpretador de comandos é igualmente válido para a situação em que o sistema operacional oferece uma **interface gráfica de usuário** (**GUI** – *graphical user interface*). A diferença está na comodidade para o usuário, que passa a usar ícones, menus e mouse para interagir com o sistema.

2 Gerência do Processador

Em um **sistema multiprogramado** diversos programas são mantidos na memória ao mesmo tempo. Vamos supor que o sistema operacional inicia a execução do programa 1. Após algum tempo, da ordem de milisegundos, o programa 1 faz uma chamada de sistema. Ele solicita algum tipo de operação de entrada ou saída. Por exemplo, uma leitura do disco. Sem multiprogramação, o processador ficaria parado durante a realização do acesso. Em um sistema multiprogramado, enquanto o periférico executa o comando enviado, o sistema operacional inicia a execução de outro programa. Por exemplo, o programa 2. Dessa forma, processador e periférico trabalham ao mesmo tempo. Enquanto o processador executa o programa 2, o periférico realiza a operação solicitada pelo programa 1.

Em geral é conveniente diferenciar um programa de sua execução. Para tanto é usado o conceito de processo. Não existe uma definição objetiva, aceita por todos, para a idéia de processo. Na maioria das vezes, um **processo** é definido como "um programa em execução". O conceito de processo é essencial no estudo de sistemas operacionais. Um **programa** é uma seqüência de instruções. É algo passivo e não altera o seu próprio estado. O processo é um elemento ativo, que altera o seu estado, à medida que executa um programa. É o processo que faz chamadas de sistema, ao executar um programa.

É possível que vários processos executem o mesmo programa ao mesmo tempo. Por exemplo, diversos usuários podem estar utilizando simultaneamente o editor de texto favorito da instalação. Existe um único programa "editor de texto". Para cada usuário, existe um processo executando o programa. Cada processo representa uma execução independente

do editor de textos. Todos os processos utilizam uma mesma cópia do código do editor de textos, porém cada processo trabalha sobre uma área de variáveis privativa.

2.1 Propriedades dos Processos

Processos são criados e destruídos. O momento e a forma pela qual eles são criados e destruídos depende do sistema operacional em consideração. Alguns sistemas trabalham com um número fixo de processos. A forma mais flexível de operação é permitir que processos possam ser criados livremente, através de chamadas de sistema. Além da chamada de sistema "cria processo", serão necessárias chamadas para "autodestruição do processo" e também para "eliminação de outro processo".

A maioria dos processos de um sistema executam programas dos usuários. Entretanto, alguns podem realizar tarefas do sistema. São **processos do sistema** (***daemons***), não dos usuários. Por exemplo, para evitar conflitos na utilização da impressora, muitos sistemas trabalham com uma técnica chamada ***spooling***. Para imprimir um arquivo, o processo de usuário deve colocá-lo em um diretório especial. Um processo do sistema copia os arquivos desse diretório para a impressora. Dessa forma, um processo de usuário nunca precisa esperar a impressora ficar livre, uma vez que ele não envia os dados para a impressora, mas sim para o disco. O processo que envia os dados para a impressora não está associado a nenhum usuário. É um processo do próprio sistema operacional.

Alguns sistemas suportam o conceito de **grupo de processos**. Por exemplo, todos os processos associados a um mesmo terminal podem formar um grupo. Em muitos sistemas os processos são criados por outros processos, através de uma chamada de sistema. Nesse caso, é possível definir uma **hierarquia de processos**. O processo que faz a chamada de sistema é chamado de **processo pai**. O processo criado é chamado de **processo filho**. Um mesmo processo pai pode estar associado a vários processos filhos. Os processos filhos podem criar outros processos. Essa situação é facilmente representada através de uma árvore.

A descrição do funcionamento da multiprogramação mostrou diversos momentos pelos quais passa o processo. A partir dessa descrição, pode-se estabelecer os **estados** possíveis para um processo. Após ser criado, o processo entra em um ciclo de processador. Entretanto, o processador poderá estar ocupado com outro processo, e ele deverá esperar. Diversos processos podem estar nesse mesmo estado. Em **máquinas multiprocessadoras** existem diversos processadores. Nesse caso, diversos processos executam ao mesmo tempo. Porém, essa não é a situação mais comum. Vamos supor que existe um único processador no computador. Nesse caso, é necessário manter uma fila com os processos aptos a ganhar o processador. Essa fila é chamada "**fila de aptos**" (*ready queue*).

A Figura 1 mostra o **diagrama de estados** de um processo. No **estado executando**, um processo pode fazer chamadas de sistema. Até a chamada de sistema ser atendida, o processo não pode continuar sua execução. Ele fica bloqueado e só volta a disputar o processador após a conclusão da chamada. Enquanto espera pelo término da chamada de sistema, o processo está no **estado bloqueado** (*blocked*).

A mudança de estado de qualquer processo é iniciada por um evento. Esse evento aciona o sistema operacional, que então altera o estado de um ou mais processos. A transição do estado executando para bloqueado é feita através de uma chamada de sistema. Uma chamada de sistema é necessariamente feita pelo processo no estado executando. Ele fica no estado bloqueado até o atendimento. Com isso, o processador fica livre. O sistema operacional então seleciona um processo da fila de aptos para receber o processador. O processo selecionado passa do estado de apto para o estado executando. O módulo do sistema operacional que faz essa seleção é chamado **escalonador** (*scheduler*).

Outro tipo de evento corresponde às interrupções do hardware. Elas, em geral, informam o término de uma operação de E/S. Isso significa que um processo bloqueado será liberado. O processo liberado passa do estado de bloqueado para o estado de apto. Ele volta a disputar o processador com os demais da fila de aptos.

Alguns outros caminhos também são possíveis no grafo de estados. A destruição do processo pode ser em função de uma chamada de sistema ou por solicitação do próprio processo. Entretanto, alguns sistemas podem resolver abortar o processo, caso um erro crítico tenha acontecido durante uma operação de E/S. Nesse caso, passa a existir um caminho do estado bloqueado para a destruição. Algumas chamadas de sistema são muito rápidas. Por exemplo, leitura da hora atual. Não existe acesso a periférico, mas apenas consulta às variáveis do próprio sistema operacional. Nesse caso, o processo não precisa voltar para a fila de aptos. Ele simplesmente retorna para a execução após a conclusão da chamada. Muitos sistemas procuram evitar que um único processo monopolize a ocupação do processador. Se um processo está há muito tempo no processador, ele volta para o fim da fila de aptos. Um novo processo da fila de aptos ganha o processador. Esse mecanismo cria um caminho entre o estado executando e o estado apto.

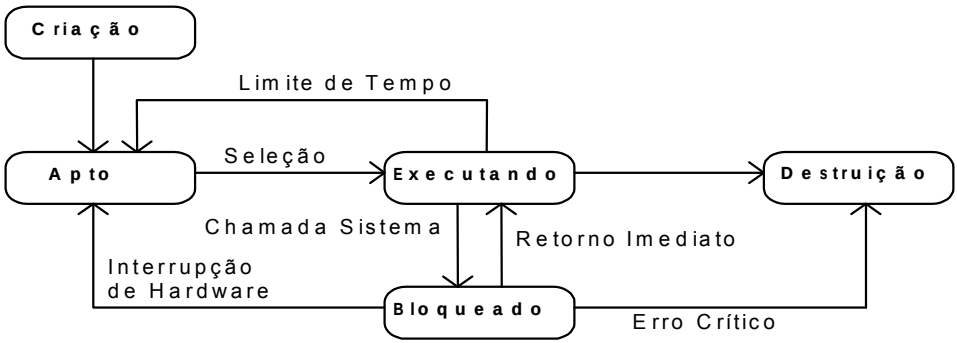

Figura 1. Diagrama de estados de um processo

O sistema operacional é responsável por implementar uma proteção apropriada para o sistema. Para isso é necessário o auxílio da arquitetura do processador (hardware). A forma usual é definir dois **modos de operação** ou **níveis de proteção** para o processador. Pode-se chamá-los de **modo usuário** e **modo supervisor**. Quando o processador está em modo supervisor, não existem restrições, e qualquer instrução pode ser executada. Em modo

usuário, algumas instruções não podem ser executadas. Essas instruções são chamadas de **instruções privilegiadas**. Se um processo de usuário tentar executar uma instrução privilegiada em modo usuário, o hardware automaticamente gera uma interrupção e aciona o sistema operacional, o qual poderá abortar o processo de usuário. As interrupções, além de acionarem o sistema operacional, também chaveiam automaticamente o processador para modo supervisor. Nesse mecanismo, o sistema operacional executa com o processador em modo supervisor. Os processos de usuário executam em modo usuário. Quando é ligado o processador, ele inicia em modo supervisor. Sempre antes de entregar o processador para um processo de usuário, o sistema operacional comuta o processador para modo usuário.

2.2 Bloco Descritor de Processo

Existem várias informações que o sistema operacional deve manter a respeito dos processos. No "programa" sistema operacional, um processo é representado por um registro. Esse registro é chamado de **bloco descritor de processo** ou simplesmente **descritor de processo** (DP). No DP, fica tudo que o sistema operacional precisa saber sobre o processo. Um processo quase sempre faz parte de alguma fila e os próprios descritores de processo são utilizados como elementos dessas filas, implementadas como listas encadeadas. Abaixo está uma lista de campos normalmente encontrados no DP:

- Prioridade do processo no sistema, usada para definir a ordem na qual os processos recebem o processador;

- Localização e tamanho da memória principal ocupada pelo processo;

- Identificação dos arquivos abertos no momento;

- Informações para contabilidade, como tempo de processador gasto, espaço de memória ocupado, etc;

- Estado do processo: apto, executando, bloqueado;

- Contexto de execução quando o processo perde o processador, ou seja, conteúdo dos registradores do processador quando o processo é suspenso temporariamente;

- Apontadores para encadeamento dos blocos descritores de processo.

Para criar um processo, um descritor é retirado da lista de descritores livres. O próximo passo é completar os campos do descritor alocado com valores apropriados. Por exemplo, o programa a ser executado pelo processo deve ser localizado no disco, e uma área de memória grande o suficiente para ele deve ser alocada. O programa pode então ser carregado do disco para a memória principal. Essas tarefas exigem a participação dos módulos de gerência de memória e sistema de arquivos. Para a gerência do processador, é simplesmente fornecida a informação "endereço e tamanho" da área de memória alocada para o processo. Essa informação é copiada para o descritor do processo. Quando todos os campos estiverem preenchidos, o descritor do processo é inserido na "lista de espera pelo processador". A partir desse momento, o processo passa a disputar tempo de processador,

junto com os demais. Em outras palavras, o processo foi criado. A base da multiprogramação é o compartilhamento do processador entre os processos. Em um sistema multiprogramado, é necessário interromper processos para continuá-los mais tarde. Essa tarefa é chamada de **chaveamento de processo**, ou **chaveamento de contexto de execução**. O local usado para salvar o contexto de execução de um processo é o seu próprio bloco descritor.

Um processo é uma abstração que reúne uma série de atributos como espaço de endereçamento, descritores de arquivos abertos, permissões de acesso, quotas, etc. Um processo possui ainda áreas de código, dados e pilha de execução. Também é associado ao processo um fluxo de execução. Por sua vez, uma ***thread*** nada mais é que um fluxo de execução. Na maior parte das vezes, cada processo é formado por um conjunto de recursos mais uma única *thread*.

A idéia de ***multithreading*** é associar vários fluxos de execução (várias *threads*) a um único processo. Em determinadas aplicações, é conveniente disparar várias *threads* dentro do mesmo processo (programação concorrente). É importante notar que as *threads* existem no interior de um processo, compartilhando entre elas os recursos do processo, como o espaço de endereçamento (código e dados). Devido a essa característica, a gerência de *threads* (criação, destruição, troca de contexto) é "mais leve" quando comparada com processos. *Threads* são muitas vezes chamadas de **processos leves**.

2.3 Algoritmos de Escalonamento

No **escalonamento do processador** é decidido qual processo será executado a seguir. Na escolha de um algoritmo de escalonamento, utiliza-se como critério básico o objetivo de aumentar a produtividade do sistema e, ao mesmo tempo, diminuir o tempo de resposta percebido pelos usuários. Esses dois objetivos podem tornar-se conflitantes em determinadas situações. Variância elevada significa que a maioria dos processos recebe um serviço (tempo de processador) satisfatório, enquanto alguns são bastante prejudicados. Provavelmente, será melhor sacrificar o tempo médio de resposta para homogeneizar a qualidade do serviço que os processos recebem.

Quando os processos de um sistema possuem diferentes prioridades, essa **prioridade** pode ser utilizada para decidir qual processo é executado a seguir. Considere o cenário no qual números menores indicam processos mais importantes. Um processo com prioridade 5 (vamos chamá-lo P5) está executando. Nesse momento, termina o acesso a disco de um processo com prioridade 2 (P2), e ele está pronto para iniciar um ciclo de processador. Nessa situação, o sistema pode comportar-se de duas formas distintas. O processo que chega na fila do processador respeita o ciclo de processador em andamento, ou seja, o P2 será inserido normalmente na fila. O processo em execução somente libera o processador no final do ciclo de processador. Temos nesse caso a "**prioridade não-preemptiva**". O processo P2, por ser mais importante que o processo em execução, recebe o processador imediatamente. O processo P5 é inserido na fila conforme a sua prioridade. Essa solução é conhecida como "**prioridade preemptiva**".

No método **fatia de tempo**, também conhecido como ***round-robin***, cada processo recebe uma fatia de tempo do processador (***quantum***). Se o processo realizar uma chamada de sistema e ficar bloqueado antes do término da sua fatia, simplesmente o próximo processo da fila recebe uma fatia integral e inicia sua execução. Se terminar a fatia de tempo do processo em execução, ele perde o processador e volta para o fim da fila. Nesse algoritmo não é possível **postergação indefinida** (ficar esperando para sempre), pois processos sempre entram no fim da fila. Um problema é definir o tamanho da fatia de tempo. É preciso levar em conta que o chaveamento entre processos não é instantâneo. Se a fatia de tempo for muito pequena, esse custo cresce em termos relativos. Por outro lado, uma fatia de tempo muito grande destrói a aparência de paralelismo na execução dos processos.

Em um mesmo sistema, normalmente convivem diversos tipos de processos. Por exemplo, em um Centro de Processamento de Dados, os processos da produção (folha de pagamentos, etc) podem ser disparados em ***background*** (execução sem interação com o usuário), enquanto os processos do desenvolvimento interagem com os programadores (execução em ***foreground***). É possível construir um sistema no qual existem **múltiplas filas**, uma fila de processador para cada tipo de processo. Com múltiplas filas, o tipo do processo define a fila na qual ele é inserido, e o processo sempre volta para a mesma fila. Quando o processo pode mudar de fila durante a sua execução, temos **múltiplas filas com realimentação**.

3 Gerência da Memória

Na multiprogramação, diversos processos são executados simultaneamente, através da divisão do tempo do processador. Para que o chaveamento entre eles seja rápido, esses processos devem estar na memória, prontos para executar. É função da gerência de memória do sistema operacional prover os mecanismos necessários para que os diversos processos compartilhem a memória de forma segura e eficiente. A técnica particular que determinado sistema operacional emprega depende, entre outras coisas, de o que a arquitetura do computador em questão suporta. Em [2], [3], [4] e [5], podem ser encontradas as descrições de algumas soluções empregadas em sistemas operacionais específicos. Uma excelente descrição de arquiteturas contemporâneas pode ser encontrada em [6] e [7].

A **memória lógica** de um processo é aquela que o processo enxerga, ou seja, aquela que o processo é capaz de acessar. Os endereços manipulados pelo processo são endereços lógicos. Em outras palavras, as instruções de máquina de um processo especificam endereços lógicos. Por exemplo, um processo executando um programa escrito na linguagem C manipula variáveis tipo *pointer*. Essas variáveis contêm endereços lógicos. Em geral, cada processo possui a sua memória lógica, que é independente da memória lógica dos outros processos. A **memória física** é aquela implementada pelos circuitos integrados de memória, pela eletrônica do computador. O endereço físico é aquele que vai para a memória física, ou seja, é usado para endereçar os circuitos integrados de memória.

O **espaço de endereçamento lógico** de um processo é formado por todos os endereços lógicos que esse processo pode gerar. Existe um espaço de endereçamento lógico por processo. Já o **espaço de endereçamento físico** é formado por todos os endereços aceitos pelos circuitos integrados de memória. A **unidade de gerência de memória** (*Memory Management Unit*, MMU) é o componente do hardware responsável por prover os mecanismos que serão usados pelo sistema operacional para gerenciar a memória. Entre outras coisas, é a MMU que vai mapear os endereços lógicos gerados pelos processos nos correspondentes endereços físicos que serão enviados para a memória.

Quando um programa é carregado em uma área de memória maior que o necessário, isso resulta em um desperdício de memória que é chamado de **fragmentação interna**, isto é, memória perdida dentro da área alocada para um processo. Outra possibilidade é termos duas partições livres, digamos, de 25 e 100 Kbytes. Nesse momento é criado um processo para executar um programa de 110 Kbytes. Observe que a memória total livre no momento é de 125 Kbytes, mas ela não é contígua. Se o programa não pode ser executado devido à forma como a memória é gerenciada, o problema é chamado de **fragmentação externa**, isto é, memória perdida fora da área ocupada por um processo.

3.1 Paginação

O desperdício de memória em função da fragmentação externa é um grande problema. A origem da fragmentação externa está no fato de cada programa necessitar ocupar uma única área contígua de memória. Se essa necessidade for eliminada, ou seja, se cada programa puder ser espalhado por áreas não contíguas de memória, a fragmentação externa é eliminada. Esse efeito é obtido com a **paginação**.

A Figura 2 ilustra o funcionamento da técnica de paginação. O exemplo da figura utiliza um tamanho de memória exageradamente pequeno para tornar a figura mais clara e menor. O espaço de endereçamento lógico de um processo é dividido em **páginas lógicas** de tamanho fixo. No exemplo, todos os números mostrados são valores binários. A memória lógica é composta por 12 bytes. Ela foi dividida em 3 páginas lógicas de 4 bytes cada uma. O endereço lógico também é dividido em duas partes: um **número de página lógica** e um **deslocamento** dentro dessa página. No exemplo, endereços lógicos possuem 5 bits.

A memória física também é dividida em **páginas físicas** com tamanho fixo, idêntico ao tamanho da página lógica. No exemplo, a memória física é composta por 24 bytes. A página física tem o mesmo tamanho que a página lógica, ou seja, 4 bytes. A memória física foi dividida em 6 páginas físicas de 4 bytes cada uma. Os endereços de memória física também podem ser vistos como compostos por duas partes. Os 3 primeiros bits indicam um número de página física. Os 2 últimos bits indicam o deslocamento dentro da página física.

Um programa é carregado para a memória página a página. Cada página lógica do processo ocupa exatamente uma página física da memória física. Entretanto, a área ocupada pelo processo na memória física não precisa ser contígua. Durante a carga é montada uma **tabela de páginas** para o processo. Essa tabela informa, para cada página lógica, qual a

página física correspondente. Para que o programa execute corretamente, é necessário transformar o endereço lógico especificado em cada instrução no endereço físico correspondente. Isso é feito com o auxílio da tabela de páginas.

O endereço lógico gerado é inicialmente dividido em duas partes: um número de página lógica e um deslocamento dentro da página. O número da página lógica é usado como índice no acesso à tabela de páginas. Cada entrada da tabela de páginas possui o mapeamento de página lógica para página física. Já o deslocamento do byte dentro da página física será o mesmo deslocamento desse byte dentro da página lógica, pois cada página lógica é carregada exatamente em uma página física. Basta juntar o número de página física obtido na tabela de páginas com o deslocamento já presente no endereço lógico para obter-se o endereço físico do byte em questão. A Figura 2 mostra como o endereço lógico do byte Y2 é transformado no endereço físico correspondente.

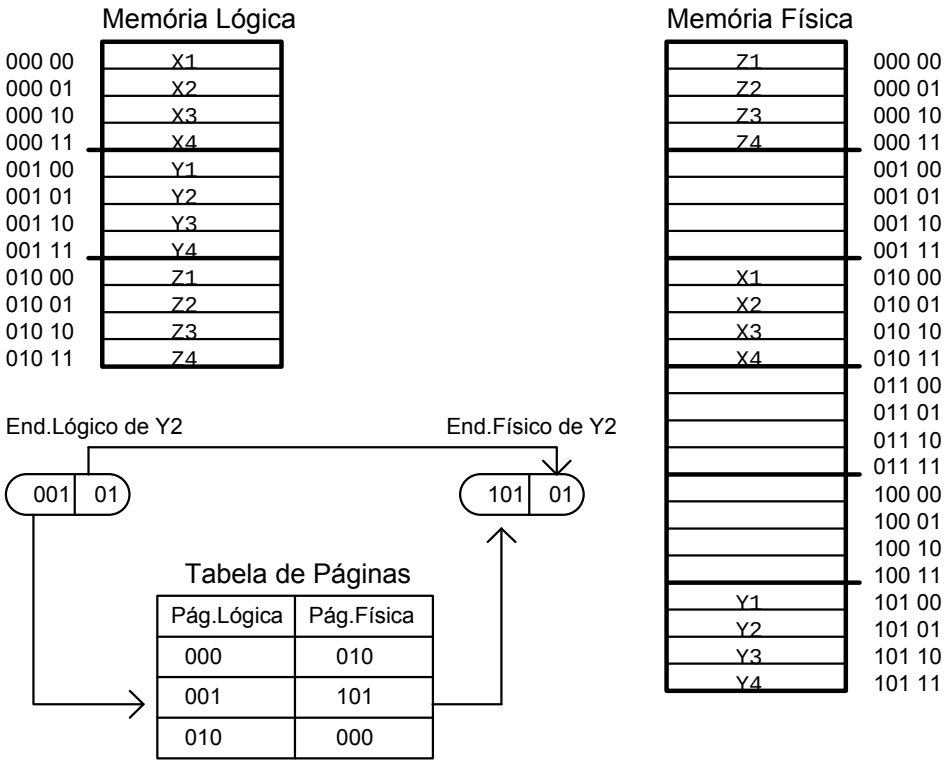

Figura 2. Mecanismo básico de paginação

Como a unidade de alocação é a página, um processo sempre ocupa um número inteiro de páginas físicas, introduzindo assim uma fragmentação interna. A gerência de memória deve manter controle das áreas ainda livres na memória. Um aspecto importante da paginação é a forma como a tabela de páginas é implementada. Observe que ela deve ser

consultada a cada acesso à memória. Uma solução é manter a tabela de páginas na própria memória. O problema desse mecanismo é que agora cada acesso que um processo faz à memória lógica transforma-se em dois acessos à memória física. Uma forma de reduzir o tempo de acesso à memória no esquema anterior é adicionar uma memória *cache* especial que vai manter as entradas da tabela de páginas mais recentemente utilizadas. Essa memória *cache* interna à MMU é chamada normalmente de ***Translation Lookaside Buffer*** (**TLB**).

Na prática, as tabelas de páginas possuem um tamanho variável, ajustado à necessidade de cada processo. Ocorre que, se as tabelas puderem ter qualquer tamanho, então teremos fragmentação externa novamente (a maior razão para usar paginação foi a eliminação da fragmentação externa). Para evitar isso, são usadas **tabelas de páginas com dois níveis**. As tabelas de páginas crescem de pedaço em pedaço, e uma tabela auxiliar chamada diretório mantém o endereço de cada pedaço. Para evitar a fragmentação externa, cada pedaço da tabela de páginas deve ter um número inteiro de páginas físicas, mantendo assim toda a alocação de memória física em termos de páginas, não importando a sua finalidade. Entradas desnecessárias em cada pedaço são marcadas como inválidas. É importante destacar que a implementação de paginação descrita aqui não é a única possível. Por exemplo, um esquema chamado **Tabela de Páginas Invertida** (*Inverted Page Table*) é usado em alguns processadores, como o PowerPC [6].

3.2 Segmentação

O conceito de página, fundamental para a paginação, é uma criação do sistema operacional para facilitar a gerência da memória. Programadores e compiladores não enxergam a memória lógica dividida em páginas, mas sim em **segmentos**. Uma divisão típica descreve um programa em termos de quatro segmentos: código, dados alocados estaticamente, dados alocados dinamicamente e pilha de execução. É possível orientar a gerência de memória para suportar diretamente o conceito de segmento. Uma posição da memória lógica passa a ser endereçada por um número de segmento e um deslocamento em relação ao início do seu segmento. Em tempo de carga, cada segmento é copiado para a memória física, e uma **tabela de segmentos** é construída. Essa tabela informa, para cada segmento, qual o endereço da memória física onde ele foi colocado e qual o seu tamanho.

Uma diferença importante é que a segmentação não apresenta fragmentação interna, visto que a quantidade exata de memória necessária é alocada para cada segmento. Entretanto, como áreas contíguas de diferentes tamanhos devem ser alocadas, temos a ocorrência de fragmentação externa. Uma solução possível é paginar cada segmento. Na **segmentação paginada** o espaço lógico é formado por segmentos, e cada segmento é dividido em páginas lógicas. O grande atrativo da segmentação está na facilidade para compartilhar memória. Cada segmento representa uma parte específica do programa, podendo ou não ser compartilhado. Segmentos tendem a ser homogêneos nesse sentido. Isto é, todo o segmento pode ser compartilhado, ou nenhuma parte do segmento pode ser compartilhada.

3.3 Paginação por Demanda

Um programa não precisa estar todo na memória para executar. Muitas partes de um programa não são necessárias todo o tempo. Por exemplo, editores de texto oferecem aos usuários funções que são raramente utilizadas. Se cada programa ocupar, a cada momento, somente a memória física que realmente necessita, haverá uma substancial economia de espaço na memória principal. A técnica conhecida como **memória virtual** permite a execução de programas que não são completamente carregados para a memória física. Tanto paginação como segmentação podem ser estendidas no sentido de prover memória virtual. Neste texto, vamos considerar apenas memória virtual implementada através da paginação por demanda, que é o método mais empregado na prática.

A **paginação por demanda** está baseada no mecanismo de paginação simples. Cada processo possui uma **memória lógica**, contígua. Essa memória lógica é dividida em **páginas lógicas** de mesmo tamanho. A **memória física** é dividida em **páginas físicas**, do mesmo tamanho das páginas lógicas. Cada página lógica é carregada em uma página física e uma **tabela de páginas** é construída.

Na paginação por demanda, apenas as páginas efetivamente acessadas pelo processo serão carregadas para a memória física. Um **bit válido/inválido** na tabela de páginas é usado para indicar quais páginas lógicas foram carregadas. Dessa forma, na paginação por demanda, uma página marcada como inválida na tabela de páginas pode significar que ela realmente está fora do espaço lógico do processo ou pode significar apenas que essa página ainda não foi carregada para a memória física. A situação exata pode ser determinada através de uma consulta ao descritor do processo em questão.

Quando a página lógica acessada pelo processo está marcada como válida na tabela de páginas, o endereço lógico é transformado em endereço físico, e o acesso transcorre normalmente. Quando a página lógica acessada pelo processo está marcada como inválida, a unidade de gerência de memória (**MMU - Memory Management Unit**) gera uma **interrupção de proteção** e aciona o sistema operacional. Cabe ao sistema operacional consultar o descritor do processo em questão. Caso a página acessada esteja fora do espaço de endereçamento do processo, o processo é abortado. Caso a página faça parte da memória lógica, mas esteja marcada como inválida apenas porque ainda não foi carregada para a memória, é dito que ocorreu uma interrupção por **falta de página** (*page fault*).

Quando o sistema operacional é acionado em função de uma falta de página, as seguintes ações devem ser realizadas:

- O processo que gerou a interrupção de falta de página é suspenso, seu descritor de processo é removido da fila do processador e inserido em uma fila especial, a "fila dos processos esperando página lógica";

- Uma página física livre deve ser alocada;

- A página lógica acessada deve ser localizada no disco;

- Uma operação de leitura do disco deve ser solicitada, indicando o endereço da página lógica no disco e o endereço da página física alocada.

A gerência do processador pode então selecionar outro processo para executar. Quando a operação de leitura do disco for concluída, a gerência de memória deve concluir o atendimento à falta de página realizando as seguintes ações:

- A tabela de páginas do processo é corrigida para indicar que a página lógica causadora da interrupção é agora válida e está na página física que fora alocada antes;

- O descritor do processo é retirado da "fila dos processos esperando página lógica" e colocado na fila do processador.

Observe que o processo deverá repetir a instrução que causou a falta de página (esta instrução não foi executada devido a falta de página). Repetir uma instrução após a carga da página faltante requer uma arquitetura de computador adequada, especialmente projetada para suportar memória virtual. Nem todos os processadores suportam esse mecanismo.

A parte do sistema operacional responsável por carregar páginas do disco para a memória principal é muitas vezes chamada de ***Pager***. É importante fazer distinção entre o *Pager* e o *Swapper*. O *Pager* carrega uma página específica de um processo e está associado com o mecanismo de memória virtual. O ***Swapper*** carrega sempre um programa inteiro do disco para a memória e vice-versa, estando associado com o mecanismo chamado *swapping*.

À medida que processos vão sendo carregados para a memória, é possível que todas as páginas físicas acabem ocupadas. Nesse caso, para atender à falta de página, será necessário antes liberar uma página física ocupada. Isso significa escolher uma página lógica que está na memória, copiar seu conteúdo de volta para o disco e marcar a respectiva página como inválida na tabela de páginas do seu processo. A página escolhida para ser copiada de volta ao disco é chamada de **página vítima**.

Vários **bits auxiliares** são adicionados à tabela de páginas, com o objetivo de facilitar a implementação do mecanismo de substituição de páginas. Embora a existência de tais bits não seja absolutamente necessária, eles tornam o mecanismo mais simples e eficiente. O **bit de sujeira** (*dirty bit*) indica se uma página foi alterada durante a execução do processo. O **bit de referência** (*reference bit*) indica se uma página foi acessada pelo processo. O **bit tranca** (*lock bit*) serve para o sistema operacional "trancar" uma página lógica na memória física.

Um algoritmo baseado em bit de referência é a **segunda chance**. Nesse caso, a gerência de memória considera que todas as páginas lógicas presentes na memória formam uma lista circular. Um apontador percorre a lista circular formada por todas as páginas e indica qual a próxima página a ser usada como vítima. O algoritmo verifica o bit de referência da página indicada pelo apontador. Caso esse bit esteja desligado, essa página é efetivamente escolhida como vítima, e o apontador avança uma posição na lista circular. Caso o bit de referência da página apontada esteja ligado, o bit de referência é desligado, e ela recebe uma segunda chance. O apontador avança uma posição na lista circular, e o procedimento é repetido para a próxima página.

O tratamento de uma falta de página é várias ordens de grandeza mais lento que um acesso normal à memória. Quando um processo possui um número muito pequeno de páginas físicas para executar, a sua taxa de falta de páginas aumenta. À medida que o número de páginas físicas diminui, a taxa de falta de páginas aumenta de tal forma que o processo pára de realizar qualquer trabalho útil. Nesse momento está ocorrendo ***thrashing***.

4 Sistema de Arquivos

O **sistema de arquivos** é a parte do sistema operacional mais visível para os usuários. Durante o tempo todo, usuários manipulam arquivos contendo textos, planilhas, desenhos, figuras, jogos, etc. Esse fato exige que o sistema de arquivos apresente uma interface coerente e simples, fácil de usar. Ao mesmo tempo, arquivos são normalmente implementados a partir de discos magnéticos. Como um acesso a disco demora cerca de 100000 vezes mais tempo do que um acesso à memória principal, são necessárias estruturas de dados e algoritmos que otimizem os acessos a disco gerados pela manipulação de arquivos. É importante observar que sistemas de arquivos implementam um recurso em software que não existe no hardware. O hardware oferece simplesmente espaço em disco, na forma de setores que podem ser acessados individualmente, em uma ordem aleatória. O conceito de arquivo, muito mais útil que o simples espaço em disco, é uma abstração criada pelo sistema operacional. Nesse caso, temos o sistema operacional criando um recurso lógico a partir dos recursos físicos existentes no sistema computacional.

Existe ampla literatura que descreve os sistemas de arquivos dos sistemas operacionais mais populares. O sistema de arquivos do UNIX System V release 2 é descrito em [2]. Em [3] também é descrita uma solução específica, utilizada no sistema operacional Xinu. A implementação utilizada no Minix é descrita em [4], inclusive com o código fonte. Em [5] é feita uma excelente descrição de diversos sistemas de arquivos utilizados no mundo UNIX. Entre eles, System V release 4, Fast File System de Berkeley, Network File System da Sun, Remote File Sharing da AT&T, Andrew File System da Carnegie-Mellon University e muitos outros.

Arquivos são recipientes que contêm dados. Cada arquivo contém dados que um usuário, por alguma razão, resolveu colocar juntos no mesmo arquivo. **Diretórios** são conjuntos de referências a arquivos. Os diretórios permitem-nos separar os arquivos em grupos, facilitando sua localização e manuseio. Existem situações nas quais é importante visualizar um único disco físico como se fossem vários. Por exemplo, o mesmo disco físico pode ser particionado em dois **discos lógicos** ou **partições**. É possível, por exemplo, manter todos os arquivos do sistema operacional (interpretador de comandos, compiladores, etc.) em uma partição e todos os arquivos de usuários na outra partição. No momento de fazer uma **cópia de segurança** (*back-up*) em fita ou CD, o particionamento do disco facilita a cópia dos arquivos do sistema e dos usuários para fitas diferentes.

Cada arquivo é identificado por um nome, o qual permite que o usuário faça referências a ele. Além do nome, cada arquivo possui uma série de outros **atributos** que são

mantidos pelo sistema operacional. Entre os mais usuais, estão: tipo do conteúdo, tamanho, data e hora do último acesso, data e hora da última alteração, identificação do usuário que criou o arquivo, lista de usuários que podem acessar o arquivo.

Diversas operações sobre arquivos são suportadas. As **operações básicas** são: criação do arquivo, destruição do arquivo, leitura do conteúdo, alteração do conteúdo, escrita de novos dados no final do arquivo, execução do programa contido no arquivo, troca do nome do arquivo, alteração na lista de usuários que podem acessar o arquivo. Em geral, essas operações básicas correspondem a chamadas de sistema que os programas de usuário podem usar para manipular arquivos. A partir das operações básicas, muitas outras podem ser implementadas. Por exemplo, as operações de impressão do conteúdo do arquivo ou a cópia de seu conteúdo para outro arquivo podem ser implementadas a partir das operações básicas de leitura e escrita.

A forma como os dados são dispostos dentro de um arquivo determina sua **estrutura interna**. Existe, na prática, uma enorme quantidade de diferentes **tipos de arquivos**, cada tipo com sua estrutura interna particular. Além disso, a cada dia, novas aplicações são criadas e, em conseqüência, novos tipos de arquivos são criados. Não é viável para o sistema operacional conhecer todos os tipos de arquivos existentes. Para o sistema operacional, cada arquivo corresponde a uma seqüência de bytes. A exceção são os arquivos contendo programas executáveis, cuja estrutura interna é definida pelo sistema operacional.

Método de acesso diz respeito à forma como o conteúdo de um arquivo é acessado. O método de acesso mais simples é o **seqüencial**. Nesse caso, o conteúdo do arquivo pode ser lido seqüencialmente, pedaço a pedaço. O acesso seqüencial é muito usado. Por exemplo, compiladores fazem uma leitura seqüencial dos programas fontes. Muitas aplicações não podem ser implementadas com o acesso seqüencial e apresentar um desempenho aceitável. No método de **acesso relativo**, o programa inclui na chamada de sistema qual a posição do arquivo a ser lida. As posições do arquivo são numeradas a partir de 0 (ou a partir de 1 em alguns sistemas), sendo que cada posição corresponde a um byte. Em muitos sistemas operacionais, existe o conceito de **posição corrente no arquivo**. Nesse caso, a chamada de sistema para leitura ou escrita não informa uma posição. Essa sempre acontece a partir da posição corrente. A posição corrente é então avançada para imediatamente após o último byte lido ou escrito.

4.1 Implementação de Arquivos

A forma básica de implementar arquivos é criar, para cada arquivo no sistema, um descritor de arquivo. O **descritor de arquivo** é um registro no qual são mantidas as informações a respeito do arquivo. Essas informações incluem os seus atributos, além de outros dados que não são visíveis aos usuários, mas que são necessários para que o sistema operacional implemente as operações sobre arquivos. A forma usual é manter o descritor de um arquivo na mesma partição onde está o seu conteúdo.

Enquanto um arquivo está sendo acessado, o seu descritor de arquivo é constantemente necessário. Entre outras coisas, o descritor é acessado a cada operação de escrita ou leitura para determinar a localização no disco dos dados a serem escritos ou lidos. Para tornar mais rápido o acesso aos arquivos, o sistema de arquivos mantém na memória uma tabela contendo todos os descritores dos arquivos em uso.

A maioria dos sistemas operacionais exige que os próprios programas informem quando um arquivo entra em uso e quando ele não é mais necessário. Para tanto, existem as chamadas de sistema `open` e `close`. Um programa deve **abrir o arquivo** antes de poder acessar seu conteúdo. Isso é feito passando o nome do arquivo como parâmetro através da chamada de sistema `open`. Também é usual passar como parâmetro o tipo de acesso que será feito, isto é, apenas leitura ("READONLY" ou "RO") ou leitura e escrita ("READWRITE" ou "RW").

O sistema de arquivos utiliza uma **Tabela dos Descritores de Arquivos Abertos** (TDAA) para manter em memória os descritores dos arquivos abertos. A TDAA mantém informações relativas aos arquivos abertos por todos os processos no sistema. Cada entrada armazena uma cópia do descritor do arquivo mantido em disco, assim como algumas informações adicionais, necessárias apenas enquanto o arquivo está aberto. Por exemplo, número de processos utilizando o arquivo no momento.

Quando o processo executando faz uma chamada de sistema `open`, o sistema de arquivos realiza as seguintes tarefas:

- Localiza no disco o descritor do arquivo cujo nome foi fornecido. Isso é feito através de uma pesquisa nos diretórios da partição. Essa operação é muitas vezes chamada de `lookup`. Não se trata de uma chamada de sistema, mas sim de uma operação interna do sistema de arquivos.

- Verifica se o arquivo solicitado já se encontra aberto. Isso é feito através de uma pesquisa na TDAA. A maioria dos sistemas operacionais mantém uma estrutura tipo tabela de dispersão (*hash table*) para localizar a entrada na TDAA correspondente a um determinado arquivo.

- Caso o arquivo ainda não esteja aberto, aloca uma entrada livre na tabela TDAA e copia o descritor do arquivo que está no disco para a entrada alocada na TDAA.

- Uma vez que o descritor do arquivo foi copiado para a memória, verifica se o processo em questão tem o direito de abrir o arquivo conforme solicitado. Isso é feito através de uma consulta aos direitos de acesso que estão armazenados no próprio descritor do arquivo.

- A partir desse momento, o arquivo está aberto e pode ser acessado. Quando um processo realiza a chamada de sistema `close`, o número de processos utilizando o arquivo em questão é decrementado na sua respectiva entrada da TDAA. Quando esse número chega a zero, o descritor do arquivo é atualizado em disco e a entrada da TDAA liberada.

Existem informações diretamente associadas com o processo que acessa o arquivo e não podem ser mantidas na TDAA pois, como vários processos acessam o mesmo arquivo, elas possuirão valores diferentes. A solução típica é criar, para cada processo, uma **Tabela de Arquivos Abertos por Processo** (TAAP). Cada processo possui a sua TAAP. Cada entrada ocupada na TAAP corresponde a um arquivo aberto pelo processo correspondente. No mínimo, a TAAP contém em cada entrada as seguintes informações: posição corrente no arquivo, tipo de acesso autorizado (apenas leitura ou leitura e escrita), apontador para a entrada correspondente na TDAA.

As operações de leitura e de escrita são realizadas através das chamadas de sistema `read` e `write`. O processo especifica o *handle* do arquivo a ser acessado, o qual aponta o respectivo descritor na memória. Existem duas funções importantes que o sistema de arquivos deve realizar na implementação das chamadas `read` e `write`: a **montagem e desmontagem de blocos lógicos** e a **localização dos blocos lógicos** no disco.

A maneira como o mapeamento entre blocos lógicos e blocos físicos é realizada depende de como é mantida a informação "onde o arquivo está no disco". A **alocação indexada** é capaz de resolver o problema do crescimento dos arquivos ao mesmo tempo que permite o acesso relativo. Na alocação indexada, cada arquivo possui uma **tabela de índices**. Cada entrada da tabela de índices contém o endereço de um dos blocos físicos que formam o arquivo. Um acesso relativo pode ser facilmente realizado através de uma consulta à tabela de índices. Para que esse acesso seja rápido, a tabela de índices é normalmente mantida na memória principal enquanto o arquivo estiver aberto. Uma forma conveniente é manter a tabela de índices dentro do próprio descritor do arquivo. Quando o arquivo é aberto, seu descritor de arquivo é copiado para a memória principal. Nesse caso, a sua tabela de índices também estaria sendo copiada.

A solução típica para compatibilizar uma maioria de arquivos pequenos com um tamanho máximo de arquivo satisfatório é empregar **níveis de indireção na indexação**. Por exemplo, suponha que o descritor de arquivos contém uma tabela de índices com 13 entradas. As primeiras 10 entradas (numeradas de 0 a 9) apontam para blocos de dados do arquivo, permitindo o acesso aos primeiros 40 Kbytes de cada arquivo (supondo blocos de 4 Kbytes). Eles são chamados de **apontadores diretos**.

A entrada 10 da tabela não aponta para um bloco de dados do arquivo, mas sim para um bloco que contém apontadores para blocos de dados. Supondo que os números de blocos físicos ocupem 4 bytes, um bloco de 4 Kbytes é capaz de armazenar 1024 apontadores. Como cada apontador representa um bloco físico, temos que um único **apontador indireto** na tabela de índices permite o acesso a 4 Mbytes de dados do arquivo. A entrada 11 da tabela contém um **apontador duplamente indireto**, o que permite o acesso a até 4 Gbytes de dados. Para suportar arquivos realmente grandes, é usada a entrada 12, a qual contém um **apontador triplamente indireto**, o que permite indexar um total de 4 Tbytes (terabytes).

4.2 Mecanismos Auxiliares na Implementação de Arquivos

Até poucos anos atrás, os sistemas operacionais incluiam um único sistema de arquivos. A partir dos anos 80 surgiram soluções que comportam a coexistência simultânea de vários sistemas de arquivos no mesmo sistema operacional. As primeiras soluções nesse sentido foram para o sistema operacional UNIX, destacando-se o "*File System Switch*" da AT&T, a arquitetura "*gnode*" da Digital Equipament Corporation e o "*Virtual File System*" da Sun. Atualmente o VFS (v*irtual file system)* é a solução mais empregada [5].

Em todas as soluções, a idéia básica é a mesma: fazer com que o sistema operacional suporte diversos sistemas de arquivos diferentes simultaneamente. As diferenças entre os sistemas de arquivos devem ser transparentes para os processos que acessam arquivos, a não ser por alguma característica específica típica do sistema de arquivos em questão (por exemplo, arquivo em CD-ROM não pode ser alterado). A solução empregada foi inspirada na gerência de periféricos, a qual suporta uma enorme diversidade de dispositivos, fornece acesso transparente e permite a instalação dinâmica de novos periféricos. Isso é conseguido separando-se a parte independente da parte dependente do sistema de arquivos em questão.

Uma importante estrutura de dados presente na implementação de um sistema de arquivos é a sua ***cache***. A *cache* não oferece funcionalidade nova, mas representa um grande aumento no desempenho de qualquer sistema de arquivos, pois o uso do disco tende a ser intenso em sistemas operacionais de propósito geral. O objetivo da *cache* é manter na memória principal uma certa quantidade de blocos do disco. Dessa forma, se algum desses blocos for requisitado, ele será encontrado na memória principal, evitando o acesso ao disco.

Uma das tarefas do sistema de arquivos é gerenciar o espaço livre nos discos. Primeiramente, muitos sistemas operacionais agrupam os setores em blocos físicos. O disco passa a ser visto como uma seqüência de blocos físicos, e não de setores. Uma forma simples de gerenciar o espaço livre em disco é através de um **mapa de bits**. Cada bit presente no mapa representa um bloco físico do disco. O espaço livre em disco também pode ser gerenciado através de uma lista contendo os números de todos os blocos físicos livres (**Lista de Blocos Livres**).

4.3 Diretórios

Os **diretórios** são as estruturas do sistema de arquivos que contêm a informação "quais arquivos existem no disco". Um diretório pode ser entendido como sendo um **conjunto de arquivos** ou um **conjunto de referências a arquivos**. Existem diversas formas de estruturar os diretórios de um sistema. A mais simples é ter um único diretório para o disco inteiro. Essa solução, conhecida como **diretório linear**, é aceitável apenas para sistemas de arquivos muito pequenos.

É possível ter no diretório principal uma entrada para cada usuário do sistema. Essa entrada não corresponde a um arquivo, mas sim a um **subdiretório** que, por sua vez, contém os arquivos do usuário correspondente. É possível estender o conceito de subdiretórios de tal

forma que os usuários também possam criar livremente os seus próprios subdiretórios. O resultado é um sistema de diretórios organizado na **forma de árvore**. Em um sistema de diretórios organizado na forma de árvore, qualquer arquivo ou subdiretório pode ser identificado de forma não ambígua através do **caminho** (*pathname*) para atingi-lo a partir da raiz da árvore. Utilizando a notação do sistema operacional UNIX, "/" representa a raiz da árvore. A subárvore pertencente ao usuário João inicia no subdiretório "/usr/joao". O usuário João possui um total de três arquivos: "/usr/joao/teste", "/usr/joao/so/trab1" e "/usr/joao/so/trab2". Facilmente, a árvore de diretórios cresce até uma altura tal que passa a ser desconfortável fornecer sempre o caminho completo até cada arquivo ou diretório. Ao mesmo tempo, a cada momento, um usuário tipicamente manipula apenas arquivos de um dado diretório. O conceito de **diretório corrente** facilita a identificação dos arquivos.

Uma questão importante é como especificar, no nome do arquivo, qual a partição de disco na qual ele se encontra. Uma possibilidade é preceder o caminho absoluto com uma identificação da partição em questão. Por exemplo, "C:\fontes\prog.c" indica, no MSDOS, o programa cujo caminho absoluto é "\fontes\prog.c" a partir da raiz da partição identificada por "C:". Uma forma alternativa é usada no UNIX. Uma partição é escolhida como principal. O diretório raiz do sistema de diretórios corresponde ao diretório raiz dessa **partição principal** ou **partição de *root***. Todas as demais partições são **montadas** em subdiretórios dessa partição principal. Através de um comando específico, o administrador do sistema faz com que um dado subdiretório permita o acesso ao diretório raiz da árvore sendo montada.

A forma mais simples de implementar diretórios e subdiretórios é considerá-los como arquivos especiais, cujo conteúdo é manipulado pelo próprio sistema operacional. Dessa forma, todo o mecanismo de alocação, liberação e localização de blocos físicos no disco, disponível para arquivos, também é usado para os diretórios.

Quando diretórios são implementados como **conjuntos de descritores de arquivos**, o conteúdo de um diretório corresponde aos descritores dos arquivos. Nesse caso, o nome do arquivo ou subdiretório faz parte do seu descritor. Nesse esquema, após a leitura do diretório, o sistema de arquivos já possui na memória todas as informações necessárias para procurar pelo nome do arquivo buscado e, caso encontre, acessar o seu conteúdo. Por outro lado, os nomes estão fortemente vinculados aos descritores de arquivos. Isso impede, por exemplo, que um mesmo arquivo possua mais de um nome. Além disso, diretórios tendem a ser maiores e estão mais sujeitos a inconsistências, pois informações importantes estão espalhadas por todo o disco.

Outra possibilidade é separar um conjunto de blocos da partição para armazenar exclusivamente os descritores de arquivos e de subdiretórios. Esse conjunto de blocos forma um **vetor de descritores**, no qual cada descritor pode ser perfeitamente identificado pelo número da partição e pela posição nesse vetor. Essa estrutura de dados forma o que é normalmente conhecido como um ***flat file system*** (na terminologia do UNIX). A estrutura em árvore é criada a partir de alguns arquivos do sistema *flat* (que na verdade atuam como subdiretórios), cada um organizado internamente como uma tabela contendo nomes e respectivos endereços no vetor de descritores. Dessa forma, para percorrer um caminho na

árvore dos diretórios, é necessário abrir o subdiretório, procurar o nome desejado, pegar o endereço *flat* associado e ler o respectivo descritor de arquivo do vetor de descritores. Esse procedimento deve ser repetido para cada nome presente no caminho.

Cada **partição** é autocontida no sentido de que todas as informações necessárias para o acesso aos seus arquivos estão contidas na própria partição. As informações incluem: diretórios e subdiretórios contidos na partição; descritores dos arquivos na partição; blocos de dados dos arquivos contidos na partição; lista ou mapa de blocos livres da partição.

5 Entrada e Saída

O objetivo primeiro de um computador é solucionar problemas. Para tanto, é necessário que algum tipo de mecanismo exista para que possamos informar esse problema ao computador e recuperar sua solução. Esse mecanismo constitui o que denominamos genericamente de dispositivos de entrada e saída. Atualmente, é possível encontrar uma grande variedade de dispositivos, desde dispositivos desenvolvidos para permitir a comunicação do homem com o computador (teclado, mouse, monitor de vídeo, etc) até dispositivos que possibilitam a comunicação entre computadores (modems, placas de redes, etc), ou ainda aqueles destinados à conexão de outros equipamentos ao computador (unidades de fita, disquetes, disco rígido, CD-ROM, etc). Apesar dessa diversidade, esses dispositivos de entrada e saída possuem alguns aspectos de hardware em comun. De acordo com o sentido do fluxo de dados entre o computador e o dispositivo, esses podem ser divididos em periféricos de entrada, periféricos de saída, ou ainda periféricos de entrada e saída. Um **periférico** pode ser visto como qualquer dispositivo conectado a um computador de forma a possibilitar sua interação com o mundo externo. Os periféricos são conectados ao computador através de um componente de hardware denominado **interface.** Considerando-se a diversidade, a complexidade, e as diferentes formas de operações em função do tipo de periférico, as interfaces empregam no seu projeto um outro componente de hardware: o **controlador**. A função básica de um controlador é traduzir operações genéricas do tipo “ler dados”, “escrever dados”, “reinicializar”, “ler status” ou “escrever comando” para uma seqüência de acionamentos eletrônicos, elétricos e mecânicos capazes de realizar a operação solicitada. Para isso, o controlador deve saber como o periférico funciona, resultando que cada tipo de periférico necessita de um controlador diferente. A gerência de E/S está intimamente relacionada com aspectos de hardware de um computador.

O mecanismo de **interrupções** permite que um controlador de periférico chame a atenção do processador. Fisicamente, o barramento de controle é usado pelos controladores de periféricos para o envio de sinais elétricos associados com a geração de uma interrupção. Uma interrupção sempre sinaliza a ocorrência de algum evento. Quando ela acontece, desvia a execução da posição atual de programa para uma rotina específica. Essa rotina, responsável por atender a interrupção, é chamada de **tratador de interrupção**. O tratador realiza as ações necessárias em função da ocorrência da interrupção. Ele é, simplesmente, uma rotina que somente é executada quando ocorre uma interrupção.

Interrupções de software (também chamadas de ***traps***) são causadas pela execução de uma instrução específica para isso. Ela tem como parâmetro o número da interrupção que deve ser ativada. O efeito é semelhante a uma chamada de sub-rotina, pois o próprio programa interrompido gera a interrupção, levando à execução do tratador correspondente. A vantagem sobre sub-rotinas é que o endereço do tratador não precisa ser conhecido pelo programa que causa a interrupção, basta usar o tipo de interrupção apropriado. Existe uma terceira classe de interrupções geradas pelo próprio processador. São as **interrupções por erro**, muitas vezes chamadas de **interrupções de exceção**. Elas acontecem quando o processador detecta algum tipo de erro na execução do programa. Por exemplo, uma divisão por zero ou o acesso a uma posição de memória que não existe.

5.1 Princípios Básicos de Software de Entrada e Saída

O objetivo primeiro do subsistema de entrada e saída é tentar padronizar ao máximo as rotinas de acesso aos periféricos de forma a reduzir o número de primitivas de entrada e saída. Tal característica também facilita a inclusão de novos dispositivos, minimizando a necessidade de alterar a interface de programação do usuário. Para atingir esse objetivo, o subsistema de entrada e saída é normalmente organizado em uma estrutura de quatro camadas, onde a camada *i* fornece abstrações de mais alto nível para a camada *i+1* do subsistema. Essa abstração é obtida através de uma interface de programação padrão, englobando uma série de operações comuns e necessárias a todos os dispositivos (figura 3).

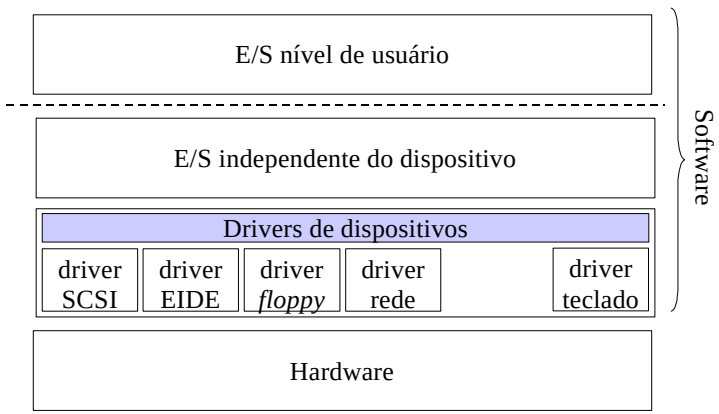

Figura 3. Estrutura em camadas do subsistema de entrada e saída

5.2 Drivers de Dispositivo

A camada inferior de software - *driver*s de dispositivos (*device drivers*) - é composta por um conjunto de módulos de software implementados para fornecer os mecanismos necessários ao acesso de um dispositivo de entrada e saída específico. O principal objetivo dos *driver*s de dispositivos é "esconder" as diferenças entre os vários dispositivos de entrada e saída fornecendo à camada superior uma "visão uniforme" desses dispositivos através de uma interface de programação única.

A camada de *driver*s de dispositivo representa uma parte significativa do sistema de entrada e saída em relação às funcionalidades. No seu nível mais baixo, ela é responsável por implementar as rotinas necessárias ao acesso e à gerência de um dispositivo específico. É nesse nível que o software de E/S realiza a programação de registradores internos de controladores que compõem a interface física dos dispositivos e implementa os respectivos tratadores de interrupção. Assim, cada tipo de dispositivo requer um *driver* apropriado.

Para atender o requisito de fornecer uma "visão uniforme", os dispositivos de entrada e saída, independente de como são interconectados às interfaces físicas (serial ou paralelo), são classificados segundo a unidade de transferência de dados em **orientados a bloco** e **orientados a caractere**. Em um **dispositivo orientado a bloco** (*block devices*), o armazenamento de informações e sua transferência entre o periférico e o sistema são realizados através de blocos de dados de tamanho fixo. Tipicamente, o tamanho de um bloco varia entre 512 bytes e 32 kbytes. Essa característica implica que o periférico de E/S e o *driver* de dispositivo estejam de acordo sobre a estrutura e o tamanho do bloco. As unidades de disco são o exemplo mais comum de dispositivos orientados a bloco. Os **dispositivos orientados a caractere** (*character devices*), por sua vez, realizam as transferências byte a byte, a partir de um fluxo de caracteres sem necessidade de considerar uma estrutura qualquer. As portas seriais são exemplos de dispositivos de E/S orientados a caractere. Essa classificação, entretanto, não é adequada, pois nem todos os dispositivos de E/S podem ser enquadrados em um desses dois grupos. Os temporizadores (relógios) e monitores de vídeo de memória mapeada são exemplos bastante comuns de dispositivos de E/S que não se enquadram em nenhuma dessas categorias.

Existe ainda um outro tipo de dispositivo denominado de **pseudo-dispositivo** (*pseudo-devices*) que na realidade não corresponde a nenhum periférico físico. Ele é apenas uma abstração empregada para adicionar funcionalidades ao sistema operacional, explorando a interface padronizada já existente para o tratamento de dispositivos. É dessa forma que o sistema operacional UNIX oferece o dispositivo nulo (*/dev/null*) para descartar dados.

5.3 E/S Independente do Dispositivo

A camada de software de E/S independente do dispositivo implementa procedimentos e funções gerais a todos os dispositivos de entrada e saída. Os principais serviços sob responsabilidade dessa camada são:

Escalonamento de E/S: Esse serviço é responsável por ordenar requisições de acesso a dispositivos de entrada e saída de forma a melhorar o desempenho total do sistema.

Denominação: Cada periférico deve possuir um nome lógico a partir do qual ele é identificado. Essa associação periférico-nome deve ter uma atribuição uniforme, independente do dispositivo, de forma a generalizar as rotinas de acesso. Um exemplo é o sistema operacional UNIX, no qual o nome de um dispositivo é um string e faz parte do sistema de arquivos.

Buferização: Um *buffer* é uma zona de memória onde dados são temporariamente armazenados enquanto eles estão sendo transferidos entre as diferentes camadas do software de E/S. Muitas vezes, a quantidade de dados que o usuário deseja ler, ou escrever, não é "natural" para o dispositivo em questão. O principal objetivo da buferização é então ajustar a velocidade e a quantidade de dados transferidos entre camadas.

***Cache* de dados:** Consiste em armazenar na memória um conjunto de dados que estão sendo freqüentemente acessados para realizar uma determinada operação, ou ainda, para serem transferidos de uma só vez a um dispositivo. O acesso a dados em memória RAM é mais rápido que em disco; por isso, esse mecanismo torna-se interessante. Como exemplo, nós podemos citar as *caches* de disco.

Alocação e liberação: Muitos dispositivos admitem, no máximo, um usuário de cada vez. O software de E/S deve então gerenciar a alocação, a liberação e o uso destes dispositivos de forma a evitar que acessos concorrentes sejam realizados. A técnica conhecida como ***spooling*** é normalmente empregada. Os pedidos são organizados em uma fila especial (*spool*), a qual é acessada por um processo especial do sistema operacional (*daemon*). O *daemon* efetua então a requisição de entrada e saída. A gerência de impressora é um exemplo clássico do emprego de *spool*.

Direitos de acesso: Nem todos os usuários podem acessar os dispositivos da mesma forma. Cabe então ao sistema operacional garantir que cada dispositivo seja acessado somente por usuários autorizados.

Tratamento de erros: O software de entrada e saída deve fornecer a capacidade de manipular erros, informando à camada superior o sucesso ou fracasso de uma operação. O tratamento de erro em si depende de seu tipo.

5.4 Entrada e Saída a Nível de Usuário

A "visão" que um usuário possui dos dispositivos de entrada e saída de um sistema é fornecida por uma interface de programação associada as bibliotecas de entrada e saída, ou aos ambientes de desenvolvimento. O fabricante do compilador de uma linguagem de programação é responsável então por implementar e fornecer rotinas que realizam entrada e saída para um determinado sistema. A interface de programação depende da linguagem em si. Por exemplo, a função `printf()` da linguagem C, que realiza saída formatada de dados.

As bibliotecas de entrada e saída não fazem parte do sistema operacional, mas estão associadas às linguagens de programação e/ou ambientes de desenvolvimento.

6 Segurança

Desde que diferentes usuários passaram a compartilhar o mesmo computador, a segurança computacional tornou-se um problema relevante. Com a informatização da sociedade, o advento da Internet, o fácil acesso a computadores remotos e a disseminação dos vírus de computador, segurança computacional tornou-se a preocupação número um de muitas pessoas. Os principais objetivos do sistema operacional com respeito a segurança é garantir a **confidencialidade** (somente usuários autorizados podem ler determinada informação), a **integridade** (informações não são modificadas sem autorização) e a **disponibilidade** (recursos permanecem disponíveis para usuários legítimos). Exemplos de violações destas propriedades são, respectivamente, ler arquivos de outros usuários sem autorização, vírus que instala-se em executáveis, formatar o disco sem autorização.

Segundo a taxonomia apresentada em [8], uma **ameaça** consiste de uma ação possível que, uma vez concretizada, produziria efeitos indesejáveis sobre os dados ou recursos do sistema. Uma **vulnerabilidade** é uma falha ou característica indevida que pode ser explorada para concretizar uma ameaça. Um **ataque** é uma ação que envolve a exploração de determinadas vulnerabilidades de modo a concretizar uma ou mais ameaças.

A **política de segurança** de uma dada instalação corresponde ao conjunto de regras que estabelecem os limites de operação dos usuários no sistema, isto é, as autorizações de cada um. Os **mecanismos de segurança** do sistema operacional são responsáveis pela implantação de uma determinada política de segurança. Por exemplo, a política de segurança típica em um sistema operacional multiusuário exige que cada usuário seja identificado univocamente pelo sistema. Para tanto, são empregados mecanismos de autenticação que podem incluir senhas, cartões magnéticos ou até mesmo dispositivos biométricos.

Neste contexto, um **sujeito** é uma entidade ativa que inicia requisições de serviços e recursos. Via de regra trata-se de um usuário ou de um processo executando em nome de um usuário. Um **objeto** é uma entidade passiva que armazena informações e/ou admite um certo conjunto de operações. Mesmo que o sistema operacional não seja construído com orientação a objetos, esses conceitos são úteis para entender a problemática da segurança computacional.

O **controle de acesso** é a mediação das requisições de acesso a objetos iniciadas pelos sujeitos. Cabe ao sistema operacional implementar um controle de acesso que imponha a política de segurança definida para a instalação em questão. As vezes a política de segurança efetivamente implantada não é aquela desejada pela administração do sistema, em função de falhas nos mecanismos de segurança ou limitações de natureza prática.

É possível classificar as diferentes soluções para o controle de acesso conforme acontece a distribuição de autoridade. No **Controle de Acesso Discricionário** o proprietário

do objeto deve determinar quem tem acesso a ele. Por exemplo, o usuário que criou o arquivo decide quem pode ter acesso a este arquivo. É o tipo de controle de acesso mais utilizado em sistemas operacionais. O **Controle de Acesso Obrigatório** baseia-se em uma administração centralizada que dita regras incontornáveis para acesso aos objetos. Por exemplo, pode-se definir **níveis de segurança** ordenados conforme a sensibilidade do objeto e a confiabilidade do sujeito (não-classificado, confidencial, secreto e ultra-secreto). Um sujeito com acesso "confidencial" jamais poderá acessar um objeto "secreto". Finalmente, o **Controle de Acesso Baseado em Papéis** requer que os direitos de acesso sejam atribuídos a papéis e não a usuários, os quais devem obter direitos através da incorporação de papéis. Leituras adicionais podem ser feitas em [9], [10] e [11].

A **Matriz de Acesso** [12] é uma forma conveniente de representar-se o controle de acesso existente em um dado sistema. A proteção do sistema é representada por uma matriz, onde as linhas correspondem aos sujeitos e as colunas correspondem aos objetos. Cada célula da matriz indica quais operações o sujeito daquela linha pode realizar sobre o objeto daquela coluna. Operações sobre a matriz são também controladas pelo mesmo mecanismo.

Na prática a matriz de acesso de um sistema multiusuário é muito grande para ser armazenada exatamente como uma matriz pelo sistema operacional. Além do que a maioria de suas células estaria vazia, pois em geral um usuário não pode acessar arquivos de outros usuários. Embora a matriz de acesso exista conceitualmente, a forma mais popular para guardar as informações é através de **listas de controle de acesso**. Cada objeto possui uma lista de controle de acesso, a qual é essencialmente a sua coluna na matriz conceitual. Os famosos bits "rwx" do Unix são uma forma simplificada de guardar esta informação.

A forma alternativa para armazenar a matriz de acesso é fazê-lo pelas linhas. Cada sujeito é associado a uma **lista de capacidades** (***capabilities***), a qual indica que operações sobre quais objetos ele tem direito. Em alguns sistemas as listas de capacidades são mantidas pelo sistema operacional (Mach [13]). Em outros, cada capacidade corresponde a uma informação criptografada que é entregue ao processo e é responsabilidade do processo apresentar esta informação no momento de acessar o respectivo objeto (Chorus [13]).

7 Organização Interna

Um sistema operacional também é um programa de computador e, como tal, possui uma especificação e um projeto. A especificação do mesmo corresponde à lista de serviços que deve executar e as chamadas de sistema que deve suportar. Por outro lado, o seu **projeto** ou ***design*** diz respeito a sua estrutura interna, como as diferentes rotinas, necessárias na implementação dos serviços, são organizadas internamente. O tamanho de um sistema operacional pode variar desde alguns milhares de linhas no caso de um pequeno núcleo de tempo real até vários milhões de linhas, como na versão 2.4 do Linux, chegando a 30 milhões de linhas no caso do Windows 2000. Embora princípios básicos como baixo acoplamento e alta coesão sejam sempre desejáveis, existem algumas formas de **organização interna** para sistemas operacionais que tornaram-se clássicas ao longo do tempo. Também

ao longo do tempo a terminologia sofreu variações. A forma como os termos são definidos neste texto procura criar uma taxonomia coerente e didática, mesmo que alguns autores, em alguns momentos, tenham usado os termos com um sentido ligeiramente diferente.

A forma mais simples de organizar um sistema operacional é colocar toda a sua funcionalidade dentro de um único programa chamado kernel. O **kernel** executa em modo supervisor e suporta o conjunto de chamadas de sistemas. Ele é carregado na inicialização do computador (*boot*) e permanece o tempo todo na memória principal. Qualquer alteração no kernel exige um novo procedimento de inicialização (um novo *boot*). Internamente, o código do kernel é dividido em procedimentos e procedimentos podem ser agrupados em módulos. De qualquer forma, tudo é ligado (*linked*) junto e qualquer rotina pode, a princípio, chamar qualquer outra rotina. O conceito de processo existe fora do kernel, mas não dentro dele, onde existe apenas um fluxo de execução. Vamos chamar este projeto de **kernel monolítico**.

Uma forma mais eficiente é executar o código do kernel com interrupções habilitadas. O **kernel monolítico interrompível** possui desempenho melhor pois os eventos associados com periféricos e temporizadores ganham imediata atenção, mesmo quando o código do kernel está executando. Entretanto, é preciso notar que tratadores de interrupções podem acessar estruturas de dados do kernel, as quais podem estar inconsistentes enquanto uma chamada de sistema é atendida. Nessa situação, a execução do tratador de interrupção poderá corromper todo o sistema. Na construção de um kernel monolítico interrompível é necessário identificar todas as estruturas de dados acessadas por tratadores de interrupção e, quando o código normal do kernel acessa estas estruturas de dados, interrupções devem ser desabilitadas (não necessariamente todas as interrupções, mas pelo menos aquelas cujos tratadores acessam as estruturas em questão). As estruturas de dados acessadas por tratadores de interrupção formam uma **seção crítica** que deve ser protegida, e o mecanismo neste caso é simplesmente desabilitar as interrupções enquanto estas estruturas são acessadas.

Vamos chamar de **kernel convencional** aquele que, além de interrompível, permite uma troca de contexto mesmo quando código do kernel estiver executando. A passagem de um kernel monolítico interrompível mas não-preemptável para um kernel convencional possui várias implicações. Agora o conceito de processo existe também dentro do kernel, uma vez que o processo pode ser suspenso e liberado mais tarde enquanto executa código do kernel. No kernel monolítico apenas uma pilha basta, pois a cada momento apenas um fluxo de execução existe dentro dele. Interrupções podem acontecer, mas ainda assim uma única pilha é suficiente. No kernel convencional um processo pode perder o processador para outro processo que também vai executar código do kernel. Logo, é necessária uma pilha interna ao kernel para cada processo, além das pilhas em modo usuário. O Linux atualmente funciona dessa forma [14].

Outra questão relevante são as estruturas de dados internas ao kernel. Elas passam a representar seções críticas e devem ser protegidas. Neste momento surgem os dois tipos de kernel convencionais. O **kernel convencional com pontos de preempção** somente suspende um processo que executa código do kernel em pontos previamente definidos do código, nos quais é sabido que nenhuma estrutura de dados está inconsistente. O desempenho deste tipo

de kernel é superior ao kernel monolítico, mas o processo de mais alta prioridade ainda deve esperar até que a execução do processo de baixa prioridade atinja um ponto de preempção.

Por outro lado, o **kernel convencional preemptável** realiza o chaveamento de contexto tão logo o processo de mais alta prioridade seja liberado. Para isto, todas as estruturas de dados do kernel que são compartilhadas entre processos devem ser protegidas por algum mecanismo de sincronização, como semáforos, mutexes ou algo semelhante. Esta solução, usada no Solaris, resulta em melhor desempenho, considerando-se as diferentes prioridades dos processos [5].

Modernamente uma organização alternativa ao kernel tem sido proposta na qual a funcionalidade típica de kernel é dividida em duas camadas. Esta solução baseia-se na existência de um **microkernel**, o qual suporta os serviços mais elementares de um sistema operacional: gerência de processador e uma gerência de memória simples. Sobre o microkernel existe um conjunto de **processos servidores** que implementam o restante da funcionalidade: gerência de periféricos, sistema de arquivos, memória virtual. Exemplos de sistemas desse tipo são o Minix [4] e o Mach [13]. Internamente, o microkernel tem a forma de um pequeno kernel monolítico. As desvantagens do kernel monolítico não são tão sérias neste caso pois o código do microkernel é pequeno. Por sua vez, cada processo servidor pode ser composto por várias *threads* e constitui um espaço de endereçamento independente.

8 Sistemas Operacionais Distribuídos

Durante a década de 1980 foram disseminadas diversas novas tecnologias, tais como microcomputadores e redes locais de computadores (**LAN - Local Area Network**). Uma conseqüência natural da existência dessas tecnologias foi a solidificação do conceito de Sistema Operacional Distribuído. Um sistema distribuído é uma coleção de computadores independentes que, para os usuários do sistema, aparecem como um computador único. Seu objetivo é prover acesso transparente a serviços e recursos distribuídos ao longo de uma rede de computadores. Para tanto, o hardware necessário consiste de máquinas autônomas interligadas por uma rede, algo fácil de ser construído. Do lado software, o desafio não tão simples é fazer os usuários perceberem todo o sistema como um computador único. Diversas aplicações poderiam fazer uso deste tipo de sistema. Desde um escritório utilizando uma rede de estações de trabalho e servidores até fábricas automatizadas e bancos [13].

Logo no início das pesquisas nesta área, foram escolhidos dois termos diferentes para denotar as duas soluções extremas para o suporte operacional. Um **Sistema Operacional de Rede** é definido como uma rede de estações de trabalho em uma LAN, cada estação tem seu próprio sistema operacional local, possivelmente diferente, com alto grau de autonomia. Neste caso os comandos são locais, mas é possível realizar operações remotas, tais como login remoto e cópia de arquivos entre máquinas.

No outro extremo do espectro, temos o **Sistema Operacional Distribuído (SOD)** propriamente dito. Ele cria a ilusão de um único sistema através da propriedade da

transparência. Ele oferece processos e atributos válidos em todo o sistema, mecanimos para sincronização de processos que são válidos não importa em que computador o processo executa, um esquema de proteção único, um sistema de arquivos com visão única e independente da localização do usuário e do arquivo. Em boa parte, o sistema operacional distribuído perfeito ainda não existe, é um objetivo a ser alcançado. Deve-se observar que, mesmo nele, algumas operações permanencem essencialmente locais, tais como gerência da memória física e escalonamento do processador local.

O termo **transparência**, associado com SOD, aceita muitas interpretações. Em geral, quanto mais transparência melhor. Temos a transparência de localização (usuário não sabe onde os recursos estão), a transparência de migração (recursos migram sem mudar o nome), a transparência de replicação (usuários não percebem existência de réplicas), a transparência de concorrência (usuários não percebem o compartilhamento com outros usuários), a transparência de paralelismo (usuário não sabe o que é feito em paralelo), entre outras.

Diversos são os serviços que um SOD pode prestar. Podemos citar: **serviço de nomes** que mapeia nomes de aplicações (ou outros nomes) para endereços de rede e de programas; **serviço de autenticação** que fornece mecanismos confiáveis para restringir o acesso a dados privados; **serviço de tempo**, capaz de manter os relógios de um conjunto de computadores em sincronismo e próximos do tempo real; **tolerância a faltas** para que um computador defeituoso não derrube todo o sistema; **execução remota**; **acesso a periféricos** remotos; **sistema de arquivos** distribuído; suporte para aplicações de **groupware**; suporte para **aplicações baseadas em RPC** (*Remote Procedure Call*) ou **objetos distribuídos**.

Na implementação de um Sistema Operacional Distribuído temos uma complexidade excessiva, uma tecnologia em rápida evolução, a possibilidade de novos serviços em função da distribuição, além da dificuldade resultante do estado global não ser conhecido. Na prática temos então a abordagem através de desenvolvedores múltiplos, e o modelo Caixa Preta é inviável. O modelo mais usado passa a ser o dos **Componentes Autônomos Cooperantes**.

Cada **Componente Autônomo Cooperante** ao mesmo tempo fornece e solicita serviços. Serviços de mais alto nível são obtidos pela composição de outros serviços. Nas soluções acadêmicas o SOD é baseado em um microkernel, o qual executa em todas as máquinas do sistema. O conjunto de serviços oferecido é selecionado por instalação.

A idéia original de SOD encontra uma realização parcial no conceito de **Middleware**. Componentes de Middleware [15] executam entre aplicação e sistema operacional local, utilizando a infraestrutura de comunicação existente. O objetivo do middleware é suportar serviços distribuídos que vão facilitar a criação de aplicações distribuídas. Desta forma, a API (*Application Program Interface*) do middleware complementa e até mesmo às vezes substitui a API do sistema operacional local, passando a definir um ambiente computacional distribuído, exatamente o papel que é esperado de um SOD. A figura 4 ilustra essa situação. O middleware procura a convivência de novas aplicações distribuídas com velhas aplicações legadas, o que facilita sua aceitação pelo mercado. Esta preocupação é menor nas soluções acadêmicas, as quais possuem maior liberdade para explorar novas idéias.

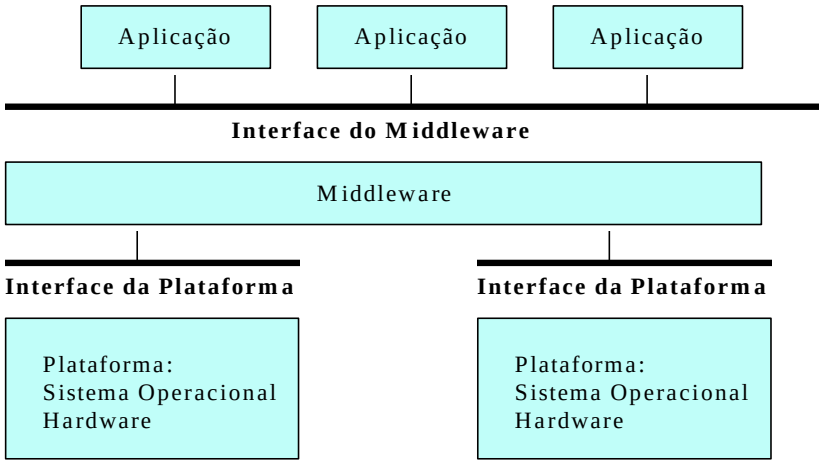

Figura 4. Middleware como a realização da idéia de SOD

Existem componentes de Middleware que fornecem serviços fundamentais, tais como: interconectividade baseada em RPC (*Remote Procedure Call*) como o DCE da Open Software Foundation (*Distributed Computing Environment* [13]) e o ONC da Sun (*Open Network Computing* [16]); interconectividade baseada em objetos como o CORBA (*Common Object Request Broker Arquitecture* [17]). Também podem fornecer serviços de mais alto nível, tais como: sistemas de arquivos distribuídos como o NFS (*Network File System*, [13]); serviço de nomes como o DNS (*Domain Name System* [18]); serviço de tempo como o NTP (*Network Time Protocol*); serviço de autenticação como o Kerberos [18]. Temos ainda middleware para processamento de transações distribuídas como o CICS da IBM [19]. Algumas soluções oferecem middleware para um grande conjunto de serviços, tais como: o CORBA da OMG, as tecnologias Java da Sun e o PontoNet da Microsoft. Por exemplo, CORBA adota o modelo orientado a objetos para implementar solicitações transparentes de serviços. Ele suporta a idéia de um sistema distribuído com múltiplos objetos que interagem apesar de habitarem uma rede heterogênea, com diferentes linguagens de programação, sistemas operacionais e arquiteturas de computador. Java faz o mesmo, mas restringe a linguagem de programação. PontoNet é uma solução proprietária da Microsoft que tenta fazer a mesma coisa.

9 Sistemas Operacionais de Tempo Real

Sistemas computacionais de tempo real são definidos como aqueles submetidos a requisitos de natureza temporal [20]. Nestes sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. Os aspectos temporais não estão limitados a uma questão de maior ou menor desempenho, mas estão diretamente associados com a funcionalidade do sistema.

Nos sistemas tempo real críticos (*hard real-time*) o não atendimento de um requisito temporal pode resultar em consequências catastróficas tanto no sentido econômico quanto em vidas humanas. Para sistemas deste tipo é necessária uma análise de escalonabilidade em tempo de projeto (*off-line*). Esta análise procura determinar se o sistema vai ou não atender os requisitos temporais mesmo em um cenário de pior caso, quando as demandas por recursos computacionais são maiores. Quando os requisitos temporais não são críticos (*soft real-time*) eles descrevem o comportamento desejado. O não atendimento de tais requisitos reduz a utilidade da aplicação mas não resulta em conseqüências catastróficas.

Sistemas operacionais de propósito geral (SOPG) encontram dificuldades em atender as demandas específicas das aplicações de tempo real. Fundamentalmente, SOPG são construídos com o objetivo de apresentar um bom comportamento médio, ao mesmo tempo em que distribuem os recursos do sistema de forma eqüitativa entre os processos e os usuários. Existe pouca preocupação com previsibilidade temporal. Mecanismos como *caches* de disco, memória virtual, fatias de tempo do processador, etc, melhoram o desempenho médio do sistema mas tornam mais difícil fazer afirmações sobre o comportamento de um processo em particular frente às restrições temporais. Aplicações com restrições de tempo real estão menos interessadas em uma distribuição uniforme dos recursos e mais interessadas em atender requisitos tais como períodos de ativação e deadlines.

Como qualquer sistema operacional, um Sistema Operacional de Tempo Real (SOTR) procura tornar a utilização do computador mais eficiente e mais conveniente. Alguns serviços são fundamentais: processos, mecanismos para a comunicação e sincronização, instalação de tratadores de dispositivos e a disponibilidade de temporizadores. A maioria das aplicações tempo real possui uma parte (talvez a maior parte) de suas funções sem restrições temporais. Logo, é preciso considerar que um SOTR deveria, além de satisfazer as necessidades dos processos de tempo real, fornecer funcionalidade apropriada para os processos convencionais, tais como sistema de arquivos, interface gráfica de usuário e protocolos de comunicação para a Internet. Em [21] é apresentado um programa de avaliação de SOTR independente de fornecedor que define vários requisitos.

Aspectos temporais estão relacionados com a capacidade do SOTR fornecer os mecanismos e as propriedades necessários para o atendimento dos requisitos temporais da aplicação tempo real. Uma vez que tanto a aplicação como o SOTR compartilham os mesmos recursos do hardware, o comportamento temporal do SOTR afeta o comportamento temporal da aplicação. Por exemplo, considere a rotina do sistema operacional que trata as interrupções do temporizador em hardware (*timer*). O projetista da aplicação pode ignorar completamente a função desta rotina, mas talvez não possa ignorar o seu efeito temporal, isto é, a interferência que ela causa na execução da aplicação.

O fator mais importante a vincular aplicação e sistema operacional são os serviços que este último presta. A simples operação de solicitar um serviço ao sistema operacional através de uma chamada de sistema significa que: (1) o processador será ocupado pelo código do sistema operacional durante a execução da chamada de sistema e, portanto, não poderá executar código da aplicação; (2) a capacidade da aplicação atender aos seus

deadlines passa a depender da capacidade do sistema operacional em fornecer o serviço solicitado em um tempo que não inviabilize aqueles deadlines.

Com respeito ao comportamento temporal do sistema, qualquer análise deve considerar conjuntamente aplicação e sistema operacional, pois os requisitos temporais que um SOTR deve atender estão completamente atrelados aos requisitos temporais da aplicação tempo real que ele deverá suportar. Uma vez que existe um amplo espectro de aplicações de tempo real, também existirão diversas soluções possíveis para a construção de SOTR, cada uma mais apropriada para um determinado contexto. Por exemplo, o comportamento temporal exigido de um SOTR capaz de suportar o controle de vôo em um avião (*fly-by-wire*) é muito diferente daquele esperado de um SOTR usado para videoconferência.

9.1 Tipos de Suportes para Tempo Real

A diversidade de aplicações de tempo real gera uma diversidade de necessidades com respeito ao suporte para tempo real, a qual resulta em um leque de soluções com respeito aos suportes disponíveis, com diferentes tamanhos e funcionalidades. De uma maneira simplificada podemos classificar os suportes de tempo real em dois tipos: núcleos de tempo real (NTR) e sistemas operacionais de tempo real (SOTR). O NTR consiste de um pequeno microkernel com funcionalidade mínima mas excelente comportamento temporal. Seria a escolha indicada para, por exemplo, o controlador de uma máquina industrial. O SOTR é um sistema operacional com a funcionalidade típica de propósito geral, mas cujo kernel foi adaptado para melhorar o comportamento temporal. A qualidade temporal do kernel adaptado varia de sistema para sistema, pois enquanto alguns são completamente reescritos para tempo real, outros recebem apenas algumas otimizações. Por exemplo, o sistema Solaris [5] implementa funcionalidade Unix, mas foi projetado para fornecer boa resposta temporal.

A figura 5 procura resumir os tipos de suportes encontrados na prática. Esta é uma classificação subjetiva, mas permite entender o cenário atual. Além do NTR e do SOTR descritos antes, existem outras duas combinações de funcionalidade e comportamento temporal. Obter funcionalidade mínima com pouca previsibilidade temporal é trivial, qualquer núcleo oferece isto. Por outro lado, obter previsibilidade temporal determinista em um sistema operacional completo é muito difícil e objeto de estudo pelos pesquisadores das duas áreas. Entretanto, é razoável supor que existirão sistemas deste tipo no futuro.

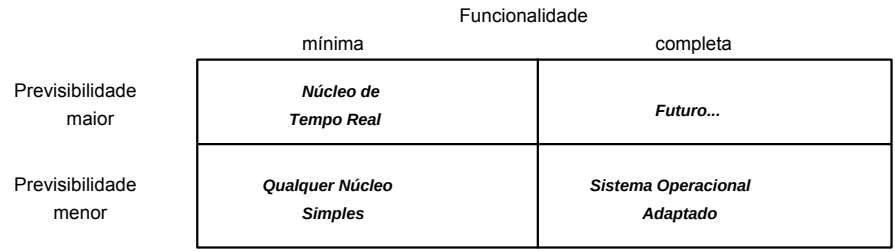

Figura 5. Tipos de suportes para aplicações de tempo real

Posix é um padrão para sistemas operacionais, baseado no Unix, criado pelo IEEE (Institute of Electrical and Eletronic Engineers). Posix define as interfaces do sistema operacional mas não sua implementação. Muitos SOTR atualmente já suportam a API do Posix, como pode ser constatado através de uma visita às páginas listadas em http://www.cs.bu.edu/pub/ieee-rts. Uma descrição completa do Posix é capaz de ocupar um livro inteiro [22]. Uma descrição detalhada do escalonamento em várias versões do Unix pode ser encontrada em [5].

Existem vários projetos envolvendo adaptações do Linux para tempo real. O RT-Linux (http://luz.cs.nmt.edu/~rtlinux/) é um sistema operacional no qual um microkernel de tempo real co-existe com o kernel do Linux. O objetivo do projeto RED-Linux (http://linux.ece.uci.edu/RED-Linux/) é fornecer suporte de escalonamento tempo real para o Linux, através da integração de escalonadores baseados em prioridade, baseados no tempo e baseados em compartilhamento de recursos [23]. O KURT-Linux (http://hegel.ittc.ukans.edu/projects/kurt/), ou "Kansas University Real Time Linux Project" é um sistema operacional que permite o escalonamento explícito de qualquer evento [24].

Agradecimentos

Os autores agradecem ao Rafael R. Obelheiro pelas várias sugestões e comentários.

Referências

[1] Oliveira, Rômulo Silva de, Carissimi, Alexandre da Silva e Toscani, Simão Sirineo. *Sistemas Operacionais.* Editora SagraLuzzatto, ISBN 85-241-0643-3, 2001.

[2] Bach, M. *The Design of the Unix Operating System.* Prentice-Hall, 1986.

[3] Comer, D. *Operating System Design: The Xinu Approach.* Prentice-Hall, 1984.

[4] Tanenbaum, A. S., Woodhull, A. S. *Operating Systems Design and Implementation.* 2nd edition. Prentice-Hall, 1997.

[5] Vahalia, U. *Unix Internals: The New Frontiers.* Prentice-Hall, 1996.

[6] Jacob, B., Mudge, T. *Virtual Memory: Issues of Implementation.* IEEE Computer, pp. 33-43, June 1998.

[7] Jacob, B., Mudge, T. *Virtual Memory in Contemporary Microprocessors.* IEEE Micro, pp. 60-75, July-August 1998.

[8] Obelheiro, Rafael R. *Modelos de Segurança Baseados em Papéis para Sistemas de Larga Escala: A Proposta RBAC-JaCoWeb.* Dissertação de mestrado, Programa de Pós-Graduação em Engenharia Elétrica da UFSC, 2001.

[9] Amoroso, Edward G. *Fundamentals of Computer Security Technology*. Prentice-Hall PTR, Upper Saddle River – NJ, 1994.

[10] Sandhu, Ravi S. and Samarati, Pierangela S. *Access Control: Principles and Practice.* IEEE Communications, 32(9):40-48, September 1994.

[11] Sandhu, Ravi S. and Samarati, Pierangela S. *Authentication, Access Control, and Audit.* ACM Computing Surveys, 28(1):241-243, March 1996.

[12] Lampson, Butler W. *Protection.* In Proceedings of the 5th Princeton Symposium on Information Sciences and Systems, pp. 437-443, Princeton University, March 1971.

[13] Tanenbaum, Andrew S. *Distributed Operating Systems.* Prentice-Hall, ISBN 0-13-219908-4, 1995.

[14] Bovet, Daniel P. and Cesati, Marco. *Understanding the Linux Kernel.* O'Reilly & Associates, ISBN 0596000, 2000.

[15] Geihs, Kurt. *Middleware Challenges Ahead.* IEEE Computer, Vol. 34, no. 6, pp. 24-31, June 2001.

[16] Corbin, John R. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls.* Springer-Verlag, 1991.

[17] Vinoski, Steve. *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.* IEEE Communications, 35(2):46-55, February 1997.

[18] Coulouris, G., Dollimore, J. and Kindberg. *Distributed Systems: Concepts and Design.* 2nd edition. Addison-Wesley, ISBN 0-201-62433-8, 1994.

[19] Horswill, John. *Designing and Programming CICS Applications.* O'Reilly & Associates, ISBN 1565926765, 2000.

[20] Farines, Jean-Marie, Fraga, Joni da Silva e Oliveira, Rômulo Silva de. *Sistemas de Tempo Real.* 12ª Escola de Computação, IME-USP, São Paulo-SP, julho de 2000.

[21] Timmeman, M., Beneden, B. V. and Uhres, L. *RTOS Evaluation Kick Off.* Real-Time Magazine, 1998-Q33, http://www.realtime-info.be, (atualmente Dedicated Systems Magazine), 1998

[22] Gallmeister, B. O. *POSIX.4 Programming for the Real World*. O'Reilly & Associates, ISBN 1-56592-074-0, 1995.

[23] Wang,Y.-C., Lin, K.-J. *Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel*. Proc. of the Real-Time Systems Symp., Dec 1999.

[24] Srinivasan, B., Pather, S., Hill, R., Ansari, F. and Niehaus, D. *A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software.* Proc. of the Real-Time Technology and Applications Symp., June 1998.